C = C++

C för den som kan C++

Definition: Variabel

- Lagringsutrymme i datorns minne som har
  - Ett namn (avgör adress i minnet)
  - En datatyp (avgör storlek och tolkning)
  - Ett värde (kan inte vara "tom")
- När namnet anges i programtext "går" exekveringen till variabelns utrymme i minnet och är redo att antingen läsa eller skriva där.
- Operationen & framför namnet ger adressen

Definition: Pekare

- Speciellt namn på en variabel som lagrar en adress. Datatypen är alltså alltid "en adress".
- Operationen * framför en pekare anger att exekveringen skall läsa variabelns värde och "gå till" den adressen och vara redo att läsa eller skriva där.

Typkonvertering

- Sträva efter kod som är enkel och tydlig!
- Om du behöver explicit typkonvertering så har du tagit ett dåligt beslut någonstans.
- Behöver du många explicita typkonverteringar, skriv en funktion som gör konverteringen på ett ställe, speciellt om du konverterar mellan olika pekartyper.

```
struct type* to_p = (struct type*)from_v;
double to_d = (double)from_i;
unsigned int to_d = (unsigned int)from_i;
```

"Objektorientering" i C

- En headerfil och implementationsfil per klass
- En struct deklararas med enbart namn i headerfilen
- Samma structen definieras med alla datamedlemmar i implementationsfilen
- Alla medlemsfunktioner tar en pekare till structen som första parameter och namnges med structens datatyp som prefix
- Publika medlemsfunktioner deklareras i headerfilen och implementeras i implementationsfilen
- Privata medlemsfunktioner deklareras static i implementationsfilen (inte i headerfilen) och implementeras i implementationsfilen

OO i C: Exempel: date.h

```
/* Note that we do not know anything about a date,
 * all details (member variables) are hidden (private).
 * Member variables can not be intentonally or by mistake,
 * since the compiler simly do not know about them. */
struct date;

/* The functions that others may use. */
void date_init(struct date* d);
int date_get_year(struct date* d);
int date_set_year(struct date* d);
```

OO i C: Exempel: date.c

```
#include "date.h"

struct date {
  int y, m, d; /* private details */
};

/* Public functions */
void date_init(struct date* d) { ... }
int date_get_year(struct date* d) { ... }
int date_set_year(struct date* d) { ... }

/* Private functions */
static int date_to_int(struct date* d) {
  return (*d).y*100*100 + (*d).m*100 + (*d).d;
}
```

Piloperatorn ->

Studera koden i exemplet igen, speciellt `(*d).y`:

```
static int date_to_int(struct date* d) {
  return (*d).y*100*100 + (*d).m*100 + (*d).d;
}
```

Uttrycket `(*d).y` skall tolkas "gå till adressen angiven i d, hämta från det minnesutrymmet ut medlemsvariabeln y".

Piloperatorn tillåter att samma sak skrivs på ett enklare, smidigare sätt:

```
static int date_to_int(struct date* d) {
  return d->y*100*100 + d->m*100 + d->d;
}
```

static

Nyckelordet static har mer än en användning och betydelse. Enkelt kan man säga att något som deklareras static kommer finnas i endast en instans, initierat endast en gång (som en global variabel), och samtidigt blir det endast åtkomligt lokalt i sitt deklarationsblock eller i sin fil om det nyttjas på global nivå. Globalt men privat.

extern

Normalt åstadkommer en variabeldeklarartion även att variabeln definieras (dvs skapas i minnet). Ibland vill man dock skapa en global variabel som skall vara åtkomlig även från andra filer.

Att då deklarera variabeln i headerfilen skulle innebära att den definieras på nytt varje gång headerfilen inkluderas, vilket inte är önskvärt.

Nyckelordet `extern` framför en deklaration gör att variabeln endast deklareras. Detta skall användas i headerfiler då en global variabel skall bli åtkomlig utifrån. Variabeln måste fortfarande definieras någonstans, lämpligen i motsvarande implementationsfil.

I korthet betyder `extern`: följande variabel finns redan på annan plats, använd den gärna men skapa den inte igen.

En vanlig variabel deklaration/definition säger: *skapa* följande variabel för senare användning, tack.

Dynamisk minnesallokering

- void* malloc(int size)
  - reserverar (lånar) angivet antal byte sekventiellt minne och returnerar adressen till den första av dessa
- free(void* adr)
  - ta en adress som tidigare returnerats från malloc och återlämar de byte som reserverades av malloc

new vs. malloc

C++	C
int* ip = new int;	int* ip = (int*)malloc(sizeof(int));
int* el = new int[SIZE];	int* el = (int*)malloc(SIZE*sizeof(int));
date* di = new date;	struct date* di = (struct date*)malloc(sizeof(struct date));
	#define NEW(type, size) (type*)malloc((size)*sizeof(type));
int* ip = new int;	int* ip = NEW(int, 1);
int* el = new int[SIZE];	int* el = NEW(int, SIZE);
date* di = new date;	struct date* di = NEW(struct date, 1);

delete vs. free

C++	C
delete ip;	free(ip);
delete[] el; // [] entire array	free(el);
delete di;	free(di);

`free` skall alltid användas exakt en gång på varje *adress* som `malloc` returnerat
`delete` skall alltid användas exakt en gång på varje *adress* som `new` returnerat

```
char* cp = (char*)malloc(sizeof(char));
void* koko = cp;  // adress from malloc copied...
char* nut = NULL;

cp = NULL;      // one copy of adress from malloc lost
nut = koko;     // adress copied again
// exit(0);     // would be ERROR, adress from malloc not freed
free(nut);      // OK, adress from malloc freed once
// free(koko); // would be ERROR, adress from malloc freed twice
```

NULL

En literal konstant för att ange adressen noll (0) i datorns minne. Denna adress är alltid ogiltig att avreferera. Den används för att markera att en pekare är ”tom”, dvs inte innhåller någon giltig adress.

```
#define NULL ((void*)0)
```

Använda pekare: FEL och RÄTT

```
void date_init(struct date* d); // initierar ett datum

struct date* di; // FARLIGT! OINITIERAD!
date_init(di); // FEL!

struct date* di = malloc(sizeof(struct date*)); // FEL!
date_init(di); // FÖLJDFEL!

struct date* di = malloc(sizeof(struct date)); // RÄTT!
date_init(di); // RÄTT!

struct date di; // BÄST! Använd stacken om det går.
date_init(&di); // RÄTT!
```

Vad säger kompilatorn om de fyra varianterna ovan?

size_t sizeof(datatype)

`sizeof()` är ett funktionslikt nyckelord som returnerar storleken, *i byte*, på angiven datatyp. Denna ”funktion” utvärderas direkt vid kompilering och kräver att kompilatorn har full kännedom om datatypen i fråga.

Med förut givna exempel (`struct date`) så är det inte möjligt att anropa `sizeof(struct date)` för den som bara inkluderat hederfilen. Detta beror på att kompilatorn inte kan göra beräkningen utan att känna till varje medlemsvariabel och dess storlek. Dock är det önskvärt att använda `sizeof` i samband med dynamisk minnesallokering. Det går faktiskt enbart att skapa pekare till date så länge inte alla detaljer är kända.

För att lösa detta kan man införa en publik funktion som returnerar storleken:

```
int date_sizeof();
```

Funktionen implementeras i `date.c` där alla detaljer är kända.

```
int date_sizeof() { return sizeof(struct date); }
```

Sizeof är systemberoende

- Storleken på en datatyp är systemberoende. Standarden definerar bara nedre gränser.
- Frågar du t.ex. om storleken på en pekarvariabel så kommer den att vara 4 byte för ett 32-bitars program och 8 byte för ett 64-bitars program. båda programmen kan exekveras på ett 64-bitars system (prova kompilatorflaggan –m32).
- Pintos är ett 32-bitars system.
- Vi kompilerar Pintos på 64-bitars Linux.
- Dessa slides utgår från 64-bitars pekare i exempel. Sitter du på ett 32-bitars system kommer alltså vissa exempel inte stämma.

typedef

`typedef` är ett nyckelord som tillåter att en `datatyp` ges ett bekvämare namn. Syntax ser ut som en vanlig variabeldeklaration, fast med nyckelordet `typedef` framför. Semantiskt är det dock *inte* fråga om att skapa någon variabel.

```
/* Create an alias for a struct date pointer. */
typedef struct date* date_p;

/* Use the alias. Advanages include that it is less to
 * write and much more flexible when you later on need
 * to change all date_p to mean struct gregorian_date*.
 * The code can also be made more readable.
 */
void date_init(date_p d);
```

Fält (array)

I C finns (mycket lite) språkstöd för att hantera kontinuerliga sekvenser med data. Exempel:

```
#define SIZE 5
/* SIZE must be a literal constant,
 * it is not allowed to use any kind
 * of variable, not even a constant.
 */
int array[SIZE] = { 0, 1, 2, 3, 4};
int sum = 0;
for (i = 0; i < SIZE; ++i) {
  sum += array[i];
}
```

Fält är Pekare

Ett fält är i själva verket mycket lite utöver en vanlig pekare. Ett fält kan ses som en variabel som lagrar adressen till första värdet i en kontinuerlig sekvens. Notera likheten med pekardefinitionen.

Detta gör det OK att hantera fält som pekare:

```
int index_one = *(array + 1);
```

Det är även OK att hantera pekare som fält:

```
int* p = array;
int index_one = p[1];
```

Pekararitmetik

Blir det inte fel när fält hanteras som pekare?

```
int index_one = *(array + 1);
```

Fältet `array` lagrar ju heltal, och varje heltal tar upp flera byte (`sizeof(int)`) i minnet.

Kompilatorn vet från deklarationen att det ligger heltal på den adress som anges i variabeln array. Då En adress adderas med ett heltal multiplicerar kompilatorn automatiskt heltalet med storleken på det som lagras på adressen. Det som faktiskt utförs blir alltså:

```
... = *(array + 1*sizeof(*array));
```

Fältstorlek

Notera att varje fält har den storlek du anger vid deklaration. Storleken går inte att ändra, och det är fel att läsa skriva utanför fältet.

OBS! Det fings inget som hindrar dig från *att* göra ovanstående fel, och det finns ingenting som varnar dig *när* du gör ovanstående fel. Det finns heller ingenting som hjälper dig hålla reda på vilken storlek ett fält har.

Fältstorlek: Fel med sizeof

```
void print_sizeof(int a[]) {
  printf("%d\n", sizeof(a)); // FEL, ger 8
}
int main() {
  int data[18] = {1,2,3,4,5,6,7,8,9};
  printf("%d\n", sizeof(data)); // FEL, ger 36
  print_sizeof(data);
  return 0;
}
```

Är utskrifterna 8 och 36 vad du förväntar?

Om du tror båda utskrifterna borde bli 18 måste du tänka om.

Fältstorlek: Rätt med sizeof

Som du såg i föregående slide ger sizeof för en array inte arraystorleken så som vi skulle vilja. Ibland går den dock som synes nedan att räkna ut.

Om namnet på något som är deklarerat med en fast fältstorlek anges så fungerar detta:

```
double data[9];
int size = sizeof(data)/sizeof(*data); // OK 72/8
```

Om namnet på något som är deklarerat som pekare anges så fungerar det fortfarande INTE:

```
double* ptr = data;
int length = sizeof(ptr)/sizeof(*ptr); // FEL 8/8
```

Notera att både en double och en pekare är 8 byte.

void och void*

Ordet "void" betyer "tomrum", "tomhet", "ogiltig" eller "ledig".

I C-program används void där språkets syntax kräver en datatyp, men programmeraren inte har behov av någonting. Det brukar vara då en tom parameterlista önskas, eller då inget returvärde önskas.

Det går inte att skapa variabler av typen void. (Vad skulle en "tom" eller "ogiltig" variabel vara bra för?)

Däremot går det att skapa pekarvariabler till void, dvs variabler som innehåller en adress, och på den adressen lagras "ingenting", eller snarare "vad som helst". Det är ganska fult och osäkert då det endast är programmeraren som kan, *och måste*, hålla reda på vad som finns på void-pekar adresser. Därmed ökar risken för fel och missförstånd. Alltså bör void-pekare undvikas.

Pekarvariabler av denna typ går varken att avreferera med * eller att indexera med []. Det är inte heller rekommenderat att utföra aritmetik (+,-) med void-pekare. Kompilatorn vet ju inte vad som lagras på adressen, hur mycket minne det tar upp, eller hur det skall tolkas. För att läsa från en adress till void måste kompilatorn först informaras om vad pekarvariabelns adress egentligen lagrar. Det kan göras med en vanlig typkonvertering:

```
int i = 4711;
void* ip = &i;
int x = *ip;          /* KOMPILERINGSFEL: vad pekar ip till? */
int y = *(double*)ip; /* FEL: ip anger inte adress till double */
int z = *(int*)ip;    /* RÄTT */
```

Teckenpekare, teckenfält, strängar

I C representeras ett tecken av en byte, 8 bitar. Det betyder att 256 olika tecken kan representeras. Dock är det inte så vanligt att tecken förekommer enskilt, oftare använder vi dem i sekvens för att bilda ord och meningar.

I C finns i egentlig mening inget språkstöd för detta. Det får lösas bäst det går med hjälp av fält och pekare. Den standard för hur detta löses i C bygger på två förutsättningar:

1. Eftersom alla gemener, versaler och siffror inte behöver mer än 60 av detta finns åtskilligt utrymme att tolka resterande värden speciellt. Speciellt tolkas teckenvärdet motsvarande att alla 8 bitar är 0 som strängavsluttecken. Detta tecken skrivs '\0'.
2. De flesta strängar är relativt korta, så att räkna ut längden på varje sträng är bättre än att lagra en extra variabel med storleken. (Detta kom till i tider då minne var en mycket dyr bristvara. Att spara in två byte genom att spara endast två siffror för årtal sågs som en lika bra idé som att inte lagra längden av strängar.)

Summan av detta blir att en sträng representeras av adressen till dess första tecken. Strängen fortsätter sekventiellt i minnet tills ett noll-tecken dyker upp.

Litterala strängar

En sträng kan skapas som ett fält:

```
char str1[6] = {'k', 'a', 'l', 'l', 'e', '\0'};
```

Detta är dock inte särskilt bekvämt. Språket medger därför lite enklare syntax, vilket i pincip är det språkstöd som finns:

```
const char* str2 = "kalle";
```

Text inom citationstecken i koden kommer lagras i statiskt (ej ändringsbart) minne av kompilatorn, ett strängavsluttecken läggs automatiskt till, och hela strängen ger som värde adressen till första tecknet. Datatypen blir därför `const char*` (adress till konstanta tecken).

Strängkopiering

Notera att varken fält eller strängar kopieras automatiskt:

```
char* str3 = str1;
```

Nu blir `str3` en variabel som innehåller adressen till första tecknet i strängen `str1` (d.v.s. en ren kopia av adressen i pekarvariablen `str1`).

För att kopiera alla tecken i strängen måste mer jobb göras:

```
int size = strlen(str1) + 1;
char* str4 = malloc(size);
strlcpy(str4, str1, size);
```

Så snart du aldrig behöver använda `str4` igen måste du komma ihåg att göra `free(str4)`.

För fält måste du göra en egen kopieringsloop, `strlcpy` fungerar endast för teckensekvenser och avbryter vid strängavsluttecken.

NULL vs '\0'

Noll-tecken skall aldrig förväxlas med NULL-pekare:

- Noll-tecken är en byte där alla bitar är 0, tolkas som ett *tecken*, och *skrivs '\0'*. Noll-tecken används som *slutmarkör för strängar*.
- NULL-pekare är 8 byte där alla bitar är 0, tolkas som en *adress*, och *skrivs NULL*. NULL-pekare används för att *markera pekarvariabler som ogiltiga* (lagrar ingen giltig adress).

Det är två fullkomligt ***olika*** begrepp.

Exempel: NULL vs '\0'

```
char* str1 = NULL;
```

Pekarvariabeln `str1` är ogiltig (innehåller adressen 0). `sizeof(str1)` ger 8. `strlen(str1)` ger *segmentation fault*.

```
char* str2 = "";
```

Pekarvariabeln `str2` är giltig och innehåller adressen till en sträng. Strängen är tom, d.v.s. den innehåller endast strängavslutttecknet. `sizeof(str2)` ger 8. `strlen(str2)` ger 0.

```
char str3[1] = {'\0'};
```

Fältet `str3` innehåller endast strängavslutteckent. Konceptuellt är detta identiskt med `str2`, men kompilatorn gör vissa skillnader. `sizeof(str3)` ger 1. `strlen(str3)` ger 0.

Exempel: Rita fält och pekare

Hur ritas följande upp med boxar och pilar?

```
const char* str = ”kalle”;
char* cpy = NULL;
int* data[6];
int** data_p;

cpy = malloc(10);
strlcpy(cpy, str, 10);
data_p = data + 3;
data[2] = data_p[1];
data_p = &data[2];
free(cpy);
```

Öva själv... fråga gärna på lab om du ritat rätt.

Exempel: Pekardeklarationer

```
int* ptoi;
```

Varianter med const:

```
const int* ptoci;
int const* ptoci;
int *const cptoi;
int const* const cptoci;
int* aofip[];
int (*ptoaofi)[]
int* (*ptof)(int**);
int* (*aofptof[])(int**);
```

Läsa deklarationer baklänges fungerar bättre!

Föregående slide utläsa

Läsa deklarationer baklänges fungerar bättre!

Dock måste parenteser, undantaget parameterlistor, behandlas först!

1. Variabeln ptoi är en pekare till heltal.
   - Variabeln ptoci är en pekare till heltal som är konstanta.
   - Variabeln ptoci är en pekare till konstanta heltal.
   - Variabeln cptoi är en konstant pekare till heltal.
   - Variabeln cptoci är en konstant pekare till konstanta heltal.
2. Arrayvariableb aofip lagrar pekare till heltal.
3. Variabeln ptoaofi är en pekare till en array som lagrar heltal.
4. Variablen ptof är en pekare till en funktion som tar en int** som parameter och returnerar en int*
5. Arrayvariabeln aofptof lagrar pekare till funktioner som tar int** som parameter och returnerar int*

Tips: Använd `typedef` i flera steg för att göra begripliga deklarationer!

typedef i flera steg

```
int* (*aofptof[5])(int**); // Ouch!
```

Deklarera så här istället!

```
typedef int* function_type(int**);
typedef function_type* function_pointer;

function_pointer aofptof[5]; // Ta-da!!
```

Och använd bättre namn än aofptof.

Utskrift med printf

Används för formaterad utskrift i C. Tar minst en parameter, den första, som måste vara av typen `const char*`. Övriga parametrar anges vid behov.

Speciella formatflaggor infogas i första strängen för att stoppa in värdet av andra variabler där. Variablerna anges sedan som extra parametrar (i rätt ordning).

```
int age = 10;
double weight = 123.4567;
printf(”Eva\n”);
printf(”Maja fyller %d år.\n”, age);
printf(”Nils blir %d år om %d dagar.\n”, 9, 5);
printf(”Bertil väger %f kilo.\n”, weight);
printf(”Stina säger %s\n”, ”Hej!”);
printf(”%s %f %s %d %s %c”,
       ”Talet”, 3.14, ”har mer än”,
       9, ”siffror”, ’\n’);
```

Formatering med printf

Det går lätt att åstadkomma mer avancerad formatering med `printf`:

```
printf(”%4d”, 1);
```

Använder minst 4 teckenpositioner att skriva ut heltalet. De positioner som inte behövs fylls med blanksteg. Talet högerjusteras.

```
printf(”%-20s”, ”Karl XII”);
```

Vänsterjusterad sträng som tar upp 20 teckenpositioner (minst).

```
printf(”%6.2f”);
```

Flyttal som tar upp 6 teckenpositioner, avrundat till 2 decimaler.

Fel med printf

Första parametern till `printf` skall alltid vara en litteral sträng.

```
char user_input[64];
printf(user_input); // FEL
```

Tänk om indatan innehåller ett % som får printf att läsa en parameter som inte finns?

```
printf("%s", user_input); // RÄTT
```

Fikonspråk

(eller korrekt terminologi)

Vet du vad följande ord betyder? Tror du några är synonymer?
Nej. Alla betyder distinkt skilda saker. Läs på skillnaden.

- parameter
- argument
- avreferering
- syntax
- semantik
- literal
- deklaration
- definition

There's more...

- struct, arrays of structs
- headerfiles is included with the extension .h
- static: it's both private and global
  - kan åtgärda kompileringsvarningen: function is not prototype
- void for empty parameter list
  - kan åtgärda kompileringsvarningen: function is not prototype
- create block for each case in a switch
- must declare variables at start of block
- can not declare variable in for-statement
- restrict: pointed to memory not accessed from other pointers
- includes: string.h, ctype.h, stdlib.h, stdio.h
- man-pages: (section 3) strlen, tolower, printf, (scanf, gets)
  - man -s1 chown (sektion 1, terminalkommandot)
  - man -s2 chown (sektion 2, systemanropet)
  - man -s3p chown (sektion 3p, POSIX funktionsanrop)