

Programmering 1

Högskolan på Åland
Joakim Isaksson

Allmän kursinformation

- Yrkesstudier inom studieprogrammet i informationsteknik
- Lärare: Joakim Isaksson (lektor, diplomingenjör)
- Tidpunkt: vecka 35 – 46
- Ingen schemalagd undervisning vecka 42
- Kurstentamen torsdagen den **16.11**
 - Omtentamen 9.12, XX.1
(mitten av januari)

Allmän kursinformation

- Omfattning: **8** studiepoäng, schemalagd undervisning 128h + tentamen
- 1,5 sp motsvarar i genomsnitt 40h arbete för den studerande
=> 8 sp motsvarar ca 210 h
=> *i medeltal* förväntas du lägga ner ca 80h självständigt arbete på kursen!
- Kursen hemsidor på Moodle:
<http://pingu.ha.ax/moodle>
 - Föreläsningsmaterial, uppgifter

Allmän kursinformation

- Fokus på grundläggande programmeringsbegrepp:
 - Datatyper och variabler
 - Flödeskontroll
 - Iteration
 - Funktioner
 - Enkla datastrukturer: Räckor, structs
- Också en del
 - Filhantering
 - Minneshantering

Allmän kursinformation

- Programmeringsspråk: C
- Programmeringsmiljö: Valfri, men under laborationer och handledningstillfällen används skolans Linux-miljö
- Vad som *inte* ingår i denna kurs:
 - Grafisk programmering
 - Objektorienterad programmering
 - Hårdvaruspecifik programmering

Kursens arbetsformer

- Föreläsningar
- Handledda laborationer
- Inlämningsuppgifter (med handledningstillfällen)
 - Lösningarna på inlämningsuppgifter skall lämnas in enligt på förhand fastslagna deadlines
- Kurstentamen

Bedömning

- Kursvitsordet bestäms enligt följande:
 - Poäng från inlämningsuppgifter **30%**
 - Poäng från tentamen **70%**
- För godkänd kurs (**G**) krävs minst 50% totalt samt minst 40% från varje delmoment
- För **VG** krävs minst 75% totalt samt minst 60% från varje delmoment
- För godkänd kurs krävs dessutom att **alla** laborationer är redovisade & godkända

Bedömning, exempel

- Kalle Kavat har skrapat ihop följande poäng:
 - 40/50 poäng från inlämningsuppgifterna (80%)
 - 22/30 poäng från tentamen (73%)
- Totalprocenten blir $0.80 \cdot 30 + 0.73 \cdot 70 = 75,1\%$
> 75% och villkoret “minst 60% från alla delmoment” är uppfyllt

=> Kalle får VG i kursen

Råd och tips

- Öva, öva, öva!
- Utnyttja handledningstillfällena effektivt, börja jobba med uppgifterna redan **före** handledningen
- Ta hjälp av läraren, tutorer, klasskamrater
- **Men** se till att du alltid själv förstår hur din lösning fungerar

Kopiering av kod

- Att kopiera andras program “rakt av” eller med endast kosmetiska förändringar räknas som plagiering
- Att hämta inspiration och idéer från andras lösningar är däremot både godkänt och att rekommendera
- Direkt kopiering av små detaljer kan också vara ok OM du inkluderar en källreferens i din kod.
- Ibland svår gränsdragning – fråga läraren om du är osäker

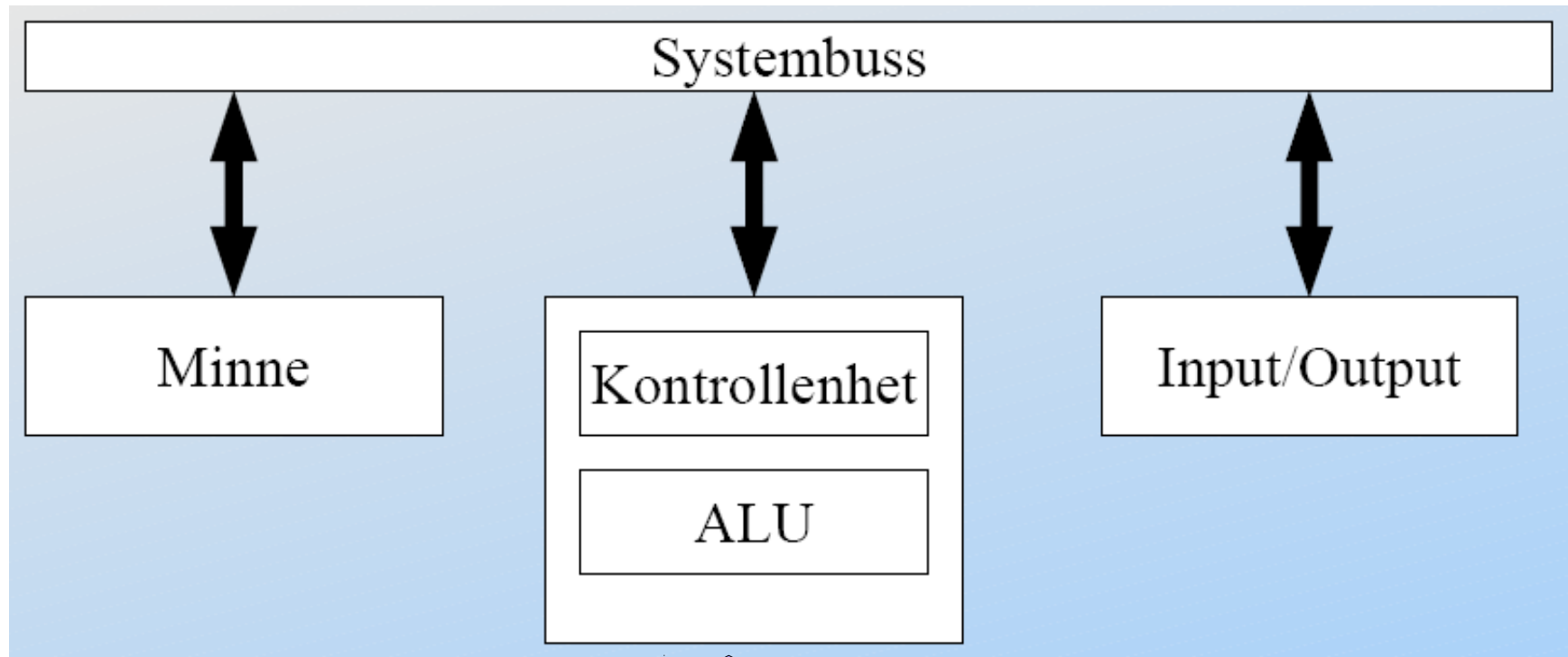
Grundläggande definitioner

- *Dator*: En programmerbar maskin som tar emot indata, utför operationer på datan och presenterar resultatet (=utdata) för användaren.
- *Algoritm*: En allmän procedur eller formel för att lösa ett problem: Sortera en lista, lös ett matematiskt problem...
- *Program*: En serie instruktioner som berättar för en dator vad som ska göras med indatan. Nära relaterat till algoritmbegreppet, kan ses som en specifik *implementation* av en algoritm

Hårdvaruarkitektur

- En typisk dator består av
 - centralenheter (processor, CPU)
 - innehåller (mycket förenklat) en kontrollenhet samt en aritmetisk-logisk enhet (ALU)
 - minne
 - enheter för in- och utmatning av data (input/output)
 - delsystemen förenas genom *systembussar*

Von Neumann-arkitekturen



(Mycket förenklad) processor

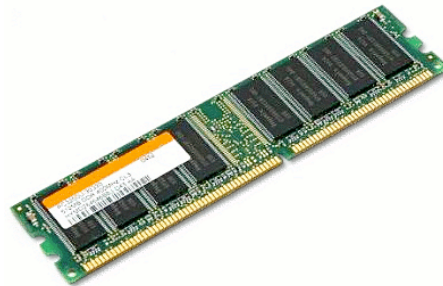
Minnet

- Innehåller programinstruktioner och lagrad data
- Olika typer av minnen

Massminnen
(ex. hårddskiva)



RAM (Centralminne)



Cacheminne



Långsammare/billigare

Snabbare/dyrare

Programkörning

- Ett program finns lagrat i ett massminne
 - Består av instruktioner samt lagrad data
- Före körningen måste programinstruktionerna kopieras till centralminnet
- Under körningen hämtas instruktionerna, en i taget, till processorn

Processorn

- Hämtar, dekodar och utför programinstruktioner
 - Flyttar data mellan olika typer av minnen samt från och till I/O enheterna
 - ALU:n utför aritmetiska och logiska operationer
- Ett program för en viss processor (eller processorfamilj) kan endast använda sådana instruktioner som just denna processor förstår - "Instruction set"
- Instruktioner för Intel x86-processorer:
http://en.wikipedia.org/wiki/X86_instruction_listings

Maskinkod och portabilitet

- För processorn är ett program endast en lång följd av siffror (egentligen endast ettor och nollor, men programmet nedan är skrivet i “hexadecimalt” format för att spara utrymme)
- Detta program skriver ut en rad text på skärmen, och kan endast köras på den processor (x86) som det är avsett för
- En instruktion består av en operationskod samt parametrar / operander för denna operation

Data segment:

00000000	48	65	6c	6c	6f	20	77	6f
00000008	72	6c	64	21	0a			

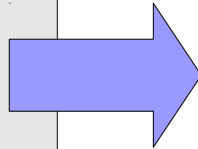
Code segment:

00000000	b8	04	00	00	00
00000005	bb	01	00	00	00
0000000a	b9	00	00	00	00
0000000f	ba	0d	00	00	00
00000014	cd	80			
0000000a	b8	01	00	00	00
0000000f	bb	00	00	00	00
00000014	cd	80			

Assembler

- Genom att använda överenskomna förkortningar för de olika operationskoderna samt “etiketter” för att namnge vissa dataområden kan maskinkoden göras mera lättläst
=> *Assemblerkod*
- Assemblerinstruktionerna måste omvandlas till maskinkod innan programmet kan köras

```
section .data
    string: db 'Hello world!',10
    length: equ $-string
section .text
    global _start
_start:
    mov eax,4
    mov ebx,1
    mov ecx,string
    mov edx,length
    int 80h
    mov eax,1
    mov ebx,0
    int 80h
```



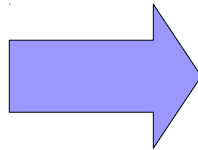
```
Data segment:
00000000  48 65 6c 6c 6f 20 77 6f
00000008  72 6c 64 21 0a

Code segment:
00000000  b8 04 00 00 00
00000005  bb 01 00 00 00
0000000a  b9 00 00 00 00
0000000f  ba 0d 00 00 00
00000014  cd 80
0000000a  b8 01 00 00 00
0000000f  bb 00 00 00 00
00000014  cd 80
```

Högnivåspråk och abstraktion

- Assemblerkod ger total kontroll över processorn, och (i teorin) den bästa prestandan, men att skriva komplexa program i assembler är mycket tidskrävande, och samma portabilitetsproblem som för den rena maskinkoden kvarstår
- Med ett *högnivåspråk* kan vi beskriva vad vårt program skall utföra på en högre *abstraktionsnivå*, utan att behöva ta ställning till hårdvarudetaljerna

```
string = 'Hello World!'
length = 13
output (string, length)
```



```
section .data
    string: db 'Hello world!',10
    length: equ $-string
section .text
    global _start
_start:
    mov eax,4
    mov ebx,1
    mov ecx,string
    mov edx,length
    int 80h
    mov eax,1
    mov ebx,0
    int 80h
```

Tolkade och kompilerade språk

- För en processor är ett program skrivet i ett högnivåspråk obegripligt
- En *tolk* översätter programmet till maskinkod *varje gång* det körs
 - Exempel på tolkade språk: Python, Perl
- En *kompilator* omvandlar programmet till maskinkod *en gång*. Resultatet av kompileringen är en binärfil som kan köras utan kompilatorns hjälp
 - Exempel på kompilerade språk: C, C++, Java
- Genom att använda tolkar och kompilatorer för olika processorer kan samma högnivåkod omvandlas till maskinkod för olika hårdvaruomgivningar

Introduktion till C – kort historia

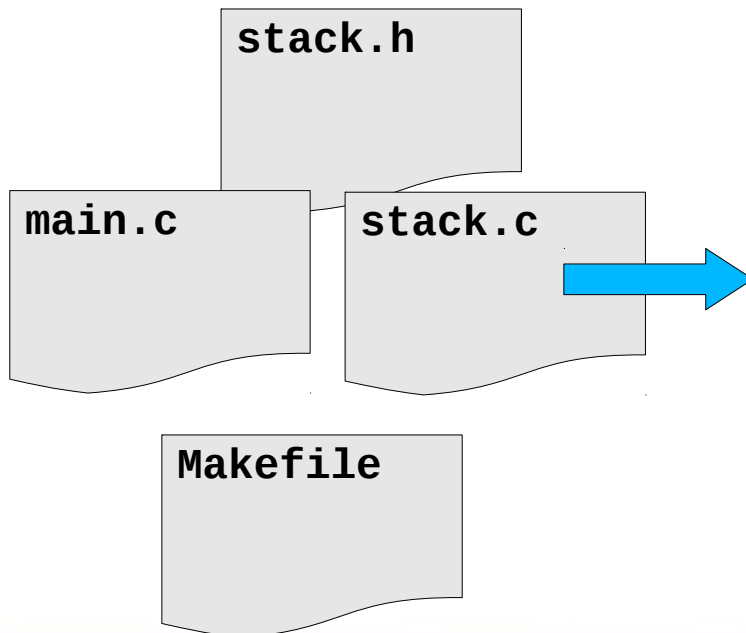
- Skapades i början av 1970-talet
- Ökade i popularitet i takt med operativsystemet Unix, vars “standardspråk” var/är C
- Ligger till grund för objektorienterade språk som C++ och Java
- “C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments.”
 - Dennis M. Ritchie (1941 - 2011),
The development of the C language

Varför C?

- Behärskar man C är det (relativt) lätt att lära sig andra “vanliga” programmeringsspråk
- Kompilatorer för så gott som alla datorarkitekturer
=> Mycket *portabelt*
- Små och snabba program
- Lämpligt såväl för hårdvarunära som mera abstrakta uppgifter

Källkod

- Program i C skrivs i textform – vilken texteditor som helst kan användas för detta ändamål
- Olika typer av filer:
 - **Källkod (*.c), headerfiler (*.h), makefiler**



```
stack.c — Kate
File Edit View Projects Bookmarks Sessions Tools Settings Help
New Open Save Save As Close Undo Redo
Documents
stack.c
1  #include "stack.h"
2
3  int stack_push(stack* s, int val)
4  {
5      if (s->top >= STACK_MAX_SIZE)
6      {
7          return -1;
8      }
9
10     s->array[s->top++] = val;
11     return 1;
12 }
```

Line 7, Column 18 INSERT Soft Tabs: 4 (8) UTF-8 C

Search and Replace Terminal

Kompilering av program

- En kompilator omvandlar källkodsinstruktioner till maskinkod för en viss datorarkitektur
 - T.ex. C-kod => maskinkod för x86- eller x64-processorer
- I Linuxmiljön använder vi oss av kompilatorn **gcc**
- Enklaste scenariot: Kompilatorn läser in **en** källkodsfil och skapar en körbar fil
- Mera realistisk: Ett antal egna filer läses in, kompileras, och *länkas* till yttre *programbibliotek*.
- Kompilatorn kan instrueras att *optimera* programmet med avseende på t.ex. storlek eller prestanda

Programbibliotek

- Ineffektivt att uppfinna hjulet på nytt, dvs. implementera all funktionalitet som behövs “from scratch”
- Kan istället använda färdiga *programbibliotek* för t.ex. input/output, matematiska beräkningar, minneshantering...
=> *Återanvändning* av kod (“Code reuse”)
- Biblioteken kan vara kommersiella eller fritt tillgängliga
- Under denna kurs kommer vi endast att använda oss av C:s standardbibliotek (ingår t.ex. i **gcc**)

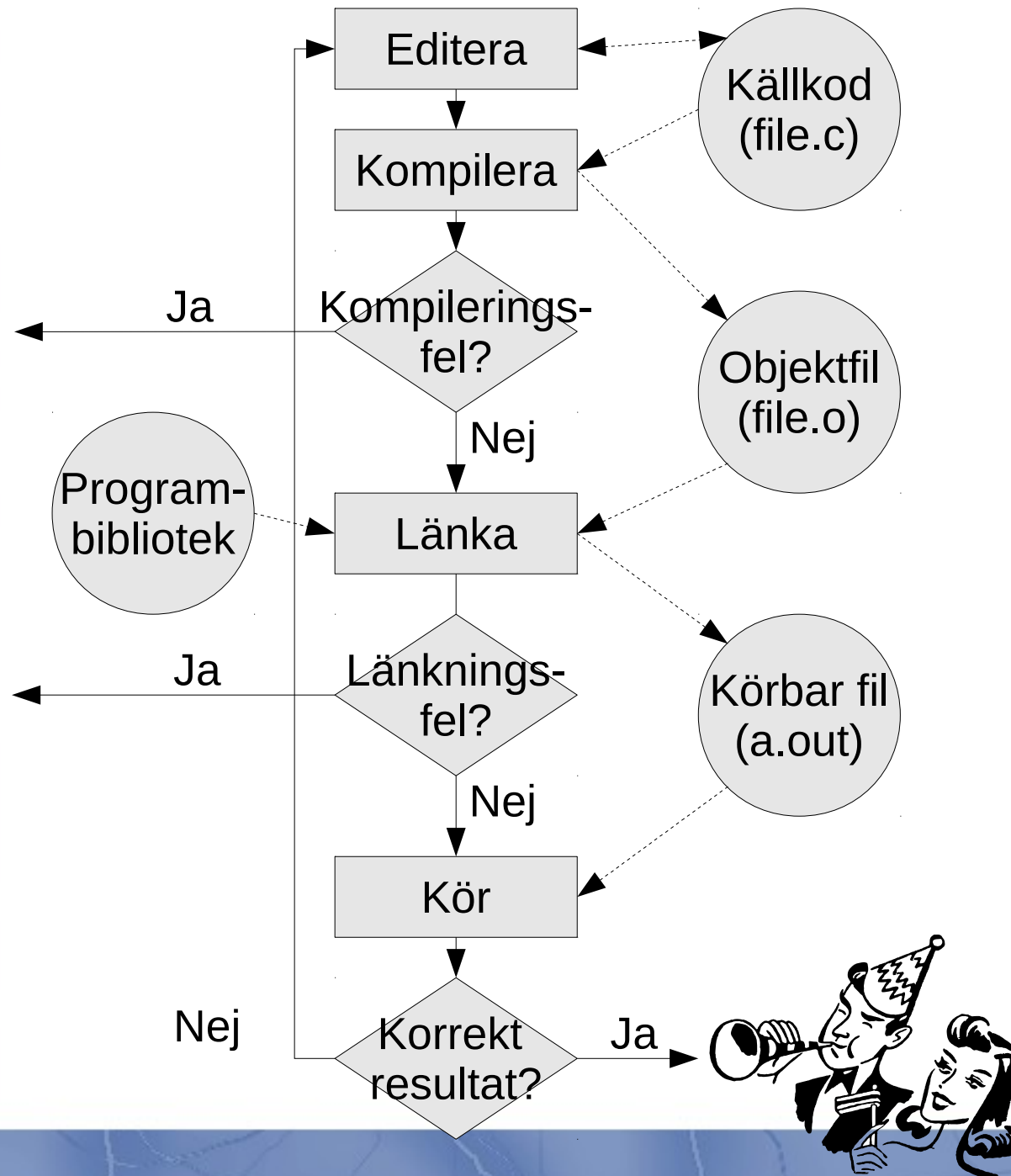
“Programmeringsprocessen”

Exempel på kommandon i Linux-miljö:

Editera:
kate file.c

Kompilera & länka:
gcc file.c

Köra programmet:
./a.out



Vårt första program i C

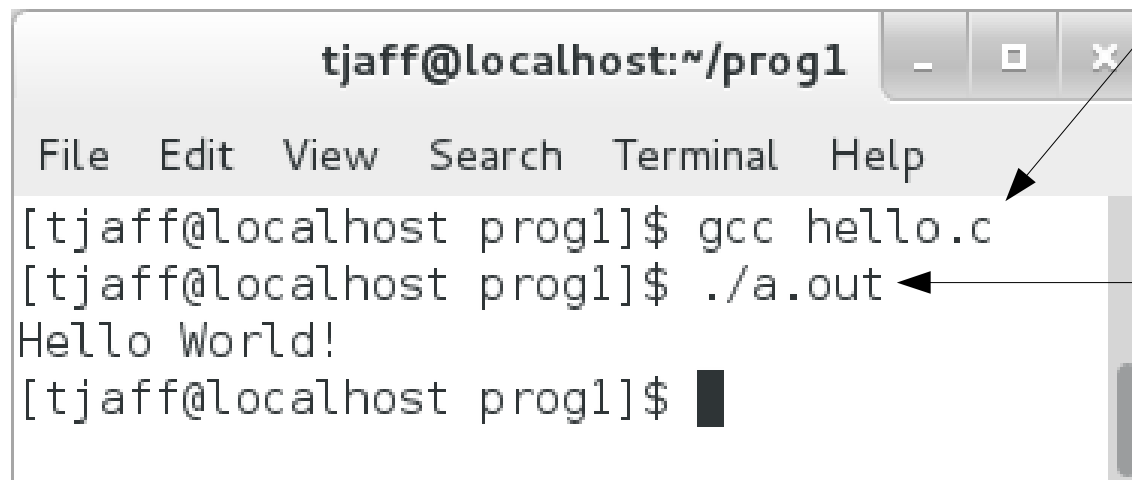
```
#include <stdio.h>

int main(void)
{
    printf("Hello World!\n");

    return 0;
}
```

Denna fil editeras i en text-editor och lagras under namnet **hello.c**

Programmet *kompileras*...



The screenshot shows a terminal window titled 'tjaff@localhost:~/prog1'. The menu bar includes 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The command history shows the user compiling 'hello.c' with 'gcc' and then running the resulting executable 'a.out'. The output of the program is 'Hello World!'.

```
tjaff@localhost:~/prog1
File Edit View Search Terminal Help
[tjaff@localhost prog1]$ gcc hello.c
[tjaff@localhost prog1]$ ./a.out
Hello World!
[tjaff@localhost prog1]$
```

...vilket resulterar i en *körbar fil* **a.out**

Vårt första program i C

“Preprocessor directive” som instruerar kompilatorn att inkludera information från C:s standardbibliotek

```
#include <stdio.h>

int main(void)
{
    printf("Hello World!\n");

    return 0;
}
```

Detta är en funktion som heter **main**.

Den returnerar ett värde av typ **int** och accepterar inga parametrar (“**void** of parameters”)

Viktigt: Varje körbart program måste innehålla en funktion som heter **main** – detta är programmets startfunktion (“entry point”)

Vårt första program i C

printf är en funktion i C:s standardbibliotek som skriver ut data till systemets standard output

```
#include <stdio.h>

int main(void)
{
    ▶ printf("Hello World!\n");

    return 0;
}
```

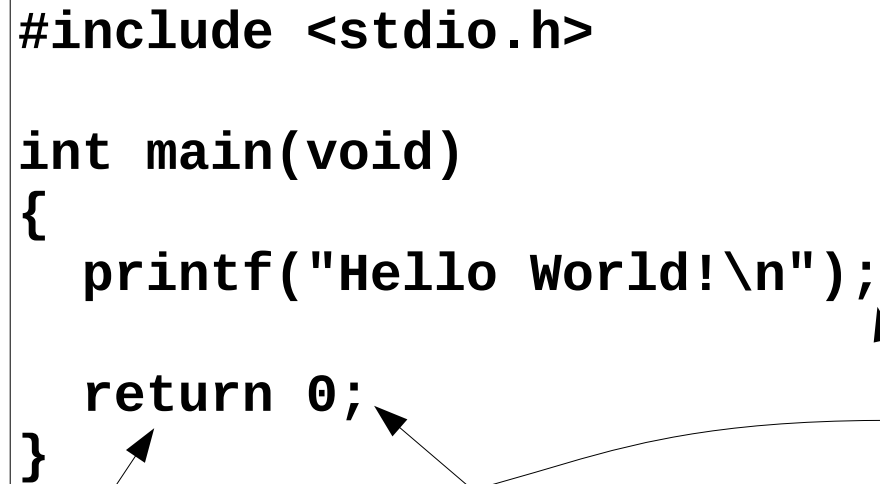
I motsats till **main** accepterar **printf** parametrar – i detta fall en teckensträng som inleds och avslutas med tecknet "

“Vågparenteserna” (*curly braces*) { och } visar var funktionen börjar och slutar. De kan också användas för att skapa *kodblock* inne i en funktion.

Vårt första program i C

```
#include <stdio.h>

int main(void)
{
    printf("Hello World!\n");
    return 0;
}
```



return avslutar körningen av funktionen och returnerar ett värde (=0) till operativsystemet. En nolla innebär att programmet avslutades normalt.

Varje instruktion måste avslutas med ett semikolon (;)

Praktiska råd och tips

- Planera programmet innan du börjar skriva kod
- Testkör ofta
- Gör säkerhetskopior
- Använd en enhetlig indentering
- Kommentera din kod
- Använd vettiga variabelnamn
- Gå igenom och korrigerar kompilatorns felmeddelanden en i taget, **uppifrån och ner**
- Sätt in extra utskrifter för debuggning

Kommentering av kod

- Till god programmeringsstil hör att *kommentera* sina program
- Gör det lättare för utomstående, och dig själv (efter några veckor eller månader) att förstå hur ditt program fungerar
- Kompilatorn ignorerar innehållet i kommentarerna
- En kommentar kan
 - inledas med `//` och avslutas med **radbyte**
 - inledas med `/*` och avslutas med `*/`
=> kan sträcka sig över flera rader

Kommentarer

```
#include <stdio.h>

/* This is a comment which
   spans several rows.
*/
int main(void)
{
    // This is a one-line comment.
    printf("Hello World!\n");

    return 0;
}
```

Obs: Kompilatorn ignorerar också extra radbyten och mellanslag

Kommentarer

```
#include <stdio.h>

/* Entry point for the application - OK(?)
   Define main() - NOT OK
*/
int main(void)
{
    // Output a friendly greeting - OK(?)
    // Use printf() - NOT OK
    printf("Hello World!\n");

    return 0;
}
```

Skriv vettiga kommentarer -
undvik självklarheter!

Källhänvisningar/referenser

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

int main(void)
{
    // Initialization of the random number generator
    // according to http://tinyurl.com/nh9xtas
    srand(time(NULL));

    printf("Welcome to the lotto generator);
    // Actual randomization of numbers
    // (using your own code) here...
    return 0;
}
```

Variabler

- Ett av de viktigaste och mest grundläggande begreppen inom programmering
- Kan ses som en namngiven plats i datorns minne där data kan lagras
- Innehållet kan modifieras under programkörningen
- Innehållet kan t.ex. skrivas ut genom en **printf**-instruktion
- Olika *typer* av variabler – i våra första exempel använder vi variabler som innehåller *heltal*

Deklaration av variabler

- En variabel måste *deklareras* innan den kan användas i ett program
- Berättar för kompilatorn vad variabeln skall heta samt vilken typ av data den skall innehålla
- Med hjälp av *tilldelningsoperatoren* = kan en variabel tilldelas ett värde
 - Kan göras både i samband med deklARATIONEN och senare under programkörningen
- En variabel kan användas endast i det kodblock där den är deklarerad

Variabler, exempel

```
#include <stdio.h>

int main(void)
{
    /* Declare a variable of type integer, name it sum and
       initialize it to zero */
    int sum = 0;

    /* Change the value of sum, just to show that we can */
    sum = 10 + 20;

    /* Output the value of sum using the printf function.
       The format specifier %d tells printf to output the
       value of an integer (digit) variable as part of the string.
       This variable is then sent as the second parameter
       to printf */
    printf("The sum of 10 and 20 is %d\n", sum);

    return 0;
}
```

Namngivning av variabler

- Ett variabelnamn måste inledas med en bokstav eller “underscore”-tecknet _
- Det inledande tecknet kan följas av valfri följd bokstäver, siffror och underscores
- Specialtecken är inte tillåtna
- *Reserverade ord* är inte tillåtna – en variabel kan t.ex. inte heta **int** eller **return**
- Se kapitel 1.2.2 i Appendix A i kursboken för en lista över reserverade ord i C

Datatyper för heltal

- Datatypen **int** används för att representera heltal
- En **int** är garanterad att inrymma minst 32 *bitar* (= 4 bytes) information
=> heltalsvärden inom intervallet (cirka) ± 2 miljarder
- En **unsigned int** representerar endast positiva värden => heltalsvärden mellan 0 och ca 4 miljarder
- I C ges inga garantier för exakt hur många bytes en viss datatyp motsvarar – operatorn **sizeof** kan användas för att kontrollera den exakta storleken.

Dat typer för decimaltal

- Decimaltal representeras i C med datatyperna **float** och **double** – kallas även för “flyttal”
 - **float** motsvarar typiskt 32 bitar
 - **double** motsvarar typiskt 64 bitar
- **Viktigt:** Decimaltal kan i allmänhet inte representeras exakt i en dators minne
- Ju flera bitar som finns tillgängliga, desto noggrannare representation
- **Rekommendation:** Använd **float** endast i undantagsfall
- **Rekommendation:** Föredra heltal framför flyttal

Flyttalsvariabler, exempel

```
#include <stdio.h>

int main(void)
{
    /* Declare a variable of type double, name it sum and
       initialize it to zero */
    double sum = 0;

    /* Change the value of sum */
    sum = 10.5 + 20.7;

    /* Output the value of sum using the printf function.
       The format specifier %f tells printf to output the
       value of a float variable. */
    printf("The sum of 10.5 and 20.7 is %f\n", sum);

    return 0;
}
```

Datatyper för tecken

- Datatypen **char** kan användas för att lagra bokstäver / tecken
- Varje tecken motsvaras egentligen av siffervärden i *ASCII-tabellen* (<http://www.asciitable.com/>)
=> En **char** kan ha värden mellan 0 – 255 (8 bitar, en byte).
- Finns ett antal användbara *specialtecken* som radbyte, tabulator, etc.
 - Se Appendix A, tabell A3 i kursboken för ytterligare specialtecken
- Oftast krävs hantering av många tecken samtidigt
=> Behov av *strängar*
 - Behandlas senare under kursen

Teckenvariabler, exempel

```
#include <stdio.h>

int main(void)
{
    /* Declare some variables of type char, name them and
       initialize them */
    char aVariable = 'a'; // Note the single
                          // quotation marks

    char newLine = '\n'; // Special character,
                        // '\n' counts as one char

    /* Output the values using the printf function.
       The format specifier %c tells printf to output the
       values as characters. */
    printf("This is a character: %c\n", aVariable);
    printf("This is another character: %c\n", newLine);

    return 0;
}
```

Mera om printf

- **printf** har ett stort antal **flaggor** och **modifierare** som kan användas för att få exakt den utskrift som önskas för olika variabler
 - Antal decimaler, “padding”, etc
 - Se tabellerna 16.1 – 16.4 i kursboken för en komplett förteckning
- **printf** kan skriva ut värdet på **flera** variabler med ett och samma anrop
 - Viktigt att antal och ordningsföljd för 'format specifiers' matchar variabelparametrarna

Mera om printf, exempel

```
#include <stdio.h>

int main(void)
{
    int var1 = 23;
    double var2 = 45.237;
    char var3 = 'z';

    // Let's do some fancy formatting
    printf("var1 with 5 'padding spaces': %5d\n", var1);
    printf("var2 with 2 decimals: %.2f\n", var2);

    // Let's output 3 values with one printf
    printf("var1 is %d, var2 is %f, var3 is %c\n",
           var1, var2, var3);

    return 0;
}
```

“Type specifiers”

- “Tilläggsdirektiv” som kan användas för att modifiera variabelers storlek / lagringskapacitet
- **unsigned** tillåter endast positiva värden
- **long** används för att utöka kapaciteten för variabler
- **short** används för att minska kapaciteten (och därmed spara minnesutrymme)
- Om du endast anger datatypen som **unsigned**, **long**, **short**, osv. antar kompilatorn att du vill deklarera en **int** med dessa egenskaper.

Type specifiers, exempel

```
#include <stdio.h>

int main(void)
{
    /* Declare a variable of type short int */
    short int small_number = 32;

    /* Change the value of the variable to something big */
    small_number = 1500000000;

    /* Output the value */
    printf("Value is %d\n", small_number);

    return 0;
}
```

sizeof, exempel

```
// Can use sizeof to check the size in bytes
// of data types or variables
// ANSI C99 defines a special output format for sizeof: %zu
// However, this does not work with all compilers

printf("Size of char is %d\n", sizeof(char));
char char_var;
printf("Size of char_var is %d\n", sizeof(char_var));

printf("Size of short is %d\n", sizeof(short));
printf("Size of int is %d\n", sizeof(int));
printf("Size of unsigned int is %d\n", sizeof(unsigned int));
printf("Size of long is %d\n", sizeof(long));
printf("Size of float is %d\n", sizeof(float));
printf("Size of double is %d\n", sizeof(double));
printf("Size of long double is %d\n", sizeof(long double));
```

Aritmetiska uttryck

- Vi har sett exempel på några enkla aritmetiska uttryck: addition(+) och subtraktion(-)
- Övriga aritmetiska operatorer:
 - * (multiplikation)
 - / (division)
 - % (“modulus”, “rest vid heltalsdivision”)
- Viktiga begrepp vid beräkning av aritmetiska uttryck: *operatorprecedens* och *reglerna för heltalsaritmetik*

Enkla aritmetiska uttryck, exempel

```
int a=22, b=5; /* Note that several variables can be declared
                simultaneously! */
int c=a*b;

printf("c equals %d\n", c);
printf("a * b equals %d\n", a*b); /* Note that we don't
                                   actually need to use an
                                   intermediate variable */

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

/* What do you think the output of the last two statements
will be? */
```

Operatorprecedens

- Ett uttryck som innehåller många operatörer beräknas i enlighet med de olika operatorernas *precedens* eller prioritet
- Samma regler gäller som då ett uttryck beräknas för hand
- Multiplikation och division har t.ex. högre precedens än addition och multiplikation
 - $8*2+5*4 = 16+20 = 36$
- Med hjälp av parenteser kan precedensreglerna ändras
 - $8*(2+5)*4 = 8*7*4 = 224$

Operatorprecedens

- Inte fel att använda parenteser även då detta i princip inte är nödvändigt
 - $(8*2)+(5*4)$
 - Mer lättläst kod
 - Mindre risk för buggar
- Se tabell A.5 i kursboken för en komplett lista över operatorer och deras respektive prioritet.

Heltals- och flyttalsaritmetik, exempel

```
int a=25;
int b=2;

/*
    Dividing and then multiplying with the same factor
    should result in the original value, right?;
    Let's try it out!
*/

printf("25 / 2 * 2 = %d\n", a/b*b);

/* What is the actual result? Why? */
```


Heltals- och flyttalsaritmetik, exempel

- Ett uttryck med två heltalsoperander resulterar i ett heltalsresultat
- En eventuell heltalsdel “slängs bort”
- Måste använda flyttal för att behålla decimaldelen

```
int a1 = 25, b1 = 2;  
double a2 = 25.0, b2 = 2.0;  
/* What are the results of these calculations? */  
printf("25 / 2 * 2 = %f\n", a1/b1*b1);  
printf("25 / 2 * 2 = %f\n", a2/b2*b2);  
printf("25 / 2 * 2 = %f\n", a1/b2*b2);  
printf("25 / 2 * 2 = %d\n", a1/b2*b2);
```

“Type cast”-operatorn

- Ett annat sätt att lösa problemet med försvinnande decimaldelar: Omvandla en datatyp till en annan med *type cast*-operatorn ()

```
int a1 = 25, b1 = 2;
```

```
printf("25 / 2 * 2 = %f\n", (double)a1/b1*b1);
```

- I exemplet ovan omvandlas heltalet i **a1** till en **double** (med värdet 25.000...)
=> resultatet av hela beräkningen blir en **double**
(Vad säger dig detta om type cast-operatorns *precedens/prioritet*?)

Tilldelningsoperatoren

- Som vi redan har sett kan en variabel tilldelas ett värde med hjälp av *tilldelningsoperatoren* =
int a = 5;
- En variabls värde kan ändras i ett senare skede av programmet:
a = 237;
- En variabls värde kan bero av *en annan variabls värde*:
int b = 12;
a = 182 * b;
- En variabls nya värde kan bero av *samma variabls tidigare värde*:
a = a + b / 2;

Tilldelningsoperatoren

- Alternativ syntax för att öka eller minska värdet av en heltalsvariabel med ett:

```
a++; // motsvarar a = a + 1;
```

```
a--; // motsvarar a = a - 1;
```

- Alternativ syntax för att ändra en variabels värde utgående från dess tidigare värde:

```
a -= 10; // motsvarar a = a - 10;
```

```
a *= b; // motsvarar a = a * b;
```

```
a += b / 2; // motsvarar a = a + b / 2;
```

Tilldelningsoperatoren

- Vid deklaration och initialisering av flera variabler samtidigt måste alla variabler initialiseras med ett eget värde

`int a, b; // values of a and b are undefined`

`int a=0, b; // value of b is undefined`

`int a, b=0; // value of a is undefined`

`int a=0, b=0; // both a and b are initialized`

- Rekommendation: **Intialisera alltid alla variabler!**

Inmatning av data - **scanf**

- **scanf** kan ses som en direkt motsvarighet till **printf**: **printf** skriver ut information, **scanf** läser in information (via tangentbordet)
- Den inmatade information lagras i en variabel
- Viktigt att ange rätt datatyp vid inmatning av data
 - I stort sett samma syntax som för **printf**: t.ex. motsvarar **%d** ett heltal.
 - **Undantag**: Vid inläsning av en **double** måste syntaxen **%lf** användas för att mata in ett “långt flyttal”; endast **%f** läser in en **float**

scanf, exempel

```
#include <stdio.h>

int main(void)
{
    int num1, num2;
    double num3;

    printf("Please input an integer! ");
    // Note: & before the variable name!
    scanf("%d", &num1);
    printf("You entered %d\n", num1);

    printf("Please input an integer and a double! ");
    scanf("%d%lf", &num2, &num3);
    // Note: You can also use %lf in printf
    printf("You entered %d and %lf\n", num2, num3);

    return 0;
}
```

Inmatning av data - **scanf**

- **&**-tecknet framför variabelnamnen i **scanf** är inte ett tryckfel
 - Denna syntax används för att komma åt *minnesadressen* till en variabel – mera om detta i samband med avsnittet om minneshantering
- Vid inmatning av flera värden kan dessa separeras med mellanslag, tabulator eller radbyte
- Tills vidare gör vi det överoptimistiska antagandet att användaren alltid matar in data av rätt typ

scanf och char-inläsning

- **scanf** läser in data från en inputbuffer
- Det radbytestecken, mellanslag, etc. som avslutat inmatningen lämnas kvar i buffern efter inläsningen
- Problem vid upprepad inmatning:
scanf("%c", &charvar) kommer att läsa in detta 'avslutningstecken' till **charvar** och användaren får aldrig en chans att mata in någon ny data
- Kan instruera **scanf** att ignorera inledande whitespace genom att lägga in ett extra mellanslag i 'inputsträngen':
scanf(" %c", &charvar);

scanf och char-inläsning

```
char c1, c2;

// First try - won't work
printf("Please input first char!\n");
scanf("%c", &c1); // Terminating character is left in buffer
printf("Please input second char!\n");
scanf("%c", &c2); // Will read the terminating char from buffer

printf("You entered %c and %c\n", c1, c2);

// Second try - skip whitespace before actual data
printf("Please input first char!\n");
scanf("%c", &c1); // Terminating character is left in buffer
printf("Please input second char!\n");
scanf(" %c", &c2); // Ignore terminating character

printf("You entered %c and %c\n", c1, c2);
```

Beslutsfattande

- Hittills har våra program alltid körts en rad i taget, från första raden till sista
- För att skriva realistiska program behövs mekanismer för att få datorn att fatta **beslut** utgående från olika **villkor**
- Gör det möjligt att köra endast delar av program (=flödeskontroll), eller köra valda delar av programmet upprepade gånger (=iteration)
- För att åstadkomma detta krävs *logiska uttryck* och *villkor* – **boolesk logik**

Logiska uttryck

- Logiska uttryck har antingen värdet *sant* eller *falskt*
 - “I dag skiner solen” \Rightarrow sant eller falskt, beroende på vädret
 - “ $1+2$ är större än 5” \Rightarrow alltid falskt
 - “ $a-3$ är mindre än 0” \Rightarrow sant om a är mindre än 3
- Datorprogrammering går till stor del ut på att analysera logiska uttryck och beroende på resultatet utföra olika instruktioner
- “Om uttrycket är sant, gör si. Om inte, gör så.”

Relationsoperatorer

- Kan använda *relationsoperatorer* liknande dem som används i vanlig algebra för att konstruera logiska uttryck:

$==$ ekvivalent med, “lika med”
(Obs! Två likhetstecken!)

$!=$ icke ekvivalent med, “olika”

$<$ mindre än

$>$ större än

$<=$ mindre än eller lika med

$>=$ större än eller lika med

Konstanta logiska uttryck

10 < 23 (sant)

10 == 23 (falskt)

10 != 23 (sant)

10 >= 10 (sant)

- Värdet på ovanstående uttryck är **konstanta**
- I ett program innehåller de logiska uttrycken vanligen en variabel
=> uttryckets värde kan **variera** under programkörningens gång

Övning: Variabla logiska uttryck

- För vilka värden på **a** och **b** är följande uttryck sanna respektive falska?:

$$a < 21$$

$$a * 4 == 20$$

$$121 != a$$

$$a > b$$

$$2 * b <= a$$

Beslutsfattande och programflöde:

- Med hjälp av logiska uttryck och **if**-satser kan valda delar av ett program köras:

```
#include <stdio.h>
int main(void)
{
    int age = 0;
    printf("Välkommen till Trafi\n");
    printf("Hur gammal är du? ");
    scanf("%d", &age);
    if (age < 18) ← Inget semikolon här!
    {
        printf("Inget körkort för dig!\n");
    }

    return 0;
}
```

Kodblock som
körs endast om
villkoret är **sant**

Beslutsfattande och programflöde:

- **if**-satsen kan utökas med en **else**-del för ett alternativt programflöde:

```
if (age < 18)
{
    printf("Inget körkort för dig!\n");
}
else // means that age >= 18
{
    printf("Ok, tuta och kör!\n");
}
```

Kodblock som
körs endast om
villkoret är **falskt**



Beslutsfattande och programflöde:

- Det kan finnas flera **else**-delar:

```
if (age < 18)
{
    printf("Inget körkort för dig!\n");
}
else if (age > 69)
{
    printf("Läkarkontroll krävs!\n");
}
else // means that age is between 18 and 69
{
    printf("Ok, tuta och kör!\n");
}
```

Nästlade if-satser

- Kan ha **if**-satser inne i andra **if**-satser:

Nästlad
if-sats

```
if (age > 69)
{
    int last_checkup;
    printf("Hur många år sedan senaste kontroll?");
    scanf(" %d", &last_checkup);
    if (last_checkup > 4)
    {
        printf("Läkarkontroll krävs!\n");
    }
    else
    {
        printf("Ok, återkom om %d år.\n",
            5-last_checkup);
    }
}
```

Iteration: While-loopen

- En **while**-loop består av ett kodblock som upprepas *så länge ett logiskt villkor är sant*

Kodblock som
skall upprepas
så länge $a < 10$

```
#include <stdio.h>

int main(void)
{
    int a = 0;
    while (a < 10) ← Inget semikolon!
    {
        printf("Loop!\n");
        a++; /* Öka värdet på a */
    }
    return 0;
}
```

Kodblock och variabler

- En variabel kan användas **inne i det kodblock där den är definierad samt i alla underliggande nästlade kodblock**
- **Inte tillåtet** att definiera en ny variabel med samma namn inom samma kodblock
- **Tillåtet** att definiera en ny variabel med samma namn i ett inre, nästlat kodblock
 - => denna variabel gäller endast i det inre kodblocket
 - => en “yttre” variabel med samma namn “skrivs över”
 - => flera variabler med samma namn gör programmet svårförståeligt och ökar risken för buggar

Kodblock och variabler, exempel

- Vad ger följande program för utskrift?

```
#include <stdio.h>

int main(void)
{
    int num = 10;
    while ( num >= 10 )
    {
        int num = 20;
        printf("num is now %d\n", num);
        num--;
    }
}
```

Kodblock utan 'curly braces'

- Om ett kodblock innehåller endast en instruktion kan vågparenteserna utelämnas
- Förekommer i kursboken, men rekommenderas inte!

```
int a=200;

while (a > 100)
    printf("%d\n", a);
    a = a - 10;

if (a<100)
    printf("Case one!\n");
else
    printf("Case two!\n");

// Utskrift?
```

Nästlade loopar

- En loop kan innehålla en annan loop
- Vad kommer nedanstående program att skriva ut?

```
int a=1;
while ( a <= 10 )
{
    int b=0;
    while ( b <= 5 )
    {
        printf("a:s värde är nu %d\n", a);
        printf("b:s värde är nu %d\n", b);
        b++;
    }
    a++;
}
```


Datatyper för booleska värden

- Kan lagra resultatet från ett logiskt uttryck i en boolesk variabel
- Traditionellt använder C heltalsvärden för att representera sant eller falskt: 0 för *false*, 1 för *true*
- Fr.o.m. C99 stöder C även en boolesk datatyp **_Bool**
 - Giltiga värden är 0 eller 1
- Kan inkludera headern **stdbool.h**

=> Kan använda datatypen **bool** och värdena **true** och **false** i dina program

=> Kompilatorn omvandlar **true** till 1 och **false** till 0

Booleska variabler, exempel

```
#include <stdio.h>
#include <stdbool.h>
int main(void)
{
    bool go_on = true;
    printf("Welcome to the merry-go-round\n\n");
    while (go_on == true)
    {
        printf("I'm getting dizzy!\n");
        printf("Once more (y/n)? ");
        char input = 0;
        scanf(" %c", &input);
        if (input == 'n')
        {
            go_on = false;
        }
    }
}
```

Booleska variabler, exempel

[illegible]

Booleska variabel, exempel

```
#include <stdio.h>
// Note: No stdbool required in this case!
int main(void)
{
    int go_on = 1; // Use integer value directly
    printf("Welcome to the merry-go-round\n\n");
    while (go_on)
    {
        printf("I'm getting dizzy!\n");
        printf("Once more (y/n)? ");
        char input = 0;
        scanf(" %c", &input);
        if (input == 'n')
        {
            go_on = 0;
        }
    }
}
```

Booleska variabler, exempel

```
#include <stdio.h>
// Note: No boolean variable needed at all!
int main(void)
{
    char input = 'y';
    printf("Welcome to the merry-go-round\n\n");
    while (input != 'n')
    {
        printf("I'm getting dizzy!\n");
        printf("Once more (y/n)? ");
        scanf(" %c", &input);
    }
}
```

Introduktion till programdesign

- Ett program skall utföra en viss uppgift...
- ...men det finns sällan bara ett sätt att lösa en uppgift
- (Nästan) alla program kräver planering innan kodningsfasen
- Olika typer av designmetoder eller paradigmer: procedurell (designen implementeras t.ex. med C), objektorienterad (designen implementeras t.ex. med C++/Java)...
- ...men de grundläggande byggstenarna är de samma oberoende av metod: datatyper, variabler, kontrollstrukturer, datastrukturer, funktioner/metoder, input/output...

Pseudokod och flödesscheman

- Rekommendation: “Förhandsplanera” även de enkla program som konstrueras under denna kurs
- Kan t.ex. använda pseudokod och/eller flödesscheman för att designa programmets algoritm
- Pseudokod: Förenklad programbeskrivning med naturligt språk istället för t.ex. C-syntax
- Flödesschema: Visualisering av programmets funktionalitet

Exempel: Pseudokod

- Pseudokod för ett program som läser in tio tal och beräknar summan av dessa, och en möjlig implementation i C:

Nollställ totalsumman
Upprepa följande tio gånger:

- Läs in ett tal
- Uppdatera totalsumman

Skriv ut totalsumman

```
int sum = 0, counter = 0;
while (counter < 10)
{
    int input = 0;
    printf("Input value: ");
    scanf("%i", &input);
    sum+=input;
    counter++;
}
printf("Sum is %i\n", sum);
```


Exempel: Pseudokod

- Finns inget standardiserat sätt att skriva pseudokod

Keep track of current number of resources in use

IF another resource is available

 Allocate a dialog box structure

 IF a dialog box structure could be allocated

 Note that one more resource is in use

 Initialize the resource

 Store the resource number at the location
 provided by the caller

 ENDIF

ENDIF

Return true IF a new resource was created; ELSE return false

Flödesschema

- Typiska komponenter:

- Aktiviteter:




Uppdatera
summan

- Input/Output:



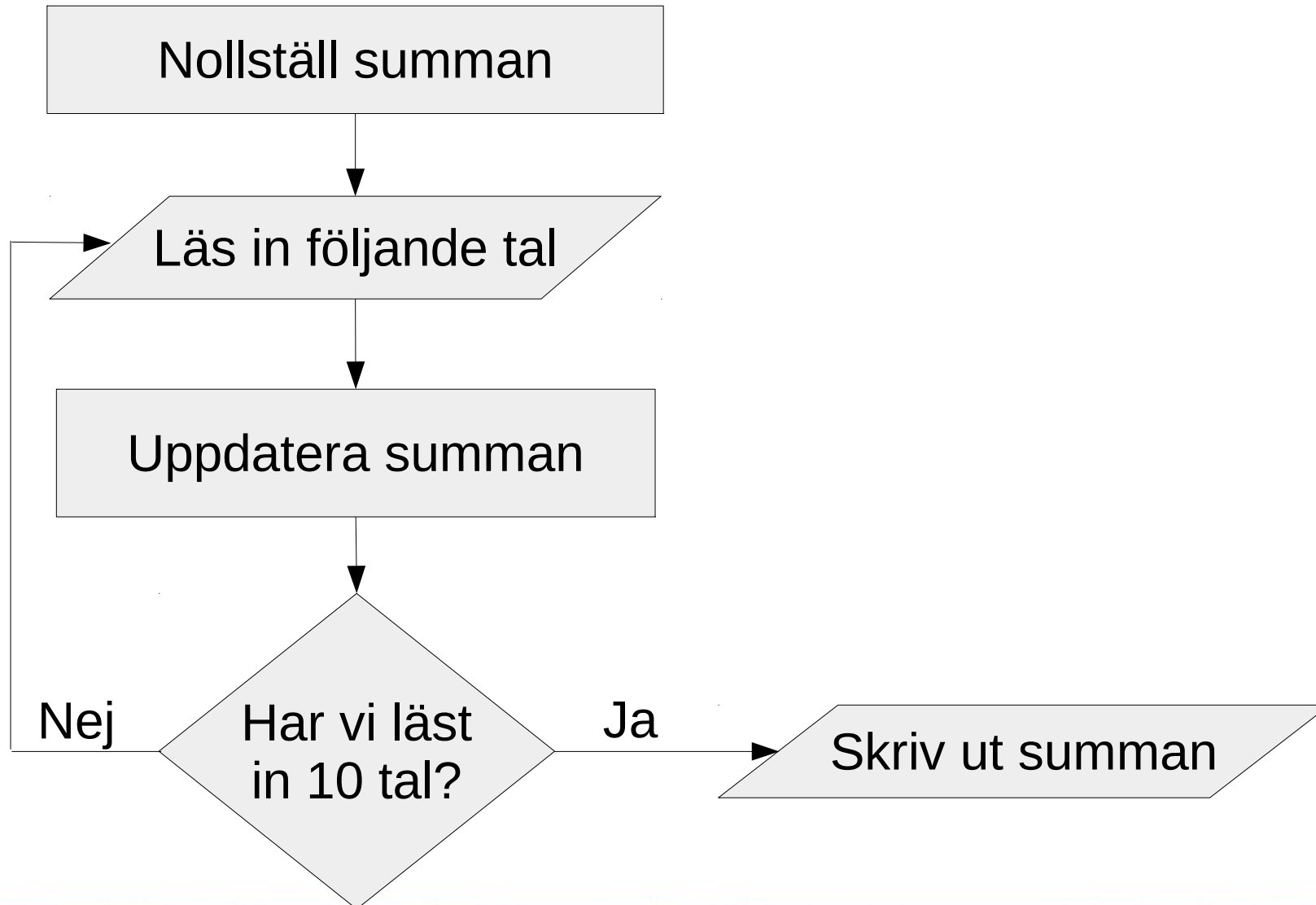
Läs in följande tal

- “Decision points”:



Har vi läst
in 10 tal?

Exempel: Flödesschema



Övning: Pseudokod och flödesschema

- Bilda grupper om 3-4 personer. Varje grupp funderar på följande problembeskrivningar:
- Problem 1: Reparera en cykel som har punktering
- Problem 2: 13 spelkort ligger utlagda i stigande ordningsföljd på bordet, från äss till kung. Byt ordningsföljd på korten.
- Problem 3: Hitta Joakims telefonnummer i en telefonkatalog
- Designa *algoritmer* för att lösa problemen. Använd er av pseudokod eller flödesscheman.

Komplext beslutsfattande

```
// Print OK if age is between 18 and 69
if (age >= 18)
{
    if (age <= 69) // Nested loop :(
    {
        printf("OK!\n");
    }
}

// Print NOT OK if age is less than 18 or more than 69
if (age < 18)
{
    printf("NOT OK!\n");
}
if (age > 69) // Duplicated code :(
{
    printf("NOT OK!\n");
}
```

Logiska operatorer

- Ett logiskt uttryck kan bestå av flera *deluttryck* som kopplas samman med hjälp av *logiska operatorer*:
- **del1 AND del2** är sant om båda deluttrycken är sanna
 - Med C-syntax: **del1 && del2**
- **del1 OR del2** är sant om del1 **eller** del2 (eller bägge villkoren) är sanna
 - Med C-syntax: **del1 || del2**
- **NOT del1** är sant om del1 **inte** är sant
 - Med C-syntax: **!del1**

Sanningstabeller

a	b	a AND b	a OR b
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE

Logiska operatorer, exempel

```
if (age >= 18 && age <= 69) // AND
{
    printf("OK!\n");
}
```

```
if (age < 18 || age > 69) // OR
{
    printf("NOT OK!\n");
}
```

```
if (!(age < 18 || age > 69)) // NOT, OR
{
    printf("OK..I think?");
}
```


Övning: Logiska operatorer

- Vad innebär följande uttryck?
(antag att *a* och *b* är heltalsvariabler)
`(a > 50) && (a < 100)`
`(b % 2 == 0) || (b == 1)`
- Formulera villkor för följande situationer med hjälp av relationsoperatorer och logiska operatorer:
 - Variabeln **age** skall vara lika med 30 eller 40
 - Variabeln **value1** skall vara större än en fjärdedel men mindre än **value2**
 - Variabeln **a** skall vara mindre än **b** som i sin tur skall vara större än eller lika med **c**.

Mera valmöjligheter: **switch**

- Alternativt sätt att styra programflödet
- Avsett för situationer där alternativet är en lång **if..else** konstruktion
- **expression** kan anta olika värden
- om **value1** => utför case 1
- om **value2** => utför case 2
- annars, utför **default**

```
switch (expression)
{
    case value1:
        program statements
        break;
    case value2:
        program statements
        break;
    ...
    default:
        program statements
        break;
}
```

switch, exempel

```
int i = 0;
printf("Enter month (1-12)");
scanf("%d", &i);
switch (i)
{
    case 1:
        printf("January\n");
        break;
    case 2:
        printf("February\n");
        break;
    // and so on...
    case 12:
        printf("December\n");
        break;
    default:
        printf("Illegal month!");
        break;
}
```

```
int noOfDays = 0;
switch (i) // assume i is an integer value
{
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        noOfDays = 31;
        break;
    case 2:
        noOfDays = 28; // Ignoring leap year
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        noOfDays = 30;
        break;
    default:
        printf("Illegal month!\n");
}
```

**Inget break after case
=> 'fallthrough'**

**Inget break after case
=> 'fallthrough'**

Iteration: Do-loopen

- En variant av **while**-loopen – kodblocket körs alltid minst en gång, även om det logiska villkoret är falskt

Kodblock som körs en gång trots att villkoret inte uppfylls

```
#include <stdio.h>

int main(void)
{
    int a = 0;
    do
    {
        printf("Loop!\n");
        a++; /* Öka värdet på a */
    }
    while (a < 10); ← Semikolon!
    return 0;
}
```

Iteration: Do-loopen

```
#include <stdio.h>

int main(void)
{
    char input; // initial value irrelevant!
    printf("Welcome to the merry-go-round\n\n");
    do // at least one round!
    {
        printf("I'm getting dizzy!\n");
        printf("Once more (y/n)? ");
        scanf(" %c", &input);
    }
    while (input != 'n');
}
```

Iteration: **for**-loopen

- Används ofta då man på förhand vet hur många iterationer som skall utföras
- Mer kompakt syntax än **while**-loopen, men en **for**-loop kan alltid uttryckas som en **while**-loop
- Allmän syntax:
for (*init_expression; loop_condition; loop_expression*)
{
 program statements
}

Iteration: **for**-loopen

- Exempel: Skriv ut "Hej!" tio gånger:

Initialisering:
Utförs **en gång**

Loopen upprepas
så länge detta
villkor är sant

Utförs **efter varje iteration**

Kodblock som
utförs medan
villkoret är sant

```
int a;  
for ( a = 0; a < 10; a++ )  
{  
    printf("Hej!\n");  
}
```


Variabeldeklaration & initialisering

- ANSI C99 tillåter deklaration av en variabel inne i for-loopens initialiseringsdel
=> Variabeln giltig endast inne i for-loopen
- **Äldre versioner av gcc (pre-5.x)** kräver flaggan **std=c99** för att nedanstående kod skall kunna kompileras

```
for ( int a = 0; a < 10; a++ )  
{  
    printf("Hej!\n");  
}  
  
// use 'gcc test.c -std=c99' if needed
```

Variationer på **for**-loopar

- Ett fält i en for-loop kan innehålla flera instruktioner som separeras med komma
- Ett fält kan lämnas helt tomt

```
/* Example: One statement in the 3rd field */
```

```
for ( int a = 0, b = 200; a < 10; a++, b = b - 4 )  
...
```

```
/* Example: First field is empty */
```

```
int c = 50;  
for ( ; c > 100; c = c + 10 )  
...
```

Nästlade for-loopar

- For-loopens kompakta syntax gör den lämplig för nästlade loopar

```
for ( int a=1; a <= 10; a++ )
{
    for ( int b=5; b >= 1; b-- )
    {
        printf("a:s värde är nu %d\n", a);
        printf("b:s värde är nu %d\n", b);
    }
}
```

Räckor, arrays

- Antag att du skall skriva ett program som läser in n värden och sedan använder sig av dessa
- Problemet kan lösas genom att deklarera n stycken variabler
 - Opraktiskt och oflexibelt
- En *räcka* (eng. array) kan ses som en *ordnad mängd* (ordered set) innehållande element av en viss typ
- I stället för att deklarera n stycken heltalsvariabler kan vi deklarera **en** räcka som **innehåller** n heltalselement

Deklaration av räckor

- En räckor kan endast innehålla värden av en **viss typ** (heltal, flyttal, bokstäver...)
- När en räckor deklarerar måste man ange hur många element som finns i räckan
- För att deklarerar en räckor som innehåller 20 heltal:
int myIntArray[20];
- För att deklarerar en räckor som innehåller 40 bokstäver:
char myCharArray[40];

Elementen i räckan

- För att komma åt ett element i som finns i räckan x används notationen $x[i]$
- i kallas för ett **index**.
- Det första elementet i en räkka har alltid indexet **0**.
- För att komma åt det **första** elementet i en räkka x med n element anges därmed $x[0]$
- För att komma åt det **sista** elementet anges $x[n-1]$
- För att t.ex. komma åt det **tredje** elementet i vår heltalsräcka och tilldela detta element värdet 100:
myIntArray[2] = 100;

Räckor i datorns minne

Hur lagras en räkka deklarerad som
int values[10] i datorns minne?

values[0]	?
values[1]	?
values[2]	?
values[3]	?
values[4]	?
values[5]	?
values[6]	?
values[7]	?
values[8]	?
values[9]	?

Efter deklarationen

values[0]	197
values[1]	?
values[2]	-101
values[3]	547
values[4]	?
values[5]	350
values[6]	?
values[7]	?
values[8]	?
values[9]	35

Efter några tilldelningar

Räcker, exempel

```
int myArray[10]; // Declare an array with 10 integers
int index;

/* Loop through the array,
   initialize each element to index*2 */

for (index = 0; index < 10; index++)
{
    // Note that the value of index ranges from 0 to 9!
    myArray[index] = index*2;
}

printf("The 5th element in the array is %d\n",
       myArray[4]);
```


Direkt initialisering av räckor

- Också möjligt att initialisera räckan samtidigt som den deklarerar
- Praktiskt användbart endast för små räckor
- Inte nödvändigt att ange hur många element räckan innehåller – **om** du initialiserar varje element i räckan
- Exemplet nedan deklarerar och initialiserar en räkka med 5 element:

```
double aShortArray[] = { 1.0, 3.14, 2.2, 4.3, -3.23 };
```



Ingen storlek
behöver anges

'Curly braces' för att
avgränsa mängden

Direkt initialisering av räckor

- Kan användas för automatisk initialisering av räckans element:

```
int myArray[10] = { 1, 2, 3 };
```

*myArray[0] = 1, myArray[1] = 2, myArray[2] = 3
myArray[3..9] får värdet 0*

```
int myArray[10] = { 0 };
```

Alla element får värdet 0

```
int myArray[10] = {};
```

*Alla element får värdet 0 **om** kompilatorn
stöder detta. **Inte** standard-c!*

*(pröva att kompilera med **gcc -pedantic**)*

Räckor, exempel

- Ett program för att beräkna statistik för enkätsvar
- Läs in x antal värden mellan 1 och 10 (som svarar mot 10 olika “kategorier”)
- Lagra antal svar per kategori i en räkka
- Skriv ut resultatet genom att gå igenom räckans element
- Källkoden finns i katalogen *Exempelprogram* i Moodle
- Observera hur programmet beaktar att en räckas första index är noll: För att göra programmet mera lättförståeligt skapas en räkka med 11 element, trots att vi egentligen endast behöver 10 element.
=> Kan använda elementen 2-11 (med *index* 1-10)

“Variable-length arrays”

- Räckor vars längd inte har slagits fast i kompileringsskedet – sådana räckors längd specificeras genom *värdet av en variabel*
- Fortfarande **inte** möjligt att ändra storleken på räckan **efter** att den har skapats

```
int size = 0;

printf("How many elements do you need?");
scanf("%d", &size);

// Declare an array with 'size' number of elements
double mySizeOnDemandArray[size];
```

Lookup tables

```
/* Declare an array containing the
   number of days per month */
int daysInMonth[12] = {31, 28, 31, 30, 31, 30, 31,
                       31, 30, 31, 30, 31};

int month = 0;
do {
    printf("Enter month (1-12):");
    scanf("%d", &month);
}
while ((month < 1) || (month > 12));

/* Perform a lookup to get the amount
   of days for this month. Note usage of -1 to
   account for array index starting from 0 */
printf("This month has %d days\n", daysInMonth[month-1]);
```

Flerdimensionella räckor

- Ett element i en räkka kan innehålla en annan räkka
=> *flerdimensionell* räkka
- En tvådimensionell räkka med i rader och j kolumner kan även ses som en *matrix*
- För att deklarera en tvådimensionell räkka med 5 rader och 10 kolumner, och sedan tilldela värdet 42 till elementet i rad 3, kolumn 7:

```
int myTwoDimensionalArray[5][10];  
myTwoDimensionalArray[2][6] = 42;
```

Flerdimensionella räckor, exempel

```
int mArray[3][5] = { {1, 2, 3, 4, 5 },  
                     {0, 0, 0, 0, 0 },  
                     {0, 0, 0, 0, 0 } };
```

```
int i;  
for (i=1; i<3; i++)  
{  
    for (int j=0; j<5; j++)  
    {  
        mArray[i][j] = mArray[0][j] * (i+1);  
    }  
}
```

// Contents of mArray after program execution is?

Visualisering av flerdimensionella räckor

1	2	3	4	5
0	0	0	0	0
0	0	0	0	0

Efter initialisering ($i=0$)

1	2	3	4	5
2	4	6	8	10
0	0	0	0	0

Efter första yttre iterationen ($i=1$)

1	2	3	4	5
2	4	6	8	10
3	6	9	12	15

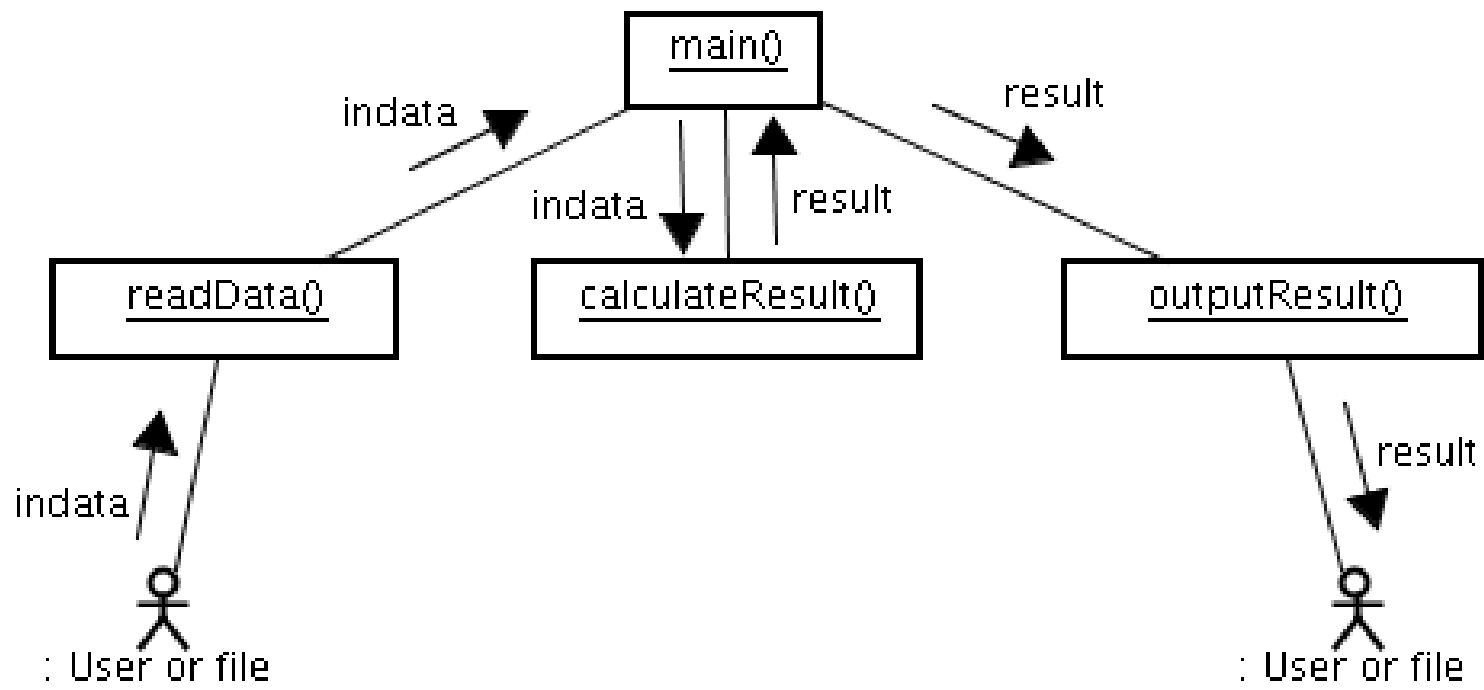
Efter andra yttre iterationen ($i=2$)

Funktioner

- Hittills har alla våra program innehållit endast en “egen” funktion: **main()**
- Vi har däremot använt andra, “inbyggda” funktioner från c:s standardbibliotek: **printf()**, **scanf()**, **pow()**
- Normalt innehåller **ett** program **många** funktioner
- Vi delar upp programmet i lämpliga “helheter” som kan göras till separata funktioner
- Grundregel: En uppgift – en funktion

Funktioner, exempel

- Typisk struktur för ett enkelt program som läser in data (från en fil, från användaren), utför beräkningar, och levererar ett resultat:



Deklaration av funktioner

- En funktion måste *deklareras*
 - Definiera vilken typ av *parametrar* (indata) som kan skickas till funktionen
 - Definiera vilken typ av data funktionen *returnerar* (utdata)

- **Exempel:**

Funktionen heter **getMax**


Funktionen accepterar
två heltalsparametrar

int getMax(int firstValue, int secondValue)

Funktionen **returnerar** ett **heltal**

Implementation av funktioner

- Efter funktionsdeklarationen följer funktionens implementation
- Implementationen utgör ett kodblock (som i vanlig ordning kan innehålla andra kodblock)



```
int getMax( int firstValue, int secondValue)
{
    if (firstValue > secondValue)
    {
        return firstValue;
    }
    else
    {
        return secondValue;
    }
}
```

Funktionsanrop

- Funktionen anropas genom dess namn
- Den indata som skickas till en funktion kallas **argument**
- Vid funktionsanropet måste funktionens parameterdeklaration följas
=> måste skicka **rätt antal argument** av **rätt typ**, i **rätt ordningsföljd**
- Ett returvärde kan t.ex. lagras i en variabel eller skrivas ut direkt
- En funktion kan returnera **endast ett värde**

Funktionsanrop

```
int main(void)
{
    // Send the arguments 10 and 20 to getMax
    // and store the return value in a variable
    int result = getMax(10, 20);
    printf("Result is %d\n", result);

    // Send the arguments 40 and result to getMax
    // Print the return value right away
    printf("Max of 40 and %d is: %d\n",
           result, getMax(40, result));

    return 0;
}
```

Funktion med returvärde

```
double calculateCubeVolume(double cubeSide)
{
    return cubeSide*cubeSide*cubeSide;
}

int main(void)
{
    double volume;

    volume = calculateCubeVolume(3);
    printf("Volume is %f\n", volume);

    volume = calculateCubeVolume(20);
    printf("Volume is %f\n", volume);
    return 0;
}
```

Funktion utan returvärde - **void**

```
void calculateCubeVolume(double cubeSide)
{
    double volume = cubeSide*cubeSide*cubeSide;
    printf("Volume is %f", volume);
    return; /* Using return for a function with void
            return type is optional */
}

int main(void)
{
    calculateCubeVolume(3);
    calculateCubeVolume(20);

    return 0;
}
```


Funktioner och variabler

- Vilken utskrift ger följande program?

```
void decreaseValue(int value1) {  
    int value2 = 300;  
    value1 = value1 - 20;  
    return;  
}  
  
int main(void) {  
    int value1 = 100;  
    int value2 = 200;  
    decreaseValue( 100 );  
    printf("value1 is now %d and value2 is %d\n",  
           value1, value2);  
    return 0;  
}
```

Funktioner och variabler

- En variabel deklarerad i en funktion kan **inte** användas i en annan funktion => *lokal variabel*
- Då ett argument sänds till en funktion skapas i själva verket en ny lokal variabel i den mottagande funktionen
- Den nya variabeln initialiseras med argumentets värde
- Förändringar i denna nya lokala variabel har ingen inverkan på innehållet i den ursprungliga variabeln
=> *Namnet* på variabeln(=parametern) i den mottagande funktionen har ingen betydelse för programmets funktionalitet – endast *datatypen* är viktig

Globala variabler

- En variabel kan deklareras *utanför* programmets funktioner
- Denna variabel kan då läsas och modifieras av alla funktioner i programmet
- Globala variabler kan användas för att möjliggöra enkel kommunikation mellan olika delar av programmet
- Å andra sidan kan ett (för) stort antal globala variabler också orsaka problem. **Varför?**

Globala variabler

```
int globalNumber = 0;

void fun1() {
    globalNumber++;
}

void fun2() {
    globalNumber--;
}

int main(void) {
    fun1();
    fun2();
    globalNumber++;
    printf("globalNumber is %d\n", globalNumber);
    return 0;
}
```

Funktioner och räckor

- Kan också skicka räckor som argument till en funktion
- **Viktig skillnad** jämfört med “vanliga” funktionsparametrar: En räkka kopieras **inte** i sin helhet när den skickas till en funktion
- I stället skickas endast information om var i datorns minne (början av) räckan befinner sig.
 - *Pekare* till en viss minnesposition
- Detta innebär att en modifikation av räckans innehåll också syns i **den anropande funktionen**

Funktioner och räckor

- För att kunna hantera räckor av olika storlek måste en funktion som arbetar med en räckan i regel ta emot två parametrar:
(pekaren till) räckan samt *antalet element i räckan*, i form av en **int**.
- Om innehållet i räckan modifieras inne i funktionen behöver funktionen oftast inte returnera någonting => returtyp **void**
 - För att explicit returnera en räckan krävs användning av pekare
- Exempel: **array_func.c** (i Exempellösningmappen i Moodle)

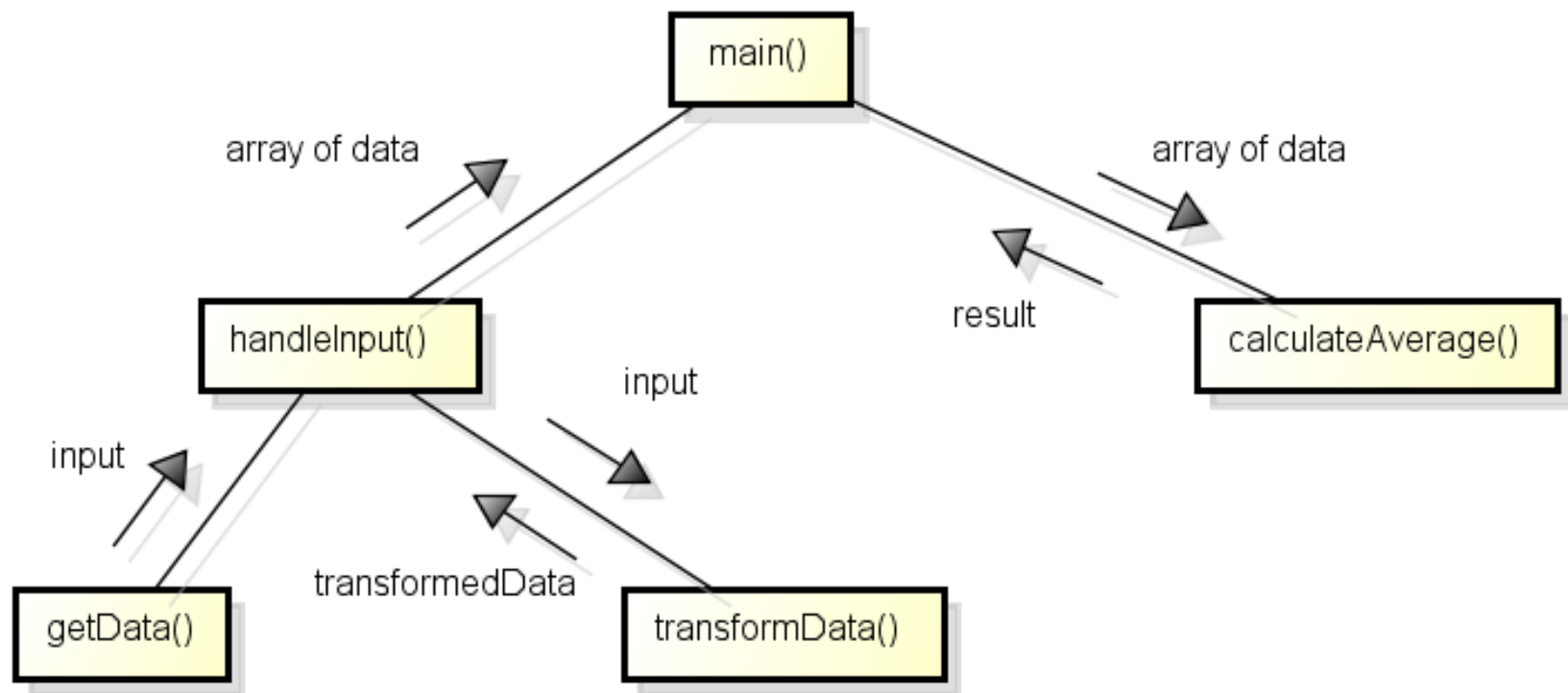
Multipla returvärden

- En funktion kan **ta emot** många parametrar, men endast **returnera** ett värde
- Om funktionen behöver returnera två eller flera värden **av samma typ** kan vi låta funktionen ta emot en räkka som parameter och modifiera innehållet i denna
- Om funktionen skall returnera värden **av olika typ** kan vi använda *strukturer* eller *pekare*

Funktioner som anropar funktioner som anropar...

- Kan finnas många nivåer i 'funktionsträdet'

– *Call stack*



- Exempel: **multilevel.c** (i Exempellösningmappen i Moodle)

Structs, “strukturer”

- Antag att du skall skriva ett program som hanterar **ett** datum (dag, månad, år)
- Hur skulle du representera detta datum i ditt program?
- Antag att ditt program skall hantera **100** datum. Hur skulle du representera dessa?
- Istället för att använda separata variabler för olika delar av datumet kan en **struct** användas för att skapa en *sammansatt datatyp*

En struktur för datum

- I stället för att definiera tre olika variabler kan vi definiera en ny *datatyp* som *innehåller* uppgifter om dag, månad och år
- Därefter kan vi deklarera *en* variabel som innehåller alla dessa uppgifter
- “Delarna” som ingår i denna nya datatyp benämns “members” (medlemmar)

```
/* Declare a new struct
   and name it "date" */
struct date
{
    int day; /* Member #1 */
    int month; /* Member #2 */
    int year; /* Member #3 */
}; /* Note the semicolon! */

/* Declare a variable of
   type "struct date" */
struct date today;
```

En struktur för datum

- För att komma åt de olika delarna av datumet används en speciell syntax:

*variabelns namn,
följt av en **punkt**,
följt av delens namn*

```
/* Declare a variable of  
   type "struct date" */  
struct date today;  
  
/* Initialize the variable */  
today.day = 24;  
today.month = 10;  
today.year = 2016;  
  
/* Output the values */  
printf("Today is %d/%d/%d\n",  
today.day,  
today.month,  
today.year);
```

Initialisering av structs

```
/* A struct can also be initialized in a  
   similar fashion as an array: */
```

```
struct date today = { 24, 10, 2016 };
```

```
/* ...Or like this: */
```

```
struct date today1 = { .day = 24,  
                       .month = 10,  
                       .year = 2016 };
```

```
/* The former syntax also enables you to  
   initialize only some members, and in any order: */
```

```
struct date today2 = { .month = 10, .day = 24 };
```

Användning av structvariabler

- Medlemsvariablerna i en struct kan läsas och uppdateras exakt som vanliga variabler:

```
/* Increase date by one, assuming we have the
   daysInMonth lookup table available.
   Note: the example below is not complete! */

if (daysInMonth[today.month] == today.day)
{
    today.month++;
    today.day = 1;
}
else
{
    today.day++;
}
```

Strukturer och funktioner

- **struct**-variabler kan förstås också skickas som argument till funktioner
- För att använda en struct i flera funktioner måste deklarationen av datastrukturen göras **globalt**, på samma sätt som för globala variabler
- Med hjälp av en **struct** kan man returnera flera enskilda värden från en funktion
 - (Ett annat alternativ för att åstadkomma detta är med hjälp av *pekare*, som behandlas senare i kursen)

Strukturer och funktioner

```
int calcDateDifference(struct date d1, struct date d2)
{
    int difference = 0;
    // Actual calculation of 'd1 - d2' omitted...
    return difference;
}
```

```
struct date firstDate, secondDate;
// Initialization of dates omitted...
```

```
int result = calcDateDifference(firstDate, secondDate);
printf("Difference between the dates is %d days\n",
                                             result);
```

Strukturer och funktioner

```
struct date calcDateDifference( struct date d1,  
                                struct date d2 )  
{  
    struct date difference = 0;  
    // Actual calculation of 'd1 - d2' omitted...  
    return difference;  
}
```

```
struct date firstDate, secondDate;  
// Initialization of dates omitted...
```

```
struct date result = calcDateDifference( firstDate,  
                                          secondDate );  
printf("Difference between the dates is \\  
      %d year(s), %d month(s) and %d day(s)\n",  
      result.year, result.month, result.day);
```


Strukturer och räckor

- Strukturer kan placeras i räckor –
en räkka kan endast innehålla en viss strukturtyp:

```
// Declare an array of 100 'struct date' elements
struct date dataArray[100];

// Initialize the 1st element – must use a type cast
dataArray[0] = (struct date){24, 10, 2016};

// Set the year member of the 2nd element
dataArray[1].year = 2017;

// Loop through the array, output day members
for (i=0; i<100; i++)
{
    printf("%d", someDates[i].day);
}
```

Strukturer som innehåller strukturer

- En struktur kan innehålla en annan struktur:

```
struct time {
    int hour;
    int minute;
    int second;
};

struct dateAndTime {
    struct date sDate;
    struct time sTime;
};

// Initialize both date and time 'substructs'
struct dateAndTime dt = { {24,10,2017}, {12,30,00} };
// Change the hour value
// Note the 'double-dot' syntax
dt.sTime.hour = 13;
```

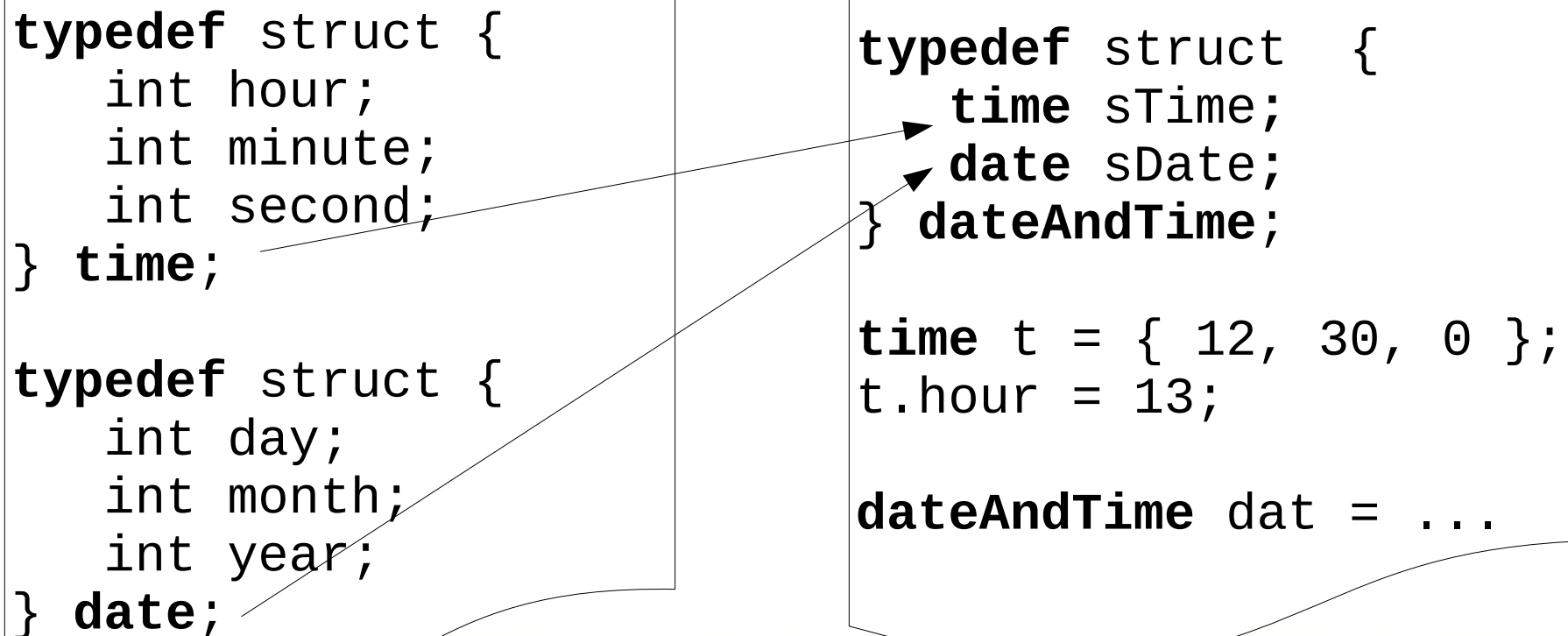
Typedef

- Kan använda **typedef** för att ge ett 'alias' åt en viss typ
- Användbart i samband med strukter – behöver inte ange **struct** vid deklaration av variabler

```
typedef struct {  
    int hour;  
    int minute;  
    int second;  
} time;
```

```
typedef struct {  
    int day;  
    int month;  
    int year;  
} date;
```

```
typedef struct {  
    time sTime;  
    date sDate;  
} dateAndTime;  
  
time t = { 12, 30, 0 };  
t.hour = 13;  
  
dateAndTime dat = ...
```

The diagram illustrates the use of typedef with two examples. On the left, two typedefs are shown: one for a 'time' struct (with fields hour, minute, second) and one for a 'date' struct (with fields day, month, year). On the right, a 'dateAndTime' struct is defined, which contains a 'time' struct and a 'date' struct. Below this, two variables are declared: 't' of type 'time' and 'dat' of type 'dateAndTime'. Arrows point from the 'time' typedef in the left box to the 'time' field in the 'dateAndTime' struct and to the 'time' variable declaration. Another arrow points from the 'date' typedef in the left box to the 'date' field in the 'dateAndTime' struct.

Headerfiler

- Vi har hittills skrivit alla våra funktioner i en enda källkodsfil (och sett till att deklarera funktionerna i rätt ordning!)
- För större program måste funktionerna delas upp på flera filer
- Hur kan vi tillåta en funktion i en fil att anropa en funktion i en annan fil?
- Hur kan vi använda samma **struct** i två olika källkodsfiler?
- Lösning: Skriv egna *headerfiler*

Headerfiler

- Vi har redan inkluderat *redan existerande* headerfiler för att använda funktioner från kodbibliotek (**printf**, **scanf**, **pow...**)
- Skapa en *egen* headerfil om du vill
 - deklarerera funktioner
 - deklarerera datatyper
 - definiera globala variabler och konstantersom ska användas i flera källkodsfiler
- För att inkludera en egen headerfil används syntaxen **#include "myownheaderfile.h"**

FunktionsdeklARATIONER

- Består av den första raden i funktionen, följt av ett semikolon
- Löfte till kompilatorn: “Jag lovar att en implementation som motsvarar denna funktionsdeklARATION existerar när programmets olika delar skall länkas ihop”
- Om löftet inte uppfylls (funktionsimplementationen saknas helt eller avviker från det utlovade) => *länkningsfel*

```
/tmp/cc0FZa0G.o: In function `main':test.c:  
(.text+0x12):  
undefined reference to `getMeaningOfLife'  
collect2: ld returned 1 exit status
```

=> Vid användning av yttre bibliotek behövs både själva biblioteket (implementationen) **och** en headerfil med deklARATIONER

Funktionsdeklarationer

```
// calc.h
```

```
double calcArea(double radius);  
double calcCirc(double radius);
```

```
// main.c
```

```
#include <stdio.h>  
#include "calc.h" // to use calcArea  
  
int main(void) {  
    printf("Radius %f gives area %f",  
           2.5,  
           calcArea(2.5));  
    return 0;  
}
```

```
// calc.c
```

```
#include <math.h> // To use M_PI  
  
double calcArea(double radius) {  
    return M_PI*radius*radius;  
}  
  
double calcCirc(double radius) {  
    return 2*M_PI*radius;  
}
```

För att kompilera:
gcc main.c calc.c

Implicita funktionsdeklARATIONER

- Vad händer om `#include "calc.h"` utelämnas från `main.c`?

=> *implicit* funktionsdeklARATION av `calcArea()`

- Funktionen antas ha returtypen `int`
- gcc gissar sig till parametertyperna baserat på funktionsanropet

=> programmet kompileras men ger fel resultat

=> använd alltid *explicita* funktionsdeklARATIONER!

- Kan kompilera med flaggan `-Wall` för att få varningar:
`gcc main.c calc.c -Wall`

Include guards

- Funktioner, datatyper etc. får endast definieras en gång
- Exempelscenario:
Filen **foo.h** inkluderar filerna **bar.h** och **calc.h**
Filen **bar.h** inkluderar också **calc.h**
=> **calc.h** inkluderas två gånger
=> alla deklARATIONER i **calc.h** körs två gånger
=> kompilatorfel!
- Lösning: Använd *include guards*
- Skapar en unik definition för varje fil när den inkluderas första gången
- Kontrollerar om definitionen redan är gjord innan den inkluderas på nytt

Include guards

```
// calc.h

#ifndef __CALC_H__
#define __CALC_H__
float calcArea(float radius);
float calcCirc(float radius);

typedef struct {
    float x;
    float y;
} point;

#endif
```

Om definitionen inte är gjord...
...gör definitionen,
dvs. inkludera filen...

...annars: inkludera ingenting
(ingen #else-del!)

Externa variabler

- Vi har tidigare skapat variabler som varit globalt tillgängliga i **en** fil
- Hur göra variablerna tillgängliga i flera filer?
- Samma princip som för funktioner:
- *Deklarera* den globala variabeln i en headerfil med nyckelordet **extern**
- *Definiera* den globala variabeln i **en** av c-filerna
- *Inkludera* headerfilen i de c-filer som använder den globala variabeln

Externa variabler

```
// main.c

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

int main(void)
{
    printf("%d\n", global);
    global = 2;
    foo();
    printf("%d\n", global);
    return 0;
}
```

```
// foo.h
```

```
// declarations
extern int global;
void foo(void);
```

```
// foo.c
```

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

```
// definitions
int global = 1;
```

```
void foo(void)
```

```
{
    printf("%d\n", global);
    global = 3;
    printf("%d\n", global);
}
```

Strängar i C

- Många programmeringsspråk har en speciell datatyp för strängar – dock inte C
- I C är en sträng helt enkelt en räkka av tecken (array of **char**)

=> Ett klumpigt sätt att skapa en sträng är att sätta in bokstäverna “en och en” i en räkka:

```
char myString[10];  
myString[0] = 'H';  
mystring[1] = 'e';  
...
```

Strängar i C

- En sträng kan också initialiseras på ett enklare sätt:
char myString[] = "Hello World!";
- En strängvariabel kan skrivas ut med **%s** i en **printf()**-sats:
printf("This is my string: %s\n", myString);
- En sträng kan läsas in från användaren med **scanf**. I detta fall behövs **inte** något **&**-tecken framför variabeln:
scanf("%s", myString);

Strängar i datorns minne

- En sträng är en array och lagras därmed på samma sätt som en vanlig räkka i datorns minne
- Jobbigt att hålla reda på när en sträng tar slut, dvs. hur många positioner det finns i räkkan

word[0]	'H'
word[1]	'e'
word[2]	'l'
word[3]	'l'
word[4]	'o'
word[5]	'!'
word[6]	'\0'

=> för att slippa lagra denna information används en “null character” `'\0'` för att indikera att strängen tagit slut

- Sätts in *automatiskt* om strängen initialiseras samtidigt som den deklarereras

Strängar och funktioner

- En sträng är en räkka, så funktioner fungerar på motsvarande sätt som för andra typer av räckor:

```
int countChar(char searchString[], char toFind)
{
    int noOfOccurences = 0;
    // Search for the # of occurences
    // of toFind in searchString...

    return noOfOccurences;
}

char myString[] = "I'm a great programmer";
int result = countChar(myString, 'm');
// result should now be 3
```


Funktioner för stränghantering

- Metoden att läsa in strängar med **scanf()** fungerar om man endast vill läsa in ett ord åt gången
- **scanf()** tillåter att man t.ex. använder mellanslag för att separera "input items"
=> att läsa in texten **Hello World** som en enda sträng fungerar **inte**
- Vanligt att man vill *jämföra* innehållet i två strängar med varandra – men en jämförelse (**string1 == string2**) fungerar **inte**
- Exempel på andra vanligt förekommande uppgifter:
Sök efter en *delsträng* inne i en annan sträng,
kombinera två strängar

Funktioner för stränghantering

- C:s standardbibliotek **stdio** och **string** innehåller funktioner för stränghantering
- Appendix B i kursboken räknar upp de olika funktioner som finns tillgängliga
- Gemensamt för alla biblioteksfunktioner som hanterar strängar är att de egentligen använder sig av *pekare* till strängar (oviktigt i detta skede)

Funktioner för stränghantering

- **fgets(s, n, stdin)** – Läser in högst **n-1** bokstäver som lagras i strängen **s** från tangentbordet (**=stdin**)
- **getchar()** för att läsa in endast ett tecken – alternativ till **scanf("%c", ch)**
- **strncat(s1, s2, n)** – Konkatererar (lägger till) högst **n** tecken från strängen **s2** efter strängen **s1**
- **strcmp(s1, s2)** – Jämför strängen **s1** och strängen **s2**
- **strncpy(s1, s2, n)** – Kopierar högst **n** tecken från **s2** till **s1**
- **strstr(s1, s2)** – Söker efter **s2** i **s1**
- **strlen(s1)** – Kontrollerar längden på **s1**

Funktioner för stränghantering

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

int main(void) {
    char row1[100] = ""; // Init to empty string
    char row2[100] = ""; // Ditto

    printf("Enter two sentences!\n");
    fgets(row1, 100, stdin); // Read at most 99 chars,
                             // adds '\0' to end of string
    fgets(row2, 100, stdin); // stdin means 'Standard input'

    if (strcmp(row1, row2) == 0) {
        printf("The two sentences are the same!\n");
    }

    return 0;
}
```

Funktioner för stränghantering

- Tillgång till inbyggda strängfunktioner förenklar programmerarens liv
- Ändå nyttigt att studera hur motsvarande funktionalitet skulle kunna implementeras utan tillgång till standardbiblioteken
- Exempel: **equalstrings.c** i modellösningssmappen
 - Jämför innehållet i två strängar
 - Endast true/false som resultat
 - Hur kan vi modifiera funktionen för lexikografisk (alfabetisk) jämförelse av strängar?

Strängar och structs

- En struct kan också innehålla räckor – och därmed även strängar

```
struct textualDate {  
    char weekday[10];  
    int day;  
    char month[10];  
    int year;  
};
```

```
struct textualDate today = { .weekday = "Wednesday",  
                             .day = 23,  
                             .month="October",  
                             .year = 2017 };
```

Strängar och räckor

- Kan lagra många strängar i en räkka
=> Tvådimensionell räkka
- Måste bestämma maxlängd för strängarna då räckan skapas

```
char weekdays[7][10] = {"monday", "tuesday",  
                        "wednesday", "thursday",  
                        "friday", "saturday", "sunday"};  
  
printf("%s\n", weekdays[3]); => output is "thursday"
```

Pekare (Pointers)

- Variablers innehåll lagras som bekant i datorns minne
- Datorns minne är organiserat i form av minnesceller, och varje minnescell har en adress
- En *pekare* “pekar på” en viss adress i datorns minne; via denna pekare kan man komma åt innehållet i denna minnescell
- Pekare används direkt eller indirekt i de flesta programmeringsspråk men i C och C++ har pekare en speciellt framträdande roll
- En förståelse för hur pekare och minneshantering fungerar är nödvändig för att kunna skriva effektiva program (oberoende av vilket språk som används)

Användningsområden för pekare i C

- Tillåt en funktion att *modifiera* variablers innehåll i den anropande funktionen
- Manipulera och traversera (=gå igenom) räckor på ett effektivt sätt
- Dynamisk minneshantering (skapa datastrukturer som växer och krymper vid behov)
- Skicka *funktioner* som parametrar till andra funktioner
- I denna kurs bekantar vi oss med de tre första användningsområdena

Faror med pekare

- Pekare medför stora möjligheter, men också risker
- Buggar pga felaktig användning av pekare ger upphov till fel som kan uppstå och uppträda på ett slumpmässigt sätt
- En felaktigt använd pekare kan peka vart som helst i datorns minne – och den platsen kan variera för varje gång programmet körs
- Beroende på vart pekaren råkar peka vid en viss programkörning kan helt olika fel uppstå

=> svårt att hitta och korrigera felen

Deklaration av pekare

- Det vanligaste sättet att använda pekare är genom att deklarera en *pekare till en variabel*.
- För att deklarera en pekare används en *asterisk (*)*:

```
int* myPointer; //kan även skrivas int *myPointer
```

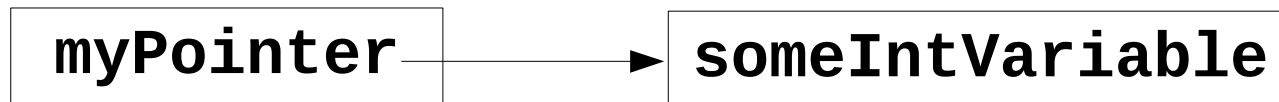
- Du har nu en namngiven pekare **myPointer** som är avsedd att “peka på” en **int**.
- Samma regel som för vanliga variabler: en pekare som inte är initialiserad kan peka vart som helst.

myPointer → ?

Initialisering av pekare

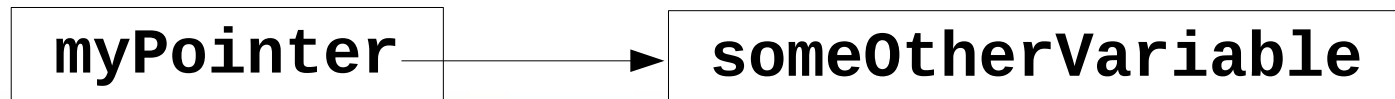
- En pekare initialiseras genom att tilldela den en minnesadress. En variabels minnesadress kan erhållas med adressoperatoren &

```
int someIntVariable = 13;  
int* myPointer = &someIntVariable;
```



- Pekarens värde kan därefter ändras så att den istället pekar mot en annan variabel:

```
int someOtherVariable = 26;  
myPointer = &someOtherVariable;
```



“Dereferencing” av pekare

- Med hjälp av “dereference” eller “indirection”-operatorn * kommer man åt *värdet* som finns i den minnesadress som pekaren hänvisar till:

```
int someIntVariable = 13;  
int* myPointer = &someIntVariable;  
printf("Content of myPointer = %d\n",  
                                             *myPointer);
```

- **Viktigt:** Pekaren och variabeln den pekar på är sammanlänkade!

=> Om värdet på variabeln förändras kommer samma förändring att synas även via dereferering av pekaren

=> Värdet på variabeln kan ändras *via pekaren*

“Dereferencing” av pekare

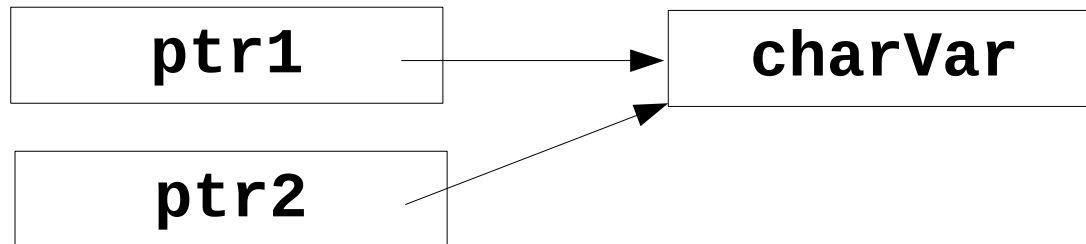
```
int varA = 40;  
int varB = varA;  
  
varA = 100;  
  
printf("Value of varA: %d\n",  
       varA);  
  
printf("Value of varB: %d\n",  
       varB);
```

```
int varA = 40;  
int* pointer = &varA;  
  
varA = 100;  
printf("Value of varA: %d\n",  
       varA);  
  
printf("Value of *pointer: %d\n",  
       *pointer);  
  
*pointer = 80;  
printf("Value of varA: %d\n",  
       varA);
```

Två pekare, en variabel

- Två pekare kan peka *till samma variabel*:

```
char charVar = 'a';  
char* ptr1 = &charVar;  
char* ptr2 = &charVar;
```



Fråga: Vad skulle hända om vi skulle skriva

a) `ptr2 = ptr1` b) `ptr2 = &ptr1` ?

Fråga: Varför är det viktigt att alltid ange *vilken typ av data* en pekare skall hänvisa till? Vore det inte enklare att endast ha en sorts pekare?

NULL-pekare

- En oinitialiserad pekare kan peka vart som helst
=> Mycket viktigt att initialisera pekare
- Kan ge en pekare startvärdet **NULL** eller **0**
- En funktion som returnerar en pekare returnerar typiskt en 'nullpointer' om operationen misslyckas

NULL-pekkare

```
int* pointerUsingFunction()
{
    int* ptr = NULL;
    // ...update ptr if possible
    return ptr;
}

int main(void)
{
    int* result = pointerUsingFunction();
    if (result != NULL ) // Could also write if (result)
    {
        // Use result
    }
    return 0;
}
```

Pekare som funktionsparametrar

- Vad blir resultatet från nedanstående program?

```
void test(int* intPointer) {  
    *intPointer = 100;  
}  
  
int main(void) {  
    int i = 50;  
    int* p = &i;  
    test(p);  
    printf("i is now %d\n", i);  
    printf("Dereferencing p gives %d\n", *p);  
  
    return 0;  
}
```

Pekare som funktionsparametrar

- Med hjälp av pekare kan en funktion ändra på de parametrar som sänts till den, och dessa ändringar kommer att synas i den anropande funktionen
- Möjliggör att en funktion returnerar mer än ett värde, utan användning av strukturer eller räckor
- Att skicka pekare kan vara mera effektivt än att skicka “normala” variabler, även om dessa inte behöver modifieras
 - Då en struktur skickas som en vanlig parameter måste alla dess medlemsvariabler kopieras – om samma struktur skickas som en pekare behövs inte detta.

Pekare som funktionsparametrar

- Det är inte nödvändigt att skapa en pekarvariabel för att skicka ett värde som en pekare (jfr. **scanf()** !)

```
void test(int* intPointer, float* floatPointer) {
    *intPointer = 100;
    *floatPointer = 3.14;
}

int main(void) {
    int intVar = 50;
    float floatVar = 9.2;
    test(&intVar, &floatVar); // Send the address of
                               // the variables
    printf("intVar is now %d\n", intVar);
    printf("floatVar is now %f\n", floatVar);
    return 0;
}
```

Pekare och struct

- Kan skicka en pekare till en **struct** som parameter
- Två möjligheter för att använda medlemsvariablerna:
 - Dereferencing
 - “Pilsyntax”

```
void test(struct date* dt) {  
    (*dt).day=10;    // Dereference  
    dt->day=10;      // Arrow-syntax  
}  
  
int main(void) {  
    struct date christmas = {24,12,2017};  
    test( &christmas );  
    return 0;  
}
```

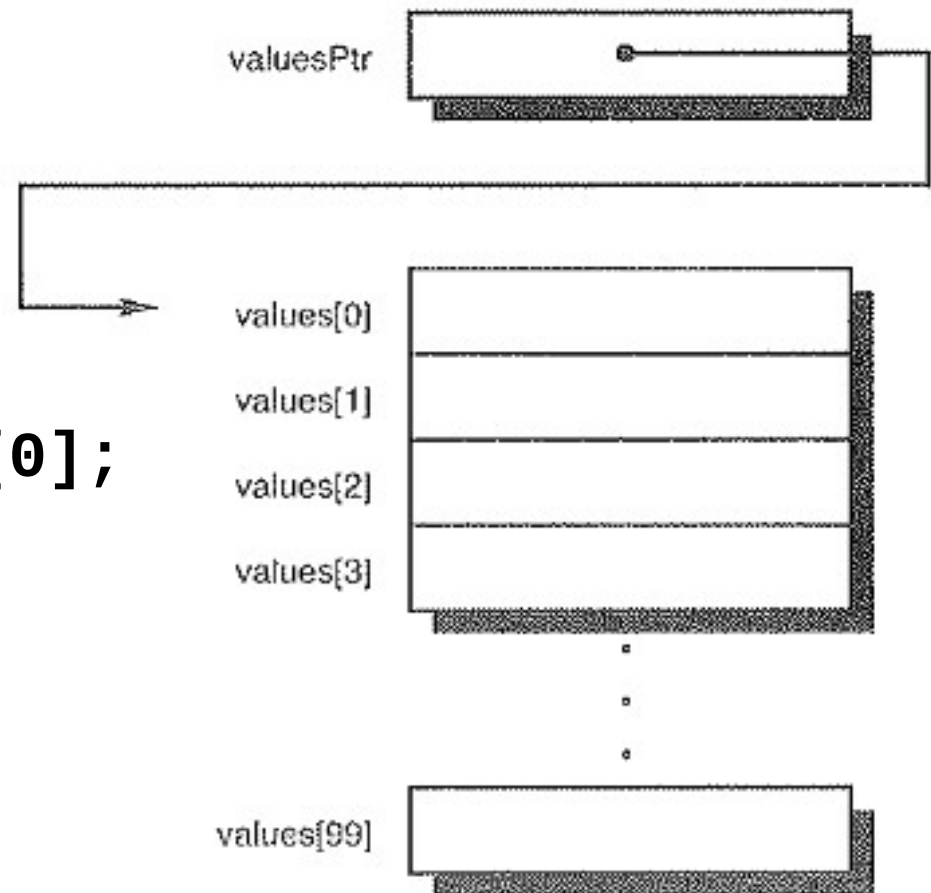
Pekare och räckor

- I C/C++ är pekare och räckor tätt sammankopplade
- För att skapa en pekare till (början av) en räckor behövs ingen **&**-operator:

```
int values[100];  
int* valuesPtr = values;
```

- Kunde också ha skrivit:

```
int* valuesPtr = &values[0];
```



Pekare och räckor

- För att “hoppa” till nästa element i räckan kan man helt enkelt addera en etta till pekaren:

```
int values[100];  
int* valuesPtr = values;  
  
valuesPtr++; // Nu pekas det på det 2:a  
             // elementet i räckan  
  
valuesPtr = valuesPtr + 10; // ...och nu på  
                           // det 12:e  
  
*valuesPtr = 437234; // Det 12:e elementet har  
                   // nu värdet 437234
```

Pekare och räckor

- När en räckan skickas som parameter till en funktion är det i verkligheten endast en *pekare* till räckan som skickas.
 - Förklarar varför **scanf** inte kräver något **&**-tecken för att läsa in en sträng
 - Förklarar varför innehållet i en räckan kan modifieras från en funktion
- Kan vara mera effektivt att använda sig av pekare istället för index för att gå igenom (traversera) räckor

Is it an array or is it a pointer?

```
int arraySum(int array[], int n) {
    int sum = 0;
    int i = 0;
    for (; i < n; i++) {
        sum = sum + array[i];
    }
    return sum;
}

int main(void) {
    int values[10] = {3, 4, 6, 6, 3, 8, -1, 5, 3, 7};
    printf("The sum is %d\n", arraySum(values, 10));
    return 0;
}
```

Is it an array or is it a pointer?

```
int arraySum(int* array, int n) {
    int sum = 0;
    int* arrayEnd = array+n-1    // Create a pointer to the
                                // last element in the array

    for (; array <= arrayEnd; array++) {
        sum = sum + *array;
    }
    return sum;
}

int main(void) {
    int values[10] = {3, 4, 6, 6, 3, 8, -1, 5, 3, 7};
    printf("The sum is %d\n", arraySum(values, 10));
    return 0;
}
```

Textsträngar och pekare

- Finns vissa skillnader mellan användning av strängpekare respektive räckor:

```
char text[80] = "This is OK!";  
text = "Will this work?"; // Not valid, only works when  
                           // initializing the string array  
  
text[0] = 'X'; // Valid, can modify the string  
  
char* textPtr = "This is also OK!";  
textPtr = "This works fine!"; // Valid  
textPtr[0] = 'X'; // Not valid. Will compile but  
                  // may cause a run-time crash  
                  // Must use dynamic memory  
                  // allocation for this to  
                  // work!
```

Textsträngar och pekare

```
void copyString( char* from, char* to ) {  
    while ( *from != '\0' ) {  
        *to = *from;  
        to++;  
        from++;  
    }  
    *to = '\0'; // Why is this needed?  
}  
  
int main(void) {  
    char string1[] = "Copy me!";  
    char string2[50];  
    copyString(string1, string2);  
    printf("string1: %s string2: %s\n", string1, string2);  
    return 0;  
}
```

Funktioner som returnerar pekare

- Fungerar nedanstående program korrekt?

(Tips: Lokala variablers giltighetsområden)

```
int* getMagicNumber() {  
    int blackMagic = 42;  
    return &blackMagic;  
}  
  
int main(void) {  
    int* result = getMagicNumber();  
    printf("Got the magic number: %d", *result);  
    return 0;  
}
```

Minnesadresser

- Vi kan också skriva ut den minnesadress som pekaren hänvisar till:

```
int a = 39;
int* a_ptr = &a;
printf("Address is %p\n", a_ptr); // Output in
                                   // hexadecimal format

int arr[5] = {1, 2, 3, 4, 5};
int* ptr_to_elem_3 = &arr[2];
int* arr_ptr = arr;
arr_ptr += 2;
printf("%p %p\n", ptr_to_elem_3, arr_ptr); // Output?
```

Nyckelordet **const**

- En variabel vars värde aldrig kommer att ändras kan deklarerars som en konstant med nyckelordet **const**
=> värdet på denna variabel kan inte ändras efter initialiseringen

```
const int meaningOfLife = 42;
```

- Möjliggör effektivare kod
- Minskar risken för buggar
- Speciellt användbart i kombination med **pekare som funktionsparametrar**

Konstanta variabler, exempel

```
int countChars( char input[] )
{
    int i=0;
    while (input[i] != '\0')
    {
        input[i] = 'X';
        i++;
    }
    return i;
}
```

Vad händer?

Hur kan problemet lösas?

```
int main(void)
{
    const char str[] = "Hello Dolly!";
    int result = countChars(str);

    printf("Length of %s is %d\n", str, result);
    return 0;
}
```


Pekare och const

- Använd const för att förhindra oavsiktlig modifikation av pekare (*const correctness*):

```
void test(const struct date* dt) {  
    (*dt).day=10;    // Compiler error  
    dt->day=10;      // Compiler error  
}  
  
int main(void) {  
    struct date christmas = {24,12,2017};  
    test( &christmas );  
    return 0;  
}
```

Enumerationer

- Vanligt problem: En variabel får endast ha vissa tillåtna värden
- Exempel: Veckodagar, månader, svarsalternativ i en meny...
- Inte bra att använda “magiska nummer”
 - gör koden svårläst och risken för fel ökar

```
int choice = getUserChoice();  
if (choice == 0) { calculateCirc(); }  
if (choice == 1) { calculateArea(); }
```

- Ett bättre sätt är att definiera konstanter:

```
const int CALCCIRCUM = 0;  
const int CALCAREA = 1;  
  
int choice = getUserChoice();  
if (choice == CALCCIRCUM) { calculateCirc(); }  
if (choice == CALCAREA) { calculateArea(); }
```

Enumerationer

- Ännu bättre sätt: Definiera en namngiven mängd med namngivna heltalskonstanter => *enumeration* (“uppräkning”)
- Som standard motsvarar den första konstanten heltalet 0, följande konstant 1, osv.

```
enum menuChoices { CALCCIRCUM, CALCAREA };  
enum menuChoices choice = getUserChoice();  
if (choice == CALCCIRCUM) { calculateCirc(); }  
if (choice == CALCAREA) { calculateArea(); }
```

- Större exempel: **gradecalc.c** i exempelmappen på Moodle
- Visar också hur man använder **switch..case** för att kontrollera enumerationens värde

Enumerationer

- Varje enumeration är egentligen ett heltalsvärde
- Anges inga explicita värden tilldelas den första enumerationen värdet 0, den andra enumerationen värdet 1, osv.

- Kan också ange ett eget startvärde för numreringen:

```
enum month { JANUARY = 1, FEBRUARY, MARCH, .... };
```

- ...och tilldela separata värden för en del eller alla enumerationer

```
enum priority { LOW = 0,  
                MEDIUM = 50,  
                ALMOST_MEDIUM, // får värdet 51  
                HIGH = 100 };
```

Enumerationer

- Har tidigare sett exempel på hur **typedef** kan användas för att skapa alias för en **struct**
- Kan också använda **typedef** för att förenkla användningen av enumerationer:

```
typedef enum {  
    MONDAY=1, TUESDAY, WEDNESDAY, THURSDAY,  
    FRIDAY, SATURDAY, SUNDAY  
} weekday;
```

```
weekday w = TUESDAY;
```

Enumerationer för tillstånd

- En del programmeringsuppgifter kan lösas genom att observera att programmet kan befinna sig i olika *tillstånd* (eng. *states*)
- Antalet tillstånd är begränsat => kan använda en enumeration för att räkna upp de tillstånden och skapa en *tillståndsvariabel* av denna typ för att hålla reda på det aktuella tillståndet
- Typiska exempel: (nätverks)kommunikation enligt ett visst protokoll, (delar av) spel...

Enumerationer för tillstånd

```
typedef enum  
{
```

```
    HEALTHY,  
    WOUNDED,  
    NEAR_DEATH,  
    DEAD
```

```
} ai_status;
```

```
ai_status status = HEALTHY;
```

```
...
```

```
switch (status)
```

```
{
```

```
    case HEALTHY: { normal_attack(); }
```

```
    case WOUNDED: { flee(); }
```

```
    case NEAR_DEATH: { kamikaze_attack(); }
```

```
}
```

Filhantering

- Repetition:
 - Funktioner för att läsa in data från tangentbordet: **scanf()**, **getchar()** och **fgets()**
 - Funktioner för att skriva ut data till skärmen: **printf()** och **putchar()**
- Kan också läsa och skriva från filer:
 - Med hjälp av *redirection*: Ett program kan instrueras att läsa från eller skriva till en viss fil genom att ange denna fil *då programmet startas från kommandoprompten/terminalen*
 - Genom att öppna och sedan läsa/skriva till namngivna filer *inne i programmet*

Redirection

- För att styra all utdata från programmet **a.out** till en fil **out.txt**:
./a.out > out.txt
- För att läsa in all indata till programmet **a.out** från en fil **in.txt**:
./a.out < in.txt
- För att läsa från **in.txt** och skriva till **out.txt**:
./a.out < in.txt > out.txt
- Inte specifikt för c-program, kan styra input/output från vilket program som helst

“End-Of-File”, EOF

- Vid inläsning av data från en fil är det viktigt att veta när slutet på filen påträffas
 - Ofta består programmet av en loop som läser in ett ord eller en rad i taget, tills all information i filen är inläst
- De flesta inputfunktioner returnerar ett speciellt heltalsvärde, **EOF**, när slutet på en fil påträffas.
- Följande exempel använder **getchar()**, **putchar()** och **EOF** för att kopiera en fil mha. redirection.

“End-Of-File”, EOF

```
/* Program to copy a file.
Usage: ./a.out < infile > outfile */

#include <stdio.h>

int main(void) {
    int c;

    /* Note that getchar returns an int, not a char.
       Read one character at a time, until we find EOF */
    while ( (c = getchar()) != EOF ) {
        putchar(c);
    }

    return 0;
}
```

Arbeta med namngivna filer

- För att t.ex. läsa eller skriva till flera filer, skapa ett mera användarvänligt gränssnitt och implementera bättre felhantering räcker “redirection”-metoden inte till
- Måste istället användas c:s inbyggda funktioner för filhantering
- Allmän princip:
 - Försök öppna filen och kontrollera att operationen lyckades
 - Läs eller skriv till filen
 - Stäng filen
- Filer kan öppnas för läsning, skrivning eller uppdatering

Öppna filer: **fopen**

- **FILE*** **fopen**(**const char*** path,
 const char* mode)
- Returvärdet är en speciell “filpekare” som senare kan användas för att läsa till eller skriva från denna fil
- **path** är sökvägen till filen
- **mode** är en sträng som beskriver hur filen skall öppnas:

"r" => läs

"r+" => läs och skriv

"w" => skapa ny fil och skriv

"w+" => skapa ny fil, läs och skriv

"a" => lägg till data

"a+" => läs och lägg till data

Läs från filer: **getc, fscanf, fgets**

- **int getc(FILE* stream)**
 - Läs ett tecken, motsvarar **getchar()**
 - Returnerar **EOF** när slutet av filen påträffas
- **int fscanf(FILE* stream,
 const char* formatString, ...)**
 - Motsvarar **scanf()**
 - Returnerar **EOF** när slutet av filen påträffas
- **char* fgets(char* buffer, int maxsize,
 FILE* inputStream)**
 - Läser in en rad i taget
 - Returnerar **NULL** (0) när slutet av filen påträffas

Skriv till filer:

putc, fprintf, fputs

- **int putc(int c, FILE* stream)**
 - Skriv ett tecken, motsvarar **putchar()**
 - Returnerar **EOF** om fel uppstår
- **int fprintf(FILE* stream, const char* formatString, ...)**
 - Motsvarar **printf()**
 - Returnerar ett negativt värde om fel uppstår
- **int fputs(const char* buffer, FILE* inputStream)**
 - skriver en rad i taget
 - Returnerar **EOF** om fel uppstår

Stänga filer: **fclose**

- **int fclose(FILE* fp)**
- Returnerar **EOF** om fel uppstår
- Viktigt att stänga filer eftersom:
 - Data som skickas till en fil inte alltid omedelbart skrivs till filen. **fclose** tömmer ('flushar') utdatabuffern och stänger sedan filen
 - Kan använda **fflush()** för att flusha filen utan att stänga den
 - Antalet filer som samtidigt kan vara öppna är begränsat, och varje öppnad fil kräver minne

Arbeta med namngivna filer, exempel

```
/* Program to display a file. */
#include <stdio.h>

int main(void) {
    FILE* infile = fopen("/home/joakim/test.txt", "r");

    if (infile) // Check if file could be opened
    {
        int nextChar;
        while ( ( nextChar = getc( infile ) ) != EOF )
        {
            putchar(nextChar);
            /* Could also have used e.g.
               putc(nextChar, stdout); */
        }
        fclose(infile);
    }
    return 0;
}
```

Speciella filer: **stdout, stdin, stderr**

- Filer eller “strömmar” (*streams*) som automatiskt öppnas när ett program startas, och stängs när programmet avslutas
- Alla tre filer motsvarar normalt terminalfönstret där programmet körs
- **stderr** används för felmeddelanden – kan t.ex. styra felmeddelanden till en viss fil
- Funktionen **freopen()** kan användas för att ändra på destinationen för dessa filer:

```
FILE* freopen( const char* path,  
               const char* mode, FILE* stream )
```

- För att t.ex. skriva felmeddelanden till **/home/joakim/errors.log**:
freopen("/home/joakim/errors.log", "a", stderr);

stderr, exempel

```
freopen("errors.log", "a", stderr);

char filename[] = "foo.txt";
FILE* f = fopen(filename, "w");
if (f==NULL)
{
    // Output error message to errors.log
    fprintf(stderr,
              "Could not open %s for writing!\n",
              filename);
    exit(0);
}

// Do some writing...

fclose(f);
```

Programparametrar

- Parametrar som skickas till **main** när programmet startas
- Deklarera main enligt följande:
int main(int argc, char* argv[])
- Det andra argumentet är en räkka som innehåller pekare till strängar
- Det första elementet anger hur många element räkkan innehåller
- Exempel: Om ett program startas med två parametrar:

./a.out test1 test2

kommer det *första* elementet i **argv** att vara namnet på programmet, dvs. “./a.out”. Det *andra* elementet kommer att vara “test1” och det *tredje* elementet kommer att vara “test2”. **argc** kommer att ha värdet 3

Programparameter

```
#include <stdio.h>

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

Dynamisk minneshantering

- När en variabel deklarerats reserveras exakt så mycket minne som behövs för denna variabel – t.ex. 1 byte för en **char**, 4 bytes för en **int**, 80 bytes för en räkka med 20 **int**...
- Tre sätt att deklarera en räkka:
 - a) Skapa en räkka där antalet element är fastslaget vid kompileringen: **int myArray[20];**
 - b) Skapa en räkka där antalet element är fastslaget när räkkan skapas, med hjälp av 'variable-length arrays'
int sizeofArray = determineNeededLength();
int myArray[sizeofArray];
 - c) Allokera minne *dynamiskt* när programmet körs, förändra storleken på räkkan efter behov

Dynamisk minneshantering: malloc

- Funktionen **malloc()** används för att reservera minne
- **malloc()** tar som parameter *antalet bytes* som skall reserveras, och returnerar en *pekare* till den första adressen i detta minnesområde
- För att med säkerhet veta hur många bytes som behövs för att t.ex. reservera minne för ett visst antal integers kan funktionen **sizeof()** användas:

```
malloc(sizeof(int)*10); // reserves memory  
                        // for 10 integers
```

Dynamisk minneshantering: malloc

- **malloc** returnerar en speciell typ av pekare: en pekare till **void**
- Kan här tolkas som “en pekare till vad som helst”.
- Programmeraren måste sedan själv omvandla denna pekare till korrekt datatyp genom en *type cast*:

```
int* pointer =  
    (int*) malloc(sizeof(int)*10);
```

- En minnesallokering kan misslyckas (dvs minnet kan ta slut!)
I så fall returnerar **malloc** en “nullpekare” (**NULL**)

Dynamisk minnesallokering: malloc

```
#include <stdio.h>
#include <stdlib.h> // required for malloc()

int main(void) {
    int size = 0;
    int* arrayPointer = 0;
    printf("How many elements? ");
    scanf("%d", &size);
    arrayPointer = (int*) malloc(sizeof(int) * size);
    if (arrayPointer == NULL) // Or like this: if (!arrayPointer)
    {
        printf("Allocation failed! Out of memory?\n");
        return 0;
    }
    // Do something nice with your array
    return 0;
}
```

Dynamisk minneshantering: **realloc**

- **realloc()** används för att öka eller minska storleken på ett reserverat minnesområde:
- Liksom **malloc()** returnerar **realloc()** en nullpointer om minnesområdet inte kan allokeras
- **realloc()** kan välja att utöka det existerande minnesområdet, eller att allokera ett helt nytt område
=> vi kan **inte** utgå från att den gamla datan ligger kvar på samma plats i minnet efter ett **realloc()**-anrop

Dynamisk minnesallokering: realloc

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

int main(void) {
    int size = 10;
    int* arrayPointer = (int*) malloc(sizeof(int)*size);
    // ...
    // Double the space
    arrayPointer = (int*) realloc(arrayPointer,
                                   sizeof(int)*(size*2));
    // ...
    // Make the array smaller
    arrayPointer = (int*) realloc(arrayPointer,
                                   sizeof(int)*(size/4));
    // ...
    return 0;
}
```

Dynamisk minneshantering: Minnesläckor

- Vad blir effekten av nedanstående kod?

```
arrayPointer = (int*) malloc(sizeof(int)*size);  
...  
// Double the space  
arrayPointer = (int*) malloc(sizeof(int)*size*2);
```

- Det andra **malloc()**-anropet allokerar nytt minne
- Det tidigare minnesområdet är fortfarande reserverat för programmets räkning, men det finns inte längre någon pekare till detta minnesblock

=> Vi har åstadkommit en *minnesläcka*

Minnesläcka i en hiss (pseudokod)

- When a button is pressed:
- Get some memory, which will be used to remember the floor number
- Put the floor number into the memory
- Are we already on the target floor?
 - If so, we have nothing to do: finished!
- Otherwise:
 - Wait until the lift is idle
 - Go to the required floor
 - Release the memory we used to remember the floor number

Dynamisk minneshantering: free

- I de flesta fall – **men inte alltid** – kommer det dynamiskt reserverade minnet att frigöras när programmet avslutas
- Säkrast att alltid frigöra dynamiskt minne manuellt när det inte längre behövs
- Minnesläckor i t.ex. operativsystem eller inbyggda system är speciellt allvarliga – varför?

Dynamisk minneshantering: free

- För att frigöra dynamiskt allokerat minne används funktionen **free()**:

```
arrayPointer =  
    (int*) malloc(sizeof(int)*size);
```

...

```
// Release the memory  
free(arrayPointer);
```

- Vanliga misstag:
 - Försök att frigöra ett minnesområde som inte har allokerats dynamiskt
 - Försök att frigöra samma minnesområde två gånger

Preprocessorn

- Ett c-program innehåller ofta *preprocessordirektiv*.
- Instruktioner som analyseras *före* själva kompileringen av programmet
- Alla preprocessordirektiv inleds med “brädgårdsmärket” #
- Vi har alltså redan träffat på flera preprocessordirektiv:
#include för att inkludera *headerfiler*
#ifndef, **#define**, **#endif** för *include guards*
- Andra vanliga preprocessordirektiv:
#ifdef, **#undef**, **#if**, **#elif**, **#else**

#define

- **#define** används för att namnge konstanter
- För att definiera en konstant **TRUE** med värdet 1 och en konstant **FALSE** med värdet 0:
#define TRUE 1
#define FALSE 0
- Observera syntaxen: Inget **=** mellan konstantens namn och dess värde, inget semikolon efter instruktionen!
- Konstanter skrivs ofta med versaler för att enkelt skilja dem från ickekonstanta värden
- Kan se **#define** som en “klipp-och-klistra” - instruktion till preprocessor: Ersätt alla förekomster av konstantens namn med det definierade värdet innan kompileringen startar.
- Tidigare har vi använt nyckelordet **const** för att definiera konstanter – vilken metod är bättre?

#define

- En **#define**-konstant kan omdefinieras;
en **const**-konstant kan inte omdefinieras

```
#define VERY_IMPORTANT_CONSTANT 42
const int ANOTHER_VERY_IMPORTANT_CONSTANT = 79;

int main(void) {

    // This works (although gcc emits a warning)
    #define VERY_IMPORTANT_CONSTANT 43

    // This gives a compile-time error
    ANOTHER_VERY_IMPORTANT_CONSTANT = 80;
}
```

Användningsområden för #define

- **#define** kan, i motsats till **const**, användas för att ange dimensioner för globala räckor (jfr. Chomp)
- **#define** kan användas för att definiera funktionsliknande makron, t.ex
#define SQUARE(x) x*x
#define SUM(x,y) x+y
...men normalt är det ändå bättre att göra “riktiga” funktioner med definierade datatyper
- **#define** kan användas för att implementera s.k. “conditional compilation”
 - “Include guards” är en variant av denna teknik

“Conditional compilation”

- Innebär att endast en del av programmet kompileras
- Typiska användningsområden:
 - Konstruktion av plattformsoberoende program
 - Debuggning
- Använd **#define** för att skapa en 'flagga' som indikerar t.ex. i vilken omgivning ett program skall köras, eller om debuggningsinformation skall skrivas ut
- Kompilera olika versioner av t.ex. samma funktion, beroende på flaggans värde.
- Kan kontrollera om en flagga **existerar** med **#ifdef**, och kontrollera **värdet** på en flagga med **#if**

“Conditional compilation” för debuggning

```
#define DEBUG
int main(void) {
    // ...part of the 'guess the secret number' game
    int secretNumber = rand()%100 + 1;
#ifdef DEBUG
    // Only output secret number if debug mode is on
    printf("Secret number is %d\n", secretNumber);
#endif
    printf("Enter a number between 1 and 100: ");
    scanf("%d", &num);
    ...
}
```

“Conditional compilation” för plattformsberoende

```
#define PLATFORM 1
...
#if PLATFORM == 1 // Windows
const char* PATH = "C:\\Windows\\System32";
#elif PLATFORM == 2 // Linux
const char* PATH = "/bin";
#elif PLATFORM == 3 // Mac
...
#else
// Unknown platform, assume Windows
const char* PATH = "C:\\Windows\\System32";
#endif

printf("Path is %s\\n", PATH);
```