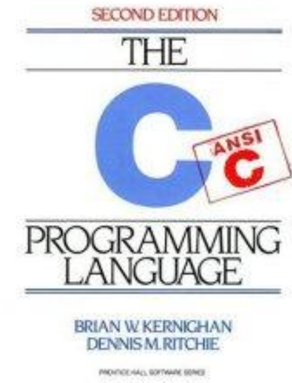
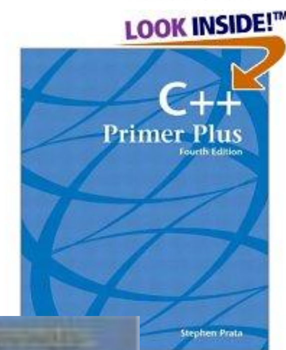
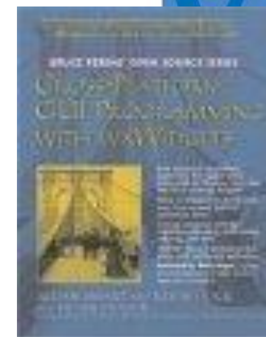
Programmering i C/C++

Böcker och resurser

- C:
  - Kernighan&Ritchie: The C programming language
- C++:
  - Frank Brokken: C++ Annotations
    - **`http://www.icce.rug.nl/documents/cplusplus/`**
  - Stephen Prata: C++ Primer Plus
    - **`www.cplusplus.com`**
- Verktyg för GUI utveckling
  - Qt (Cross-Platform GUI toolkit)
    - **`www.wxwidgets.org`**
    - Multipla platformar
    - Multipla språk





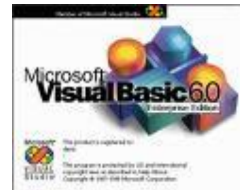


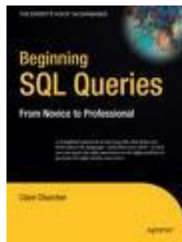
Grunduppgifter 2012

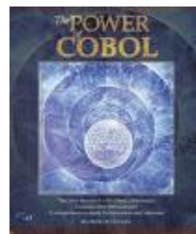
- Föreläsare: Jerker Björkqvist, rum B4052
- Föreläsningsslides, RÖ, tidtabell

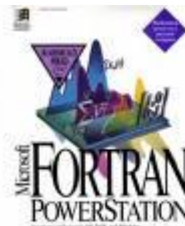
  `https://xprog28.cs.abo.fi/ro.nsf/W/cprog`
- Föreläsningar,
  - Måndagar 10-12, A3058
  - Onsdagar 15-17, A3058 (i praktiken -> 16.30)
- Kurstenter: 26.10, 9.11
- Övningsuppgifter / labbar:
  - 1. Elektronisk på Ville, mer ifnromation senare
  - 2. Inlämande övningar 1-3
  - Onsdagar 10-12, Linux-klassen
- Tenrätt då 80% av övningar i Ville slutföra
  - Resultat till STURE då tent+övningar gjorda

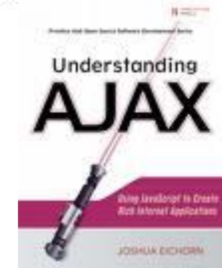
Varför lära sig C/C++ ?





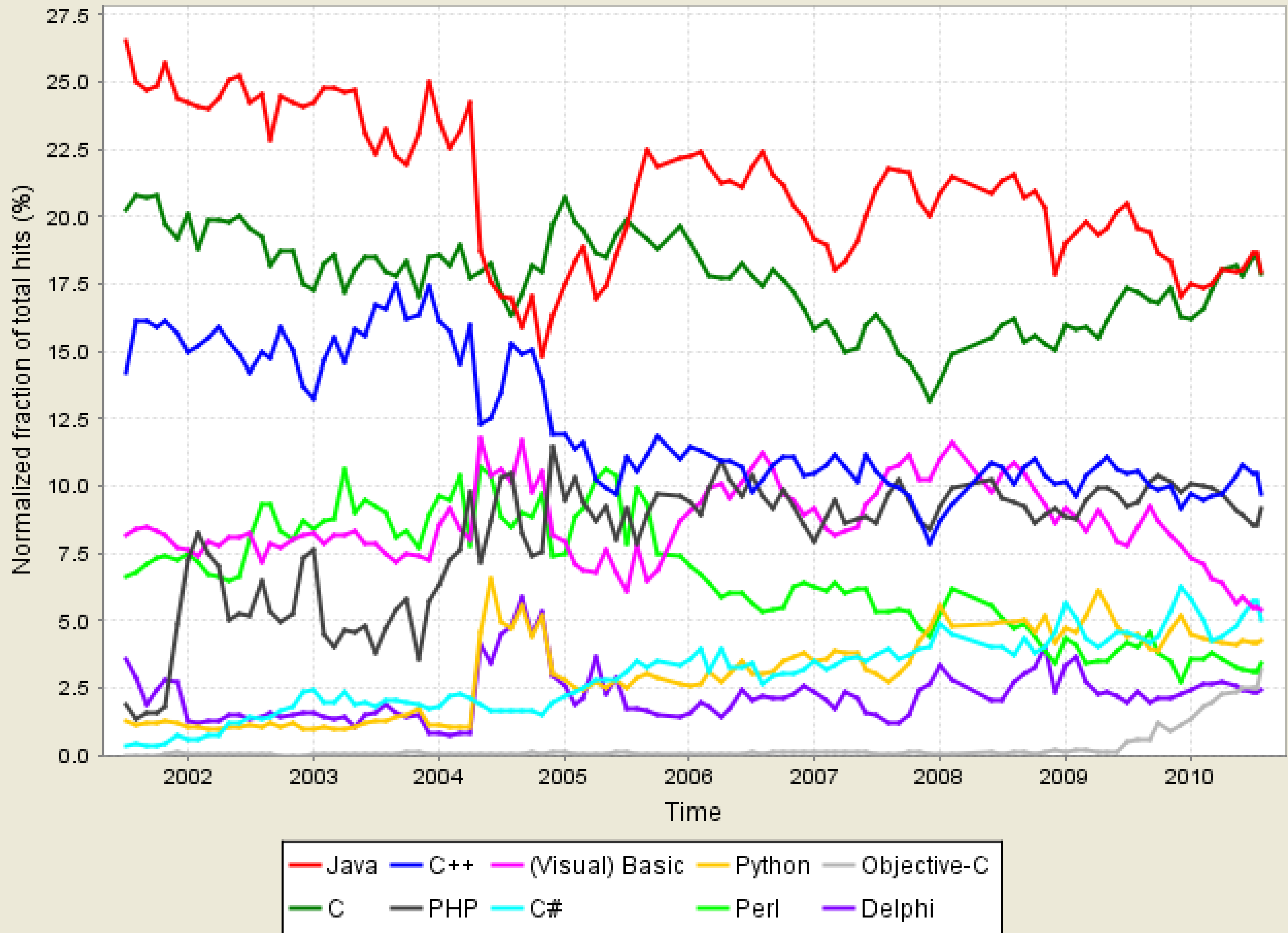






Position Aug 2010	Position Aug 2009	Delta in Position	Programming Language	Ratings Jul 2010	Delta Jul 2009	Status
1	1	=	Java	17.994%	-1.53%	A
2	2	=	C	17.866%	+0.65%	A
3	3	=	C++	9.658%	-0.84%	A
4	4	=	PHP	9.180%	-0.21%	A
5	5	=	(Visual) Basic	5.413%	-3.07%	A
6	7	↑	C#	4.986%	+0.54%	A
7	6	↓	Python	4.223%	-0.27%	A
8	8	=	Perl	3.427%	-0.60%	A
9	19	↑↑↑↑↑↑↑↑↑↑	Objective-C	3.150%	+2.54%	A
10	11	↑	Delphi	2.428%	+0.09%	A
11	9	↓↓	JavaScript	2.401%	-0.41%	A
12	10	↓↓	Ruby	1.979%	-0.51%	A
13	12	↓	PL/SQL	0.757%	-0.23%	A
14	13	↓	SAS	0.715%	-0.10%	A
15	20	↑↑↑↑↑	MATLAB	0.627%	+0.07%	B
16	18	↑↑	Lisp/Scheme/Clojure	0.626%	0.00%	B
17	16	↓	Pascal	0.622%	-0.05%	B
18	15	↓↓↓	ABAP	0.616%	-0.12%	B
19	14	↓↓↓↓↓	RPG (OS/400)	0.606%	-0.15%	B
20	-	↑↑↑↑↑↑↑↑↑↑	Go	0.603%	0.00%	B

Källa: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html



Källa: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

C - användningsområden

- Systemprogrammering
  - Operativsystem:
    - Kernel
    - Drivrutiner
  - Databasmotorer
  - Middleware i olika utsträckning
- Inbyggda datorsystem
  - DSP
  - Andra små datorsystem
- I allmänhet där
  - Prestandan är viktigare än produktiviteten
  - Man vill ha god kontroll över vad man gör

Exempel på C-implementationer

- Operativsystemens kärnor:
  - Linux, Windows 7, MacOS, RTOS (uC/OS-II),RTEMS
- Tolkar för skriptade språk
  - Python, Perl, PHP, Ruby
- Inbyggda system
  - ECU:s (Elektroniska kontrollsystem)
  - DSP (Digital Signal Processors)
  - Microcontrollers
  - ....

C - Historia

- Multics var ett samarbetsprojekt mellan MIT, General Electric, Bell laboratories.
  - Grundideer för dagens operativsystem: Processer, trädstrukturerat filsystem, generell access till hårdvaruenheter
  - Blev för dyrt och försenat
- Typlöst språk BCPL mitten på 1960-talet
- Thompson utvecklade ett språk för systemprogrammering utgående från BCPL: B
- Dennis Ritchie utvecklade språket B och kallade det C, tidigt 1970-tal
- 1978: Brian Kernighan och Ritchie skrev en bok: K&R C: De facto standard
- 1983: ANSI började standardisera språket, 1989 skapades ANSI C89
- 1999: Ny version av standarden för C, C99

http://www.ixonos.com/easydata/customers/ixonos/files/OuluLinuxPresentations/Linux_seminar_stroustrup.pdf

Fördelar med C

- Enkelt att skriva program som direkt accesserar minne och annan hårdvara.
- Stora delar av Unix, Linux, Windows och MacOS är skrivet i C = lätt att anropa systemet från C-program.
- Finns otroligt många bibliotek till C.
- Finns bra kompilatorer som genererar effektiv kod.
- Finns till nästan alla plattformar
- Kommer antagligen att finnas kvar i många år framöver.
- Många andra språk är baserade på C (eller närmast C syntax): C++, Java, C#, Javascript, PHP

Nackdelar med C

- Alldeles för enkelt att göra katastrofala programmeringsfel.
- Kan vara svårt att debugga C-program.
- Inget modulsystem
- Är för lågnivå för att man ska kunna skriva stora program och samtidigt ha god överblick.
- Lockar till att göra ”hacks”

HelloWorld i C

```
/* helloword.c */
#include <stdio.h>

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

Kompilering och körning
% gcc helloworld.c -o helloworld
% ./helloworld
```

Kompilatorn *gcc* finns ofta installerad på Linux-distrubtioner.
Vid ÅA kan alla använda datorn tuxedo.abo.fi (Linux).

Exempel på systemanrop

- Då helloworld.c körs:

```
main() {
        printf(”Hello World\n”);
}
```

  - 25 systemanrop (kernel-funktioner)
- Då HelloWorld.java körs

```
public class HelloWorld {
    public static void main(String args[])
{
        System.out.println("Hello
World!");
    }
}
```

  - 2248 systemanrop !!

Exampel: Lista systemanrop (2)

```
$> strace ./helloWorld

execve("./helloWorld", ["./a.out"], [/* 24 vars */]) = 0
uname({sys="Linux", node="borken", ...}) = 0
brk(0)                                  = 0x804962c
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x40016000
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=82445, ...}) = 0
old_mmap(NULL, 82445, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40017000
close(3)                                = 0
open("/lib/libc.so.6", O_RDONLY)        = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0 \304\1"..., 1024) = 1024
fstat64(3, {st_mode=S_IFREG|0755, st_size=5735106, ...}) = 0
old_mmap(NULL, 1267176, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x4002c000
mprotect(0x40158000, 38376, PROT_NONE)  = 0
old_mmap(0x40158000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3, 0x12b000) =
0x40158000
old_mmap(0x4015e000, 13800, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,
-1, 0) = 0x4015e000
close(3)                                = 0
munmap(0x40017000, 82445)               = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 3), ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x40017000
write(1, "Hello world\n", 12Hello world
)               = 12
munmap(0x40017000, 4096)                = 0
_exit(0)                                = ?
```

C grunder

- Liknar till sin struktur Pascal eller Fortran (proceduralt språk)
- Värden sparas i variabler
- Program struktureras genom att definiera och kalla på funktioner
- Programflöden kontrolleras via loopar, if-statements och funktionsanrop
- Användning av **pekare** (eller indirekt adressering) är centralt

```
/* test.c */
#include <stdio.h>
void test(int a);
int main(void) {
int a=23;
int *b = &a;
test(a);
}
void test(int a) {
if (a>54) {
printf(“parametern till test()
är större än 54\n”);
}
}
```

C – språket - filer

- Källfiler (.c)
  - Innehåller implementationen av funktioner
  - Program indelas gärna i många separata källfiler
- Header-filer (.h)
  - Innehåller prototyper för funktioner, definitioner av datatyper
  - Definitionerna används (inkluderas i källkoden) genomgenom #include-direktivet i källkoden

Prototyp: Deklaration av funktioner, inklusive antal parametrar och deras typer

```
int myfunc(float a, int b);
```

Ett enkelt C-program

- Direktiv till preprocessorn :`#include <stdio.h>`
- Variabeldeklarationer: `int a;`
- Funktionsprototyper: `int main(void);`
- Funktionsdefinitioner

```
int main(void) {
....
}
```

- C är case-sensitive
- Variabler måste deklareras innan de kan användas
- Funktioner kan anropas utan att det finns en prototyp (men rekommenderas ej)
- Kommentarer med `/* ... */` (ANSI C99 även `//`)

Datatyper - heltal

- `char` – minst 1 byte
- `short` – minst 2 byte
- `int` – minst 2 byte, idag oftast 4 byte
- `long` – minst 4 byte
- `long long` – minst 8 byte
- Gränser finns limits.h
- Konstanter
  - `2, 6L, 88LL, 0x20, 0775`

I C-kod kan man specificera vilken datatyp man vill att kompilatorn skall tolka ASCII-texten som då den går igenom (parsar) källkoden
L – long, LL - long long, 0x – hexadecimalt, ...

Datatyper - flyttal

- `float` – ofta 4 byte → 10^{37}, 6 siffrors noggrannhet
- `double` – minst som float
- `long double` – minst som double
- Konstanter
  - `3.14, 0.5e-7, 12.4f, 145.67L`
- Gränser finns i `float.h`

Datatyper - boolska

- I C finns inga direkta boolska värden (njaa ANSI C99 definierar...)
- Heltal fungerar som boolska värden → 0 betraktas som false, alla andra heltal betraktas som true

```
a=99;
if (a) do_something();
```

Räckor

- En dimensionella - räckor

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int lottorad[7]; /* 7 siffror per rad */
    for (i=0; i<7; i++)
      lottorad[i] = rand() % 37;
    return 0;
}
```

- Flerdimensionella - räckor

```
#include <stdio.h>
int lottblankett[10][7];
lottoblankett[2][5] = rand() % 37;
```

Räckor

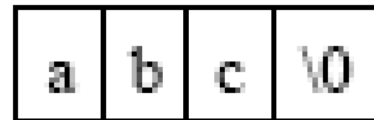
- Måste intialiseras före användning

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   int lottorad[7];
   printf("%d\n", lottorad[5]); /* odefinerat */
   printf("%d\n", lottorad[33]); /* typisk buffer
   overrun, resultat varierande: Odefinierat /
   segmentation fault / general protection fault
   */
}
MEN OBS: Kompilatorn har inget att klaga på här
```

Strängar

- Det finns ingen speciell sträng-typ i C. C är en räcka tecken som avslutas med ett null-tecken

```
char a[] = "abc";
a[0] = ’a’
a[1] = ’b’
a[2] = ’c’
a[3] = ’\0’
```



- I filen `string.h` finns funktionsprototyper för många användbara sträng-funktioner

Exempel

```
#include <stdio.h>

int main(void)
{
   int nstudents = 0; /* Initialisering */

   printf("How many students does ÅA have ?:");
   scanf ("%d", &nstudents);  /* Läs input */
   printf("ÅA has %d students.\n", nstudents);

   return 0;
}
$How many students does ÅA have ?: 5000 (enter)
ÅA has 5000 students.
$
```

Typkonverteringar

```
#include <stdio.h>
int main(void)
{
   int i,j = 12;           /* endast j initialiseras*/
   float f1,f2 = 1.2;

   i = (int) f2;           /* explicit: i <- 1, 0.2 lost */
   f1 = i;                 /* implicit: f1 <- 1.0 */

   f1 = f2 + (float) j; /* explicit: f1 <- 1.2 + 12.0 */
   f1 = f2 + j;            /* implicit: f1 <- 1.2 + 12.0 */
}
```

Typkonvertering

- Överraskningar / feltänkt??

```
   int a, b=1;
   a = 2 * (b / 2);  /* a har nu värdet 0 */

MEN:
   a = 2 * (b / 2.0); /* a har nu värdet 1 */

----------------
   int sum=7, count=3;
   float average;

   average = sum/count; /* ger medeltalet 2.0 */
MEN:
   average = (float)sum/count; /* ger medeltalet 2.3333 */
```

Operatorer

- Arithmetiska
  - `int i = i+1; i++; i--; i *= 2;`
  - `+, -, *, /, %,`
- Relationer and logik
  - `<, >, <=, >=, ==, !=`
  - `&&, ||, &, |, !`
- Observera:

`= tilldelning`

`== jämförelse`

Konstruktioner

- **`if (expr) statement`**
  - Om värdet på **`expr`** är olika 0 exekveras **`statement`**
- **`if (expr) statement else statement2`**
  - Om värdet på **`expr`** är lika med 0 exekveras **`statement2`**

`statement` är antingen enkel: **`a=2;`**, eller sammansatt: **`{a=2; b=1;}`**

`if` - exempel

```
/*
if.c
*/
#include <stdio.h>
int main(void)
{
   int a = 2;
   int b = 3;
   if(a > b) printf("a är större än b!\n");
   else printf("a är inte större än b!\n");
   return 0;
}
```

while

- **`while (expr) statement`**
  - Så länge värdet på expr är olika 0 exekveras statement
- **`do statement while (expr)`**
  - Annan version av while

for

- **`for (pre_expr; loop_expr; post_expr) statement`**
- Loop-konstruktion
  - Först utförs **`pre_expr`**
  - Så länge **`loop_expr`** är olika 0, utförs först **`statement`** och sedan **`post_expr`**
- Mycket effektiv konstruktion

for - exempel

```
int a,i,n=5;
a=0;
for (i=0; i<n; i++) a += i*i;
```

Ekvivalent med

```
int a,i,n=5;
for (a=0,i=0; i<n; a += i*i++) ;
```

Funktioner

- Program delas in i mindre block
  - Modularitet – enklare att
    - debugga, koda
  - Återanvändning av kod

- Funktionsdeklaration – prototyp
- Funktionsdefinition – prototyp + kod = implementation

Funktionsdefinitioner

```
type name(parameter_list){
    deklarationer satser
}
```

- Om en funktion inte returnerar något ges **`void`** som returtyp
- Parameterlistan består av typ + variabelnamn **`(int a, float b)`**
- Vid definitionen kallas **`a`** och **`b`** för de formella parametrarna
- Vid anropet **`f(x,4)`** kallas **`x`** och **`4`** för de aktuella parametrarna, eller argument

Funktionsanrop

- Före funktioner anropas rekommenderas att funktionen är deklarerad (dvs kompilatorn känner till funktionens prototyp)
  - returtypen är korrekt
  - parametertyperna kan valideras
  - detta görs vanligen genom att includera en header-fil

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

- Funktionsanrop = funktionens namn + parametrarna i parentes: **`pow(2,4)`**
- Funktionsanrop sker alltid call-by-value !!

Funktioner - exempel

```
#include <stdio.h>
int sum(int a, int b);
      /* function prototype at start of file */

void main(void){
   int total = sum(4,5); /* call to the function
  */

printf(“The sum of 4 and 5 is %d”, total);
}

int sum(int a, int b){   /* the function itself
                  - arguments passed by value*/
  return (a+b);       /* return by value */
}
```

Variabler – synlighet

- I C är grunden för variablers synlighet per fil - kompileringsblock
  - En variabel som är deklarerad utanför en funktion är *global* inom nuvarande kompileringsblock
  - En variabel som deklareras inom en funktion är lokal inom funktionen
  - En variabel som deklareras inom en konstruktion (**`for, while, if`**) är lokal inom konstruktionen

Synlighet - exempel

```
#include <stdio.h>

int gCommonPar; /* Global variabel */

void main(void){
   int i=0; /* Lokal inom funktionen */
   for (i=0; i<5;i++) {
       int j; /* Lokal inom kompileringsblocket */
       j=i*5+gCommonPar;
       printf("Beräkningen gav: %i\n", j);
   }
   j=3; /* Ger kompileringsfel, variabeln j finns
   ej inom detta block */
}
```

Variabler - livslängd

- **`static, auto`** definierar variablers livslängd
  - **`static`** – variabeln placeras i heapen , får en egen plats i minnet som existerar under programmets hela exekvering
  - **`auto`** – variabeln placeras på stacken – minnesutrymmen finns bara så länge exekveringen är inom konstruktionen
  - **`auto`** är default för lokala variabler – **`static`** är default för globala variabler

Livslängd exempel

```
#include <stdio.h>

int gimmenext() {
  static int next=0; /* Heap */
  return ++next;
}
int gimmenext2() {
  int next=0; /* Auto(stack) */
  return ++next;
}
int main() {
   printf("%i\n",gimmenext());
  printf("%i\n",gimmenext());
   printf("%i\n",gimmenext2());
  printf("%i\n",gimmenext2());

}
```

```
% ./gimmenext
1
2

1
1
```

Andra lagringsklasser

- **`register`** – direktiv för kompilatorn att (om möjligt) placera variabeln i ett processorregister
- **`volatile`** – direktiv för kompilatorn att inte utföra optimeringar kring denna variabel
  - t.ex. variabler som hänvisar till hårdvaruregister
- **`extern`** – variabeln finns definierad som global i ett annat kompileringsblock, global variabel delad mellan kompileringsblock
- **`const`** – variabel definieras som konstant, d.v.s. kompilatorn skall ej godkänna modifieringar av variabelvärde

Exempel - **extern**

```
/* fil_a.c */

int g_CommonParameter = 5;
```

```
/* fil_b.c */
#include <stdio.h>
extern int g_CommonParameter;
int main() {
  printf("Parameter=%i\n", g_CommonParameter);
}
```

```
% gcc fil_a.c fil_b.c -o prog
% ./prog
Parameter=5
```

Strukturer

- C implementerar strukturer

```
#include <string.h>
#define MAX_STRING 30
struct t_person {
   char namn[MAX_STRING];
   int alder;
   int skonummer;
};
int main() {
   struct t_person Person; /* Strukturen på
   stacken */
   Person.alder = 23;  /* Access av strukturmedle
                                via "."*/
   strcpy(Person.namn, "kalle");
}
```

Nästade strukturer

```
#define MAX_DELTAGARE 100
#define MAX_STRING 30
struct t_kurs {
  char namn[MAX_STRING];
  int nDeltagare;
  struct t_person Deltagare[MAX_DELTAGARE];
};

main() {
  struct t_kurs Kurs;
  strcpy(Kurs.namn, "Programmering i C/C++");
  Kurs.nDeltagare = 1;
  Kurs.Deltagare[0].alder = 77;
  strcpy(Kurs.Deltagare[0].namn, "Axel Eklund");
}
```

Synonymer för datatyper

```
typedef unsigned short WORD;

WORD wIndex; /* samma som unsgined short wIndex */

typedef struct t_person Person;

Person Jag;    /* som struct t_person Jag; */

typedef struct t_point {
  WORD x;
  WORD y;
} Point;

Point point;
point.x = 12; point.y=44;
```

Minneslayout och adresser

En variabel har alltid en representation i minnet.
Datatyp med visst värde
→ given bitsekvens i minnet

```
int x = 5, y = -10;
float f = 12.5, g = 9.8;
char c = 'c', d = 'd';
```

Variabels värde (med given datatyp)

5	-10	12.5	9. 8	c	d
0x00000005	0xFFFFFFF6	0x41480000	0x411CCCCD	0x63	0x64

4300 4304 4308 4312 4316 4317

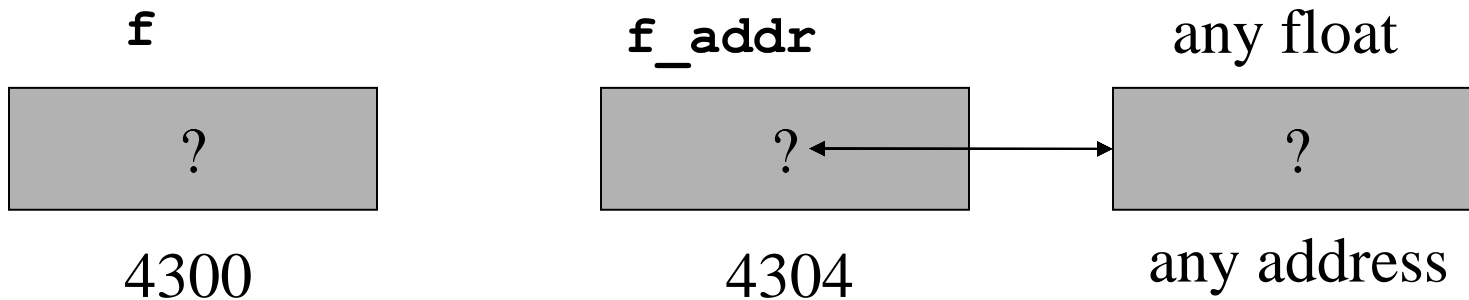
Binär sekvens i minnet

Adress i minnet

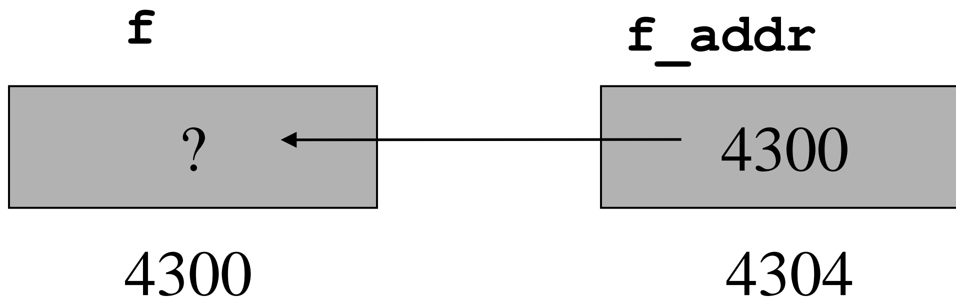
Pekare

- *Pekare* = variabel som innehåller adress till annan variabel

```
float f;          /* datavariabel */
float *f_addr;    /* pekarvariabel */
```




```
f_addr = &f; /* & = adressoperator */
```

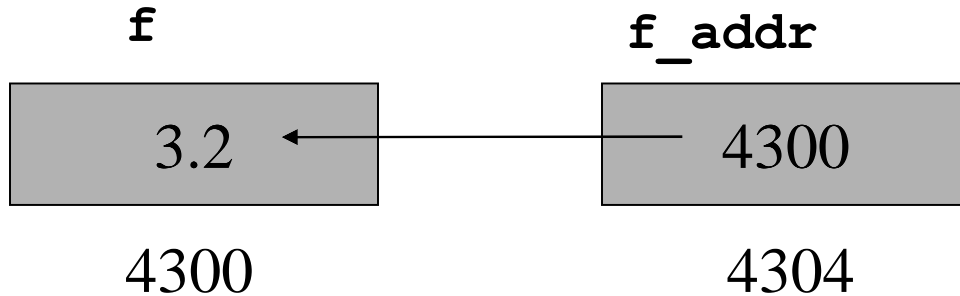


Exempel på pekare

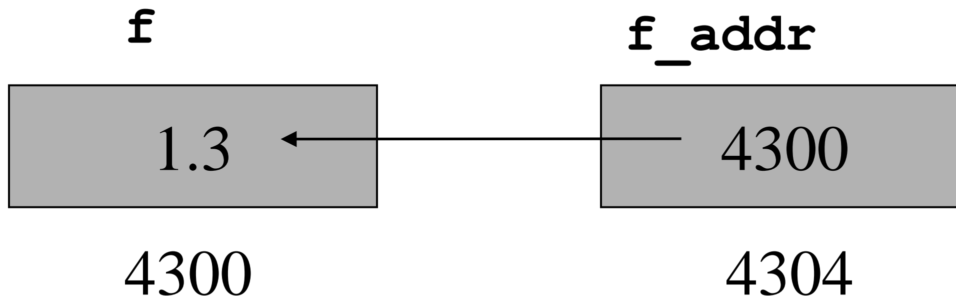
```
#include <stdio.h>

void main(void) {
int j;        /* j = <odefinierad> */
int *ptr;     /* ptr = <odefinierad> */

ptr=&j;     /* initialisera före användning */
            /* *ptr=4 initialiserar ej ptr */

*ptr=4;       /* j <- 4 */

j=*ptr;       /* j <- ??? */
}
```

Pekare 2

```
*f_addr = 3.2;    /* indirekt adressering */
```

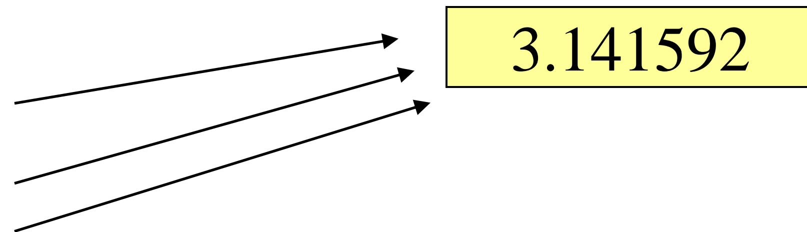



```
float g=*f_addr; /* indirektion:g är nu 3.2 */

f = 1.3;
```



Pekare – byte mellan datatyper

```
#include <stdio.h>

int main(void) {
float f = 3.141592;
void *ptr;  /* allmän pekare, ej till typ */
float *f_ptr;
int *i_ptr;

ptr = (void *)&f;
f_ptr = &f;
i_ptr = (int *)&f

printf("Val: %f, adress: %p, : int %i\n",
      *f_ptr, ptr, *i_ptr);
}
% Val: 3.141592, adress: 0xbf82e238, int: 1065353216
```



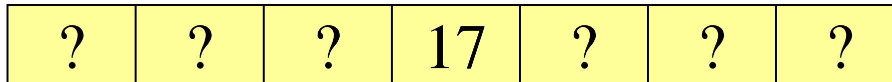
sizeof()-operatorn

- **`sizeof(x)`** – ger storleken för datatypen x
- t.ex.
- **`sizeof(int) = 4`**
- **`sizeof(float) = 4`**
- **`sizeof(int *) = 4`**
- **`sizeof(Person) = 40`**

  `char[30] + int + int + alignment`

Vad är en räcka ?

```
int main(void) {
   int lottorad[7];

   lottorad[3] = 17;
}
```




```
variabeln lottorad är de fakto en pekare till det
första elementet i räckan
lottorad[3] kan räknas som
   lottorad + 3 * sizeof(int)
```

Mer om adressering av räckor

Kompilatorn gör följande beräkningar

```
&lottorad[3] = lottorad + sizeof(integer)*3
```

Man kan också accessera räckan såhär:

```
*(lottorad+3) = 17 /* ekvivalent med lottorad[3]=17 */
```

Dock:

```
*lottorad+3 = <int>
```

Vi kan också accessera räckan via generell pekare

```
int *lotto_ptr = lottorad;
lotto_ptr[3] = 17;
```

Mer om räckor

Vad är då skillnaden mellan

```
int lottorad[7];
  OCH
int *lotto_ptr;
```

lottorad : Konstant pekare till en räcka, samtidigt som utrymme reserveras)

lotto_ptr: Är en variabel pekare (vi kan ändra värdet på själva pekaren)

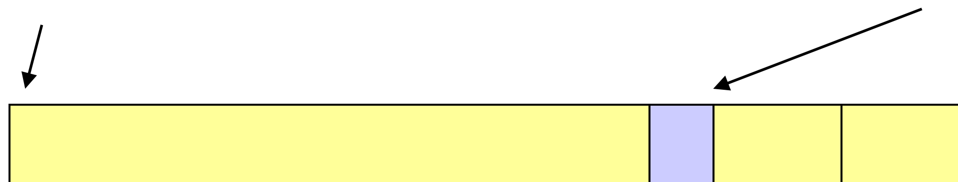
Mera pekare

```
int month[12]; /* month is a pointer to base address 430*/
int *ptr;

month[3] = 7;  /* month address + 3 * int elements
        => int at address (430+3*4) is now 7 */

ptr = month + 2; /* ptr points to month[2],
        => ptr is now (430+2 * int elements)= 438 */
ptr[5] = 12;
         /* ptr address + 5 int elements
              => int at address (438+5*4) is now 12.
            Thus, month[7] is now 12 */

ptr++;         /* ptr <- 438 + 1 * size of int = 442 */
(ptr + 4)[2] = 12;   /* accessing ptr[6] i.e., month[9] */
```

- Nu är `month[6]`, `*(month+6)`, `(month+4)[2]`, `ptr[3]`, `*(ptr+3)` samma heltalsvariabel !!.

Pekare till strukturer

```
typedef struct t_person {
   char namn[30];
   int alder;
   int skonummer;
} Person;

typedef Person * pPerson;

int main() {
   Person pers;
   pPerson ptr_pers = &pers;
}
```



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;
}
```

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... */
```