By-value / by-reference

Tidigare sades att ALLA parametrar i C överförs by-value

Hur överförs då t.ex. strängar ??

```
char str1[30] = "Hej", str2[30];
strcpy(str2, str1);
```

Det som nu överförs är pekarens värde ”*by-value*” (str1, str2 är konstanta pekare), detta kunde i princip också kallas *by-reference*

By-value / by-reference

I C har du alltid kontroll över det minne du själv allokerat

→ Ingen funktion du anropar kan ändra ditt minne utan ”ditt tillstånd” (dvs för att en funktion skall kunna ändra lokala variabler, måste den få en pekare till denna variabel (by reference))

By value / by reference

```
int inc(int a) { /* denna funktion kan ej direkt
                    ändra värdet variabeln */
   return a+1;
}
void inc_ref(int *a) { /* denna funktion får en
                          pekare till variabeln, kan
                          direkt inkrementera
                          variabeln */
   *a++;
}

int main() {
   int b=0;
   b = inc(b);   /* by value: variabelvärdet */
   inc_ref(&b);  /* by value: pekaren till variabeln */
}
```

By value / by reference

```
#include <stdio.h>
void swap(int *, int *);

main() {
  int num1 = 5, num2 = 10;
  swap(&num1, &num2);
  printf("num1 = %d and num2 = %d\n", num1, num2);
}

void swap(int *n1, int *n2) { /* passed and returned by
                                 reference */
  int temp;

  temp = *n1;
  *n1 = *n2;
  *n2 = temp;
}
```

Räckor som parametrar

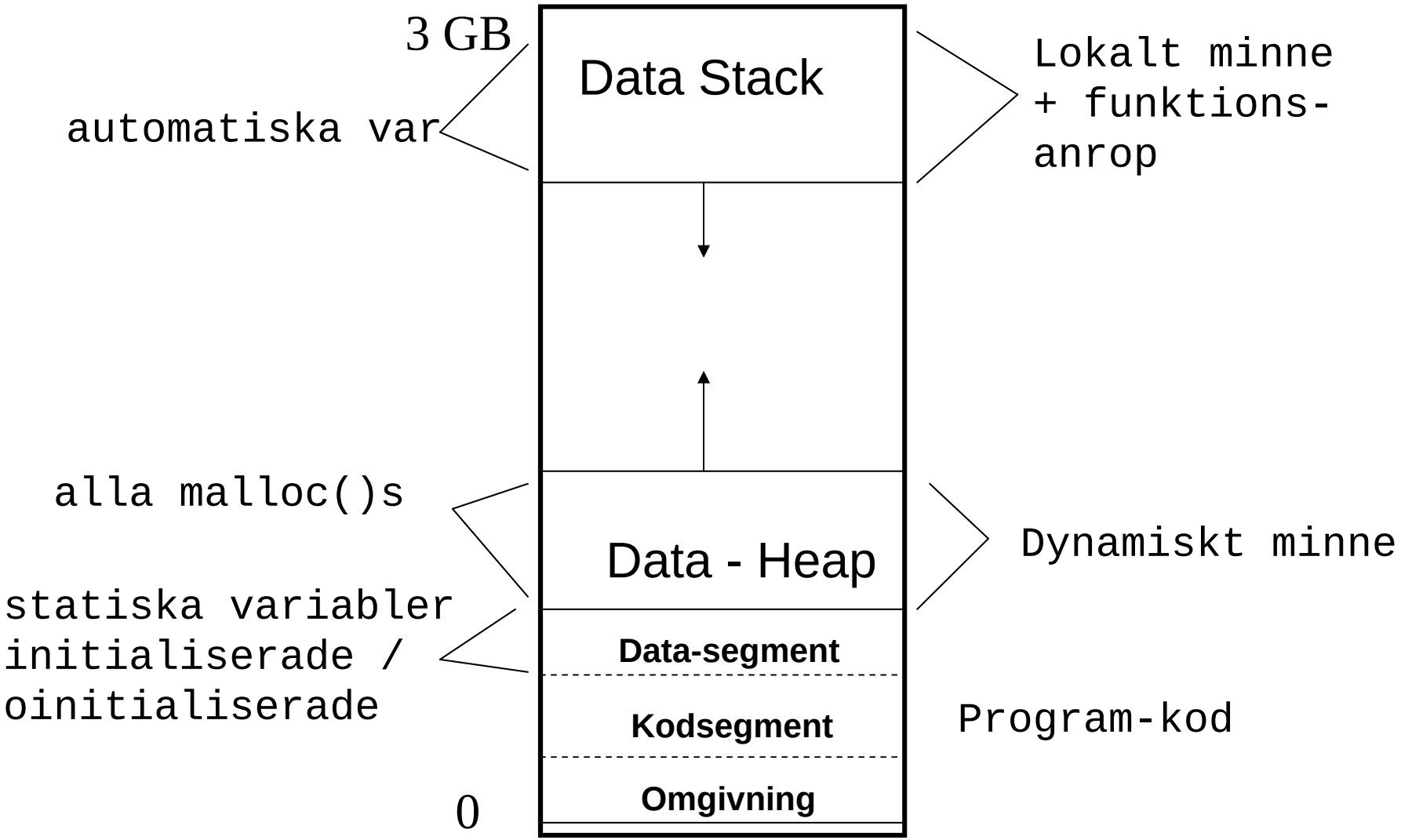
```
void init_array(int array[], int size) ;

int main(void) {
  int list[5];

  init_array(list, 5);
  for (i = 0; i < 5; i++)
    printf("next:%d", array[i]);
}

void init_array(int array[], int size) { /* varför storlek
   ? */
        /* räckor alltid by-reference = pekaren by-value */
 int i;
 for (i = 0; i < size; i++)
    array[i] = 0;
}
```

Minnesrymd för program



Pekare till funktioner

```
int func(); /*function returning integer*/
int *func(); /*function returning pointer to integer*/
int (*func)(); /*pointer to function returning integer*/
int *(*func)(); /*pointer to func returning ptr to int*/
```

- Fördel ? mera flexibilitet

Pekare till funktion - Exempel

```
#include <stdio.h>

void myproc (int d);
void mycaller(void (* f)(int), int param);

void main(void) {
      myproc(10);         /* call myproc with parameter 10*/
      mycaller(myproc, 10); /* and do the same again ! */
}

void mycaller(void (* f)(int), int param){
      (*f)(param);        /* call function *f with param */
}

void myproc (int d){
      . . .               /* do something with d */
}
```

Varför pekare till funktioner?

- Exempel 1:
  - Registrering av funktioner som kan behandla en viss datamängd
    - T.ex. funktioner i bildbehandling
- Exempel 2:
  - Då man registrerar en funktion som svarar på olika signaler, installering av signalhanterare

Mer komplicerat

Deklarera en räcka med N pekare till funktioner som returnerar pekare fill funktioner som returnerar pekare till tecken

1. char *(*(*a[N])())();

2. Build the declaration up in stages, using typedefs:

```
typedef char *pc; /* pointer to char */
typedef pc fpc(); /* function returning pointer to char
*/
typedef fpc *pfpc; /* pointer to above */
typedef pfpc fpfpc(); /* function returning... */
typedef fpfpc *pfpfpc; /* pointer to... */
pfpfpc a[N]; /* array of... */
```

Dynamisk minnesallokering

- Explicit allokering och deallokering

```
#include <stdio.h>

int main(void) {
      int *ptr;
            /* allokera utrymme för ett heltal */
      ptr = malloc(sizeof(int));

            /* används minnesutrymmet */
      *ptr=4;

      free(ptr);
            /* frigör det allokerade minnet */
}
```

Dynamiskt minne - funktioner

```
#include <malloc.h>
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *free(void *ptr);
void *realloc(void *ptr, size_t size);
```

malloc – returnerar pekare till minnesblock av storlek size

realloc – ökande / minskade av minne, gemensam det oförändrad

free – frigör minne på pekare ptr

Dynamisk minnesallokering - exempel

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

int main() {
   char *buffer;
   pPerson ptr_pers;

   buffer = (char *)malloc(50 * sizeof(char));
   ptr_pers = (pPerson)malloc(40 * sizeof(Person));

   if (!buffer) exit(-1); /* om malloc ej lyckades */
   if (!ptr_pers) exit(-1);
   strcpy(buffer, "Hej, här är texten");
   strcpy(ptr_pers[0].namn, "Axel Eklund");
   ptr_pers[0].alder = 32;
   return 0;
}
```

Dynamiska matriser

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

float ** matrix_alloc(int rows, int cols) {
   float **matr;
   int i;
   /* Allokera pekare till radräckor */
   matr = (float **)malloc(sizeof(float *)*rows);
   for(i=0; i<cols; i++) {
        /* Allokera radräckorna */
        matr[i] = (float *)malloc(sizeof(float)*cols);
   }
   return matr;
}
main() {
   float **m;
   m = matrix_alloc(20,20);  /* Allokera matrix */
   m[6][7] = 2.97;            /* Sätt element */
   return 0;
}
```

Minnesläckor

- Allokerat minne (malloc(), realloc(), calloc()) har inte ett motsvarande free()
- Med tiden växer den allokerade minnesmängden
  - I serverbruk med tiden katastrofalt
  - För klientprogram (typ webläsare) främst störande
- Hur kan man undvika
  - Allokering / deallokering på välkontrollerade platser
  - Noggrann felhantering (så att inte free() blirbortglömt p.g.a. ovanligt progamflöde)
  - Verktyg vid utvecklingsskedet

Minnesläckor - exempel

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

int myfunc() {
  char *buffer;

  buffer = (char *)malloc(50 * sizeof(char));
  if (!buffer) exit(-1); /* om malloc ej lyckades
  */
  scanf(“%d”, &newval);
  if (newval<0) return –1; /* error */
  …
  free(buffer); /* programflödet når inte alltid
  hit = möjlig minnesläcka */
}
```

Funktioner i bibliotek

- (Nästan) Alla C-program använder sig funktioner som systemet i sig erbjuder
- Dessa funktioner grupperas i bibliotek enligt den domän de behandlar
  - t.ex. kryptografi, näthantering, LDAP
- Det vanligaste ”standard”-biblioteket:
  - LIBC (finns automatisk tillgängligt vid länkning)
    - statisk länkning / dynamisk länkning

libc

- Memory: Allocating virtual memory and controlling paging.
- Character Handling: Character testing and conversion functions.
- String and Array Utilities: Utilities for copying and comparing strings and arrays.
- Character Set Handling: Support for extended character sets.
- Locales: The country and language can affect the behavior of library functions.
- Message Translation: How to make the program speak the user's language.
- Searching and Sorting: General searching and sorting functions.
- Pattern Matching: Matching shell ``globs'' and regular expressions.
- I/O Overview: Introduction to the I/O facilities.
- I/O on Streams: High-level, portable I/O facilities.
- Low-Level I/O: Low-level, less portable I/O.

...libc

- File System Interface: Functions for manipulating files.
- Pipes and FIFOs: A simple interprocess communication mechanism.
- Sockets: A more complicated IPC mechanism, with networking support.
- Low-Level Terminal Interface: How to change the characteristics of a terminal device.
- Syslog: System logging and messaging.
- Mathematics: Math functions, useful constants, random numbers.
- Arithmetic: Low level arithmetic functions.
- Date and Time: Functions for getting the date and time and formatting them nicely.
- Resource Usage And Limitation: Functions for examining resource usage and getting and setting limits.
- Non-Local Exits: Jumping out of nested function calls.
- Signal Handling: How to send, block, and handle signals.
- Program Basics: Writing the beginning and end of your program.

...libc

- Processes: How to create processes and run other programs.
- Job Control: All about process groups and sessions.
- Name Service Switch: Accessing system databases.
- Users and Groups: How users are identified and classified.
- System Management: Controlling the system and getting information about it.
- System Configuration: Parameters describing operating system limits.
- Cryptographic Functions: DES encryption and password handling.
- Debugging Support: Functions to help debugging applications.

Filer i C

- Nästan alla program använder sig av filer och input / output
- UNIX är mycket filkoncentrerat, de flesta operationer kan göras genom att läsa från / skriva till filer
  - filer på filsystem (det normala)
  - läsa från / skriva till hårdvaruenheter
  - läsa från / skriva till nätverket
  - m.m.
- Normal indelas normala filer i klasser
  - text filer och binära filer

Stöd för filer i libc

- Prototyper i
  - stdlib.h, stdio.h
- Öppna / stänga filer

```
FILE *fopen(const char *filename, char *mode);
int fclose(FILE *);
```

Mode
r = reading
w = writing
a = appending
r+ = reading and writing
w+ = reading and writing
a+ = reading and appending

Filer - exempel

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

int main() {
   FILE *fp;
   char [] filename = “results.txt”;
   fp = fopen(filename, “w”);
   if (fp == NULL) {
      printf(“Couldn’t open %s\n”, filename);
      exit(1);
   }
   fprintf(fp, “Hej\n”);
   close(fp);
}
```

Formatterad input / output

```
int fprintf(FILE *fp, char *fmt, …);
int fscanf(FILE *fp, const char *fmt, …);
```

- Fungerar som printf and scanf

```
fprintf(fp, “%s %s %u %u\n”, cog.first, cog.last,
   cog.ssn, cog.credit_rating);
fscanf(fp, “%s %s %u %u\n”, &cog.first, &cog.last,
   &cog.ssn, &cog.credit_rating);
```

- Skillnad på printf fprintf ??
  - Första parametern i **`fprintf`** ger filen som man skall skriva till
  - i **`printf`** är filen som defualt **`stdout`**, som finns färdigt deklarerad då prgram startas

Formattering - fprintf

```
int fprintf(FILE *fp, char *fmt, …);
```

- Formatsträng: fmt
  - %i,d – heltalsargument skriv i decimalform
  - %o,u,x, X – teckenlöst heltalsargument skrivs i: octalt (o), decimalt (u), hexadeximalt (x,X)
  - %e,E – flyttalsargument skrivs i formen (-)d.ddde(-)dd, t.ex. – 1.27e12
  - %f,F – flyttalsargument skrivs i formen (-)ddd.ddd
  - %c – heltalargument skriv ut som unsigned char
  - %s – pekarargumentet tolkas som pekare till sträng och skriv ut som eh nollterminerad sträng
  - %p pekarargument skriv i formen %x
- Dessutom finns
  - formatspecificerare för utskriftsbredd, precision, mm
  - t.ex. printf(”pi = %.5f\n”, 4*atan(1.0)) innebär attflyttalet skrivs ut med 5 siffrors noggrannhet

Binära filer

- Skall man använda sig av binära filer eller filer i textform ??
- Textform – enkelt att läsa och skriva, debuggning lättare, vanligen platform-oberoende
- Binär – snabbt att läsa skriva, enklare att hitta viss position, kompakt, svårare att debugga, måste hantera saker såsom little/big-endian

Binära filer - exempel

```
int fwrite(void *buf, int size, int
  count, FILE *fp);
int fread(void *buf, int size, int count,
  FILE *fp);


int buf[100];  int i;
fp = open(“numbers”,’rb’);
fread(buf, sizeof(int), 100, fp);
close(fp);
for (i = 0; i < 100; i++) buf[i] = buf[i]+1;
out = open(“numbers.out”,’wb’);
fwrite(buf, sizeof(int), 100, out);
close(out);
```

Binära filer –random access

- Varje FILE-struktur har en indikator för offset i filen
- void rewind(FILE *fp);
- long ftell(FILE *fp);
  - first byte is 0. Returns -1 if there's an error
- fseek(FILE *fp, long offset, int origin);

- origin can be three values
- SEEK_SET - moves indicator offset bytes from beginning
- SEEK_CUR - moves indicator offset bytes from current position
- SEEK_END – moves indicator offset bytes from end of file

Hitta slutet på en fil

- Många gånger är längden på en fil okänd
- I textfiler kan EOF-tecknet användas

```
while ((c =fgetc(fp)) != EOF)
    ...
}
```

- I binära filer

```
int feof(FILE *fp);
```

returnerar 0 on slutet på filen inte ännu är nått.
Returner annat än noll om filen är slut

Filfunktioner i libc <-> systemet

- UNIX (Linux)-varianter använder internt i OS heltal för att identifiera filer <int>
- libc anväder internt en datastruktur (FILE) för att buffra operationer till filer

```
libc: size_t fread( void *ptr, size_t
  size, size_t nmemb, FILE *stream);
OS: ssize_t read(int fd, void *buf,
  size_t count);
```

Sträng-hantering

- Prototyper för sträng hantering i **`string.h`**
- Funktioner för att
  - Beräkna längd på sträng (**`strlen`**)
  - Kopiera och förena strängar (**`strcpy, strcat`**)
  - Jämföra strängar (**`strcmp`**)
  - Söka i strängar (**`strchr, strrchr, strcspn`**)
  - Konvertering av strängar till numeriska värden (**`atoi, atol, atof`**)
  - Test för specifika värden (**`isalnum, isalpha, isdigit,`** …)

Stränghantering - exempel

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

main () {
   char str1[100] = "Testing string handling";
   char str2[100];
   char *p;

   int i;
   printf("Längd sträng1: %u\n", strlen(str1));
   strcpy(str2, str1);
        /* ~Ekvivalent med */
   for (i=0; i<=strlen(str1);i++) str2[i] = str1[i];
        /* Men … strängbiblioteket använder sig t.ex. av processorns
        Single-Instruction-Multiple-Data (SIMD) egenskaper */

   p = strstr(str1, "hand");
   printf("hand hittades på position %u i '%s'\n", p-str1, str1);
   return = 0;
}
```

Nätverk - funktioner

- BSD Socket
  - Ändpunkt för kommunikation
  - Fil som det går att skriva till / läsa från
    - Skapas med **`socket()`**
    - Kopplas med **`connect()`**
- Nätverk OBS. Host byte/bit order är kanske inte samma som på nätet
  - Funktioner som htons() – host to network byte order

Nätverk – läs från webserver

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

char msg[] = "GET / HTTP/1.0\n\n";
char retbuf[4096];

int main() {
  int sockfd;
  struct sockaddr_in raddr;
  int count;

  sockfd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

  raddr.sin_family = AF_INET;
  raddr.sin_port = htons(80);
  /* www.abo.fi = 130.232.213.59 */
  inet_pton(AF_INET, "130.232.213.59", &raddr.sin_addr);

  if (-1 == connect(sockfd, (struct sockaddr *)&raddr, sizeof (struct
   sockaddr_\
in))) {
    perror();
    exit(-1);
  }

  write(sockfd, msg, strlen(msg)+1);
  count = read(sockfd, retbuf, 4096);
  retbuf[count] = 0;
  printf(retbuf);
}
```

Nätverk – liten server

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

char msg[] = "GET / HTTP/1.0\n\n";
char retbuf[4096];

int main() {
  int sockfd, c_sock;

  struct sockaddr_in saddr;
  struct sockaddr_in caddr;
  int count;

  sockfd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

  saddr.sin_family = AF_INET;
  saddr.sin_port = htons(9876);
  saddr.sin_addr.s_addr = inet_addr("127.0.0.1");
  bind(sockfd, (struct sockaddr *)&saddr, sizeof(saddr));

  listen(sockfd, 5);
  while (1) {
    char ch;
    socklen_t clen = sizeof(caddr);
    printf("Server waiting\n");

    c_sock = accept(sockfd, (struct sockaddr *)&caddr,
&clen);
    printf("Connection from '%s'\n",
inet_ntoa(&caddr.sin_addr.s_addr));

    read(c_sock, &ch, 1);
    ch++;
    write(c_sock, &ch, 1);
    close(c_sock);
  }
}
```

Preprocessor

- Kompilering av C kod föregås av en preprocessor
- Preprocessorn genererar en ny ”fil” med källkod genom att
  - ersätta definierade textsträngar
  - inkludera andra filer
  - konditional inkludering av textavsnitt
- Denna nya enhetliga fil körs genom den egentliga C-kompilatorn

Preprocessor – exempel

```
/* test.c */

#define MAX(a,b) ( ((a)>(b))? (a) : (b) )

#ifdef  DEBUG
#define DPRINT(x) printf("Debug: '%s\n",
   x);
#else
#define DPRINT(x)
#endif

main () {
  int x = 33;
  int y = 12;

  printf("Max av 2 och 5 ar %i\n",
   MAX(2,5));
  printf("Max av x och y ar %i\n",
   MAX(x,y));

  DPRINT("Nu narmar sig slutet");
}
```

```
gcc –E test.c -DDEBUG
```

```
main () {
  int x = 33;
  int y = 12;

  printf("Max av 2 och 5 ar %i\n", (
   ((2)>(5))? (2) : (5) ));
  printf("Max av x och y ar %i\n", (
   ((x)>(y))? (x) : (y) ));

  printf("Debug: '%s\n", "Nu narmar sig
   slutet");;
}
```

```
gcc –E test.c
```

```
main () {
  int x = 33;
  int y = 12;

  printf("Max av 2 och 5 ar %i\n", (
   ((2)>(5))? (2) : (5) ));
  printf("Max av x och y ar %i\n", (
   ((x)>(y))? (x) : (y) ));

}
```

C Preprocessorn

- **`#define`** – Definiera / substitionera
  - **`#define MAX_ARRAYSIZE 100`**
  - **`#define DOUBLE(X) ((x)*2)`**
- **`#include <fil.h>`** – Inkludera fil
  - Sök reda på fil och sätt filens innehåll på motsvarande plats
- **`#if #elif #else #endif`**
  - Inkludera textavsnitt om ett uttryck gäller – dynamisk kod

C Preprocessorn

- Effektiv för
  - debuggning
  - ”konstanter” som är lätta att vid omkompilering ändra på
  - konditionella kodblock (typisk vid multiplatform / -arkitektur)
  - konfigurering av system

```
#ifdef USE_WLAN
    kod för WLAN-modul
#endif
```

- OBS! Ingen run-time overhead