INF-0103 Minne og pekere

Jakob Peder Pettersen, UiT – Norges arktiske universitet

Høst 2025

Dagens tema

- Pass-by-value og pass-by-referense

Dagens tema

- Pass-by-value og pass-by-referense
- Hva er pekere?

Dagens tema

- Pass-by-value og pass-by-referense
- Hva er pekere?
- Tabeller (**arrays**)

Dagens tema

- Pass-by-value og pass-by-referense
- Hva er pekere?
- Tabeller (**arrays**)
- Peker-aritimetikk

Dagens tema

- Pass-by-value og pass-by-referense
- Hva er pekere?
- Tabeller (**arrays**)
- Peker-aritimetikk
- Størrelsen på variabler

Et (forhåpentligvis) kjent eksempel: `printf` og `scanf`

- Vi kan skrive `printf("Tallet er: %d\n", tall);`

Et (forhåpentligvis) kjent eksempel: `printf` og `scanf`

- Vi kan skrive `printf("Tallet er: %d\n", tall);`
- Hva er da problemet med `scanf("%d", tall);`?

Et (forhåpentligvis) kjent eksempel: `printf` og `scanf`

- Vi kan skrive `printf("Tallet er: %d\n", tall);`
- Hva er da problemet med `scanf("%d", tall);`?
- Hvorfor må vi skrive `scanf("%d", &tall);`?

Et (forhåpentligvis) kjent eksempel: `printf` og `scanf`

- Vi kan skrive `printf("Tallet er: %d\n", tall);`
- Hva er da problemet med `scanf("%d", tall);`?
- Hvorfor må vi skrive `scanf("%d", &tall);`?
- Hva er feil med `printf("Tallet er: %d\n", &tall);`?

Et (forhåpentligvis) kjent eksempel: `printf` og `scanf`

- Vi kan skrive `printf("Tallet er: %d\n", tall);`
- Hva er da problemet med `scanf("%d", tall);`?
- Hvorfor må vi skrive `scanf("%d", &tall);`?
- Hva er feil med `printf("Tallet er: %d\n", &tall);`?
- Hva gjør egenlig `&`?

Forskjell på pass-by-reference og pass-by-value

- `printf()` bruker **pass-by-value**, verdiene som skrives ut kopieres inn i funksjonen

Forskjell på pass-by-reference og pass-by-value

- `printf()` bruker **pass-by-value**, verdiene som skrives ut kopieres inn i funksjonen
- Hvorfor vil dette ikke fungere for `scanf()`?

Forskjell på pass-by-reference og pass-by-value

- `printf()` bruker **pass-by-value**, verdiene som skrives ut kopieres inn i funksjonen
- Hvorfor vil dette ikke fungere for `scanf()`?
- `scanf()` må bruke **pass-by-reference**, en referanse til variablene blir sendt til funksjonen uten kopiering av de underliggende verdiene

Pekere (**pointers**)

- Alle C-funksjoner bruker **pass-by-value**, men det finnes en egen kategori variabler som inneholder referansen til andre variabler

Pekere (**pointers**)

- Alle C-funksjoner bruker **pass-by-value**, men det finnes en egen kategori variabler som inneholder referansen til andre variabler
- Disse kalles **pekere** (**pointers**)

Pekere (**pointers**)

- Alle C-funksjoner bruker **pass-by-value**, men det finnes en egen kategori variabler som inneholder referansen til andre variabler
- Disse kalles **pekere** (**pointers**)
- Notasjon for type: Grunntype med `*` etter (`int -> int*`, `double -> double*`).

Pekere (**pointers**)

- Alle C-funksjoner bruker **pass-by-value**, men det finnes en egen kategori variabler som inneholder referansen til andre variabler
- Disse kalles **pekere** (**pointers**)
- Notasjon for type: Grunntype med `*` etter (`int` -> `int*`, `double` -> `double*`).
- `&` er en operator som returerer **minneaddressen** til en variabel

Pekere (**pointers**)

- Alle C-funksjoner bruker **pass-by-value**, men det finnes en egen kategori variabler som inneholder referansen til andre variabler
- Disse kalles **pekere** (**pointers**)
- Notasjon for type: Grunntype med `*` etter (`int` -> `int*`, `double` -> `double*`).
- `&` er en operator som returerer **minneaddressen** til en variabel
- Dereferenceringsoperatoren `*` (brukt unitært) gjør det motsatte, den henter ut verdien som det pekes på

Eksempel med pass-by-value og pass-by-reference

Hva foregår her?

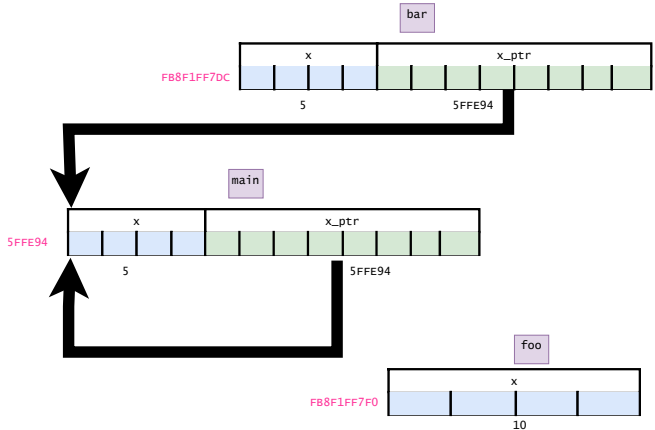

Figure 1: Variabler og addresser i programmet vårt

Semantikk for verdier

Hvorfor fungerer ikke dette?

```
int a = 4;
int b = 9;
int* tall_sum_peker = &(a + b);
```

- a og b er venstreverdier (**lvalues**) -> Har sin egen plass i minnet. Man kan tilordne verdier og ta addressen.

Semantikk for verdier

Hvorfor fungerer ikke dette?

```
int a = 4;
int b = 9;
int* tall_sum_peker = &(a + b);
```

- a og b er venstreverdier (**lvalues**) -> Har sin egen plass i minnet. Man kan tilordne verdier og ta addressen.
- 4 og (a + b) er høyreverdier (**rvalues**) -> Har ikke sin egen plass i minnet. Man kan **ikke** tilordne verdier eller ta addressen.

Forskjellige typer pekere

- Fundamentale datatyper: `int*`, `char*`, `double*`

Forskjellige typer pekere

- Fundamentale datatyper: `int*`, `char*`, `double*`
- Sammensatte strukturer (**structs**): `Person*`

Forskjellige typer pekere

- Fundamentale datatyper: `int*`, `char*`, `double*`
- Sammensatte strukturer (**structs**): `Person*`
- Andre pekere: `int**`, `char**`

Forskjellige typer pekere

- Fundamentale datatyper: `int*`, `char*`, `double*`
- Sammensatte strukturer (**structs**): `Person*`
- Andre pekere: `int**`, `char**`
- Funksjoner: `void (*)(int *)` (signaturen til `&bar` i eksemplet vårt)

Forskjellige typer pekere

- Fundamentale datatyper: `int*`, `char*`, `double*`
- Sammensatte strukturer (**structs**): `Person*`
- Andre pekere: `int**`, `char**`
- Funksjoner: `void (*)(int *)` (signaturen til `&bar` i eksemplet vårt)
- Ubestemt type: `void*`

Minneaddresser er “bare” tall

- Pekere kan hardkodes til å peke på en bestemt addresse:

```
// Nesten alltid en dårlig idé
int *p = x03ab; // Heksadesimale tall for minneaddresser
```

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

Minneaddresser er “bare” tall

- Pekere kan hardkodes til å peke på en bestemt addresse:

```
// Nesten alltid en dårlig idé
int *p = x03ab; // Heksadesimale tall for minneaddresser
```

- Vanligvis er det bare *en* hardkodet addresse vi behøver å bry oss om: `NULL` med definisjonen:

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

Minneaddresser er “bare” tall

- Pekere kan hardkodes til å peke på en bestemt addresse:

```
// Nesten alltid en dårlig idé
int *p = x03ab; // Heksadesimale tall for minneaddresser
```

- Vanligvis er det bare *en* hardkodet addresse vi behøver å bry oss om: `NULL` med definisjonen:

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

- Brukes ofte som feilkode eller for å indikere at pekeren ikke er i bruk

Oppgave i pausen: Hva er galt her?

```
int* foo() {
    int verdi = 204;
    return &verdi;
}

int main() {
    int *p = foo();
    int verdi = *p;
    return 0;
}
```

Pekere og gyldighet

- Å dereferere ugyldig minne er udefinert oppførsel (**undefined behavior**)

Pekere og gyldighet

- Å dereferere ugyldig minne er udefinert oppførsel (**undefined behavior**)
- Å *ha* en peker som ikke peker på gyldig minne er derimot OK (jamfør `NULL`)

Pekere og gyldighet

- Å dereferere ugyldig minne er udefinert oppførsel (**undefined behavior**)
- Å *ha* en peker som ikke peker på gyldig minne er derimot OK (jamfør `NULL`)
- Vi må selv avgjøre utefra konteksten når det er trygt å bruke en peker

Pekere og gyldighet

- Å dereferere ugyldig minne er udefinert oppførsel (**undefined behavior**)
- Å *ha* en peker som ikke peker på gyldig minne er derimot OK (jamfør `NULL`)
- Vi må selv avgjøre utefra konteksten når det er trygt å bruke en peker
- Vi har ingen garanti for hva som skjer

Pekere og gyldighet

- Å dereferere ugyldig minne er udefinert oppførsel (**undefined behavior**)
- Å *ha* en peker som ikke peker på gyldig minne er derimot OK (jamfør `NULL`)
- Vi må selv avgjøre utefra konteksten når det er trygt å bruke en peker
- Vi har ingen garanti for hva som skjer
- Ikke garantert noe kræsj, kan endre seg av hvilken som helst grunn

Statiske tabeller (**arrays**)

- Struktur der flere elementer av samme type ligger etter hverandre

```
int arr[5];
```

```
int arr[] = {-4, 5, 2, 130, -11};
```

Statiske tabeller (**arrays**)

- Struktur der flere elementer av samme type ligger etter hverandre
- Kompilatoren må vite størrelsen på tabellen på forhånd

```
int arr[5];
```

```
int arr[] = {-4, 5, 2, 130, -11};
```

Statiske tabeller (**arrays**)

- Struktur der flere elementer av samme type ligger etter hverandre
- Kompilatoren må vite størrelsen på tabellen på forhånd
- Syntaks:

```
int arr[5];


int arr[] = {-4, 5, 2, 130, -11};
```

Statiske tabeller (**arrays**)

- Struktur der flere elementer av samme type ligger etter hverandre
- Kompilatoren må vite størrelsen på tabellen på forhånd
- Syntaks:

```
int arr[5];
```

- eller

```
int arr[] = {-4, 5, 2, 130, -11};
```

Statiske tabeller (**arrays**)

- Struktur der flere elementer av samme type ligger etter hverandre
- Kompilatoren må vite størrelsen på tabellen på forhånd
- Syntaks:

```
int arr[5];
```

- eller

```
int arr[] = {-4, 5, 2, 130, -11};
```

- Oppfører seg som en mellomting mellom "vanlige" datatyper og pekere

Peker-aritimetikk

- Mest relevant med tabeller og liknende strukturer

Peker-aritimetikk

- Mest relevant med tabeller og liknende strukturer
- Aritmetriske operasjoner på pekere og tabeller

Peker-aritimetikk

- Mest relevant med tabeller og liknende strukturer
- Aritmetriske operasjoner på pekere og tabeller
- For variabler bestående av flere bytes bruker vi den første addressen

Peker-aritimetikk

- Mest relevant med tabeller og liknende strukturer
- Aritmetriske operasjoner på pekere og tabeller
- For variabler bestående av flere bytes bruker vi den første addressen
- Tabeller er null-indekserte av naturlige grunner

Peker-aritimetikk

- Mest relevant med tabeller og liknende strukturer
- Aritmetriske operasjoner på pekere og tabeller
- For variabler bestående av flere bytes bruker vi den første addressen
- Tabeller er null-indekserte av naturlige grunner
- Ofte er indekseringsnotasjonen `arr[3]` (`*(arr + 3)`) mer hensiktsmessig

Eksempel med pekeraritimetikk

Oppførsel til tabeller i funksjoner

- Kan være argument til andre funksjoner, men i virkeligheten det er addressen til tabellen som kopieres (unntak fra pass-by-value)

Oppførsel til tabeller i funksjoner

- Kan være argument til andre funksjoner, men i virkeligheten det er addressen til tabellen som kopieres (unntak fra pass-by-value)
- Fenomenet kalles **pointer decay**

Oppførsel til tabeller i funksjoner

- Kan være argument til andre funksjoner, men i virkeligheten det er addressen til tabellen som kopieres (unntak fra pass-by-value)
- Fenomenet kalles **pointer decay**
- Ingen informasjon om antall elementer det pekes til -> Vi må often oppgi lengden i tillegg til pekeren

Eksempel med sortering av tabell

Hva hvis antall elementer ikke kjent på forhånd?

- Buffer: Lag en tabell som forhåpentligvis er stor nok til å romme data

Hva hvis antall elementer ikke kjent på forhånd?

- Buffer: Lag en tabell som forhåpentligvis er stor nok til å romme data
- Bruk bare den første biten

Hva hvis antall elementer ikke kjent på forhånd?

- Buffer: Lag en tabell som forhåpentligvis er stor nok til å romme data
- Bruk bare den første biten
- Ofte brukt til tekststrenger (mer om det senere i kurset)

Hva hvis antall elementer ikke kjent på forhånd?

- Buffer: Lag en tabell som forhåpentligvis er stor nok til å romme data
- Bruk bare den første biten
- Ofte brukt til tekststrenger (mer om det senere i kurset)
- Sløsing med minne

Hva hvis antall elementer ikke kjent på forhånd?

- Buffer: Lag en tabell som forhåpentligvis er stor nok til å romme data
- Bruk bare den første biten
- Ofte brukt til tekststrenger (mer om det senere i kurset)
- Sløsing med minne
- To hovedstrategier:

Hva hvis antall elementer ikke kjent på forhånd?

- Buffer: Lag en tabell som forhåpentligvis er stor nok til å romme data
- Bruk bare den første biten
- Ofte brukt til tekststrenger (mer om det senere i kurset)
- Sløsing med minne
- To hovedstrategier:
  - Tabeller med variabel lengde (**variable-length array**, **VLA**)

Hva hvis antall elementer ikke kjent på forhånd?

- Buffer: Lag en tabell som forhåpentligvis er stor nok til å romme data
- Bruk bare den første biten
- Ofte brukt til tekststrenger (mer om det senere i kurset)
- Sløsing med minne
- To hovedstrategier:
  - Tabeller med variabel lengde (**variable-length array**, **VLA**)
  - Dynamisk minneallokering (kommer senere i kurset)

Tabeller med variabel lengde

- Deklareres på samme måte som statiske tabeller, men lengden er ikke et konstantuttrykk

Tabeller med variabel lengde

- Deklareres på samme måte som statiske tabeller, men lengden er ikke et konstantuttrykk
- Allokeres ved deklarasjon

Tabeller med variabel lengde

- Deklareres på samme måte som statiske tabeller, men lengden er ikke et konstantuttrykk
- Allokeres ved deklarasjon
- Deallokeres automatisk når tabellen går ut av scope

Tabeller med variabel lengde

- Deklareres på samme måte som statiske tabeller, men lengden er ikke et konstantuttrykk
- Allokeres ved deklarasjon
- Deallokeres automatisk når tabellen går ut av scope
- Oppfører seg ellers som en vanlig tabell

Tabeller med variabel lengde

- Deklareres på samme måte som statiske tabeller, men lengden er ikke et konstantuttrykk
- Allokeres ved deklarasjon
- Deallokeres automatisk når tabellen går ut av scope
- Oppfører seg ellers som en vanlig tabell
- Ikke tilgjengelig i C++

Eksemplet vårt med brukerbestemt lengde

`sizeof`-operatoren

- Forteller hvor mange bytes en variabel eller en datatype tar opp i minnet

`sizeof`-operatoren

- Forteller hvor mange bytes en variabel eller en datatype tar opp i minnet
- Kan brukes direkte på datatypen `sizeof(int)` eller på en variabel `sizeof(x)`.

`sizeof`-operatoren

- Forteller hvor mange bytes en variabel eller en datatype tar opp i minnet
- Kan brukes direkte på datatypen `sizeof(int)` eller på en variabel `sizeof(x)`.
- Alle pekere tar like mye plass (de er alle minneaddresser)

`sizeof`-operatoren

- Forteller hvor mange bytes en variabel eller en datatype tar opp i minnet
- Kan brukes direkte på datatypen `sizeof(int)` eller på en variabel `sizeof(x)`.
- Alle pekere tar like mye plass (de er alle minneaddresser)
- Relasjon til peker-aritimetikk: `p + n` gir en peker som er `n * sizeof(*p)` bytes lenger framme enn `p`

Eksempel med `sizeof`-operatoren