

Effektiv Kode med C og C++

Forelesning 3, vår 2014
Alfred Bratterud

Agenda:

- * Semesterplan lagt ut
- * Ulike skop
- * Typer av minne og objekters levetid
- * Stack
- * Minnelekkasjer
- * Referanser

Repetisjon

- * Er en peker en datatype?
- * Hvorfor?
- * Er pekere trygge? Eksempler?
- * Når er de nødvendige?

Minne

Viktig viktig pensum til prøven
...Få det både inn i hodet og i fingrene

Minnetilgang = mye kraft + mye ansvar

- * Hva skjer her?

```
void* verySmart(){  
    int arr[5000000];  
    //Do some stuff.  
    return (void*)arr;  
}
```

- * Initialisering? Allokering? Type?
Størrelse?

Fem typer skop dere må vite om

- * “Global scope” - alt som ikke er i et annet skop
- * “Namespace scope” - definert i et “namespace”, men ikke dypere
- * “Local scope” - inni en funksjon
- * “Class scope” - inni en medlemsfunksjon
- * “Statement scope” - inni et uttrykk, som feks. `while(true){int i=8};`

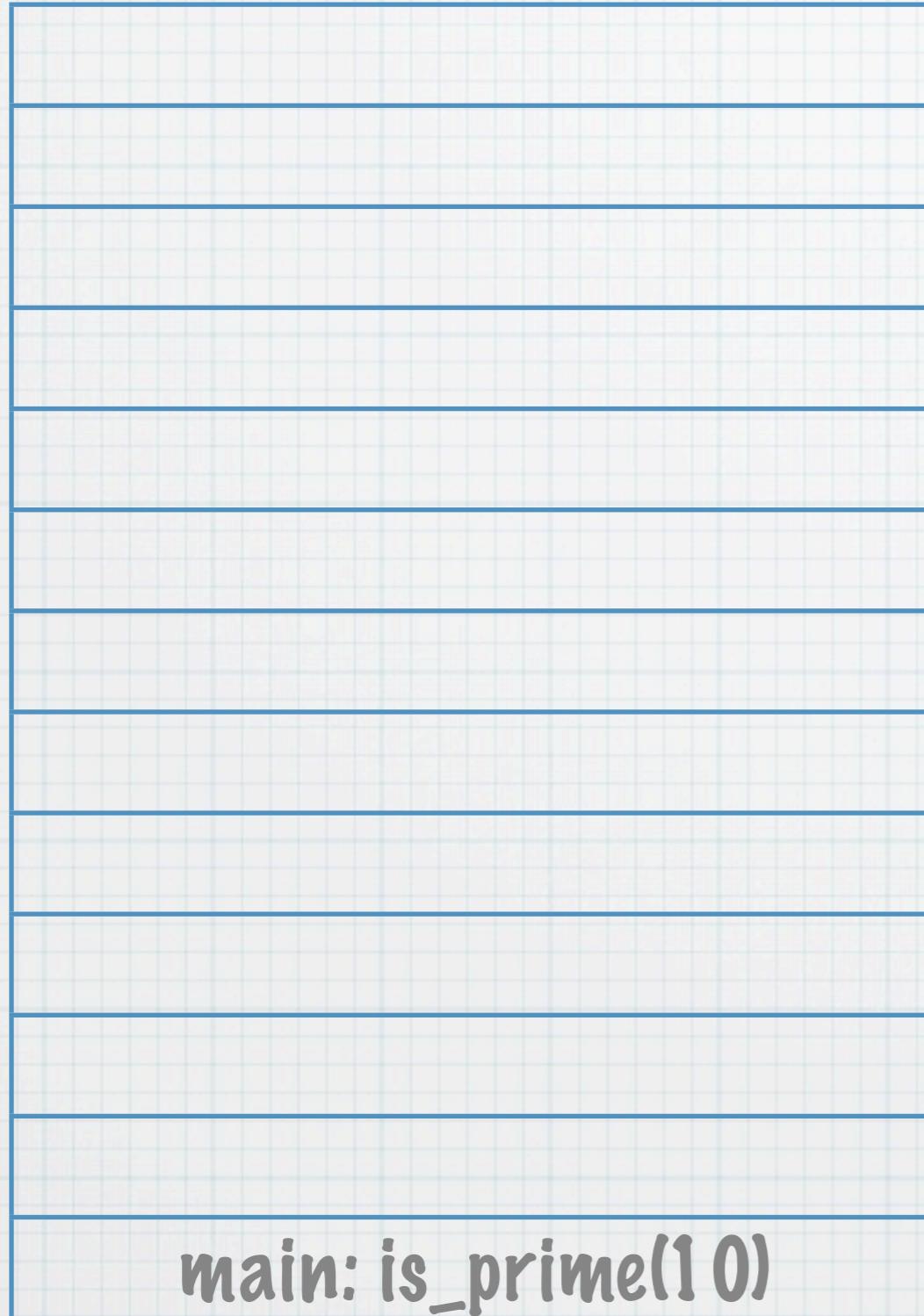
Tre typer minnelagring dere må vite om

- * “Automatic storage”:
 - * Blir borte når funksjonen returnerer
- * “Static storage”:
 - * Blir borte når prosessen avslutter
- * Dynamic memory, “Free store (heap)”:
 - * Blir borte når DU gir beskjed (og når prosessen avslutter)

Automatic Storage

- * Kalles gjerne "Stack"
- * Variable definert inni funksjoner - og parametre - legges her
- * Kan gjenbrukes av andre funksjoner, når kallet har returnert
- * Mange "kopier" kan eksistere, hvis funksjonen kaller seg selv
- * Er svært begrenset i størrelse (feks. 10 MB)

Stack (Forts.)



- * OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
- * Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
- * Hvert funksjonskall dyster sp videre opp...
- * Blir det er fullt - Stack overflow!
- * Return...? Håper vi lagret forrige kalls stack pointer!
- * C++: Stack unwind. Alle destructorer kalles

Stack (Forts.)

sp>

```
int x=10; ...
is_prime(10)
main: is_prime(10)
```

- * OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
- * Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
- * Hvert funksjonskall dyster sp videre opp...
- * Blir det er fullt - Stack overflow!
- * Return...? Håper vi lagret forrige kalls stack pointer!
- * C++: Stack unwind. Alle destruktører kalles

Stack (Forts.)

sp>

```
int i=9;  
gcd(1 0)  
int x=1 0; ...  
is_prime(1 0)  
main: is_prime(1 0)
```

- * OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
- * Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
- * Hvert funksjonskall dyster sp videre opp...
- * Blir det er fullt - Stack overflow!
- * Return...? Håper vi lagret forrige kalls stack pointer!
- * C++: Stack unwind. Alle destruktører kalles

Stack (Forts.)

sp>

```
int i=8;  
gcd(9)  
int i=9;  
gcd(10)  
int x=10; ...  
is_prime(10)  
main: is_prime(10)
```

- * OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
- * Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
- * Hvert funksjonskall dyster sp videre opp...
- * Blir det er fullt - Stack overflow!
- * Return...? Håper vi lagret forrige kalls stack pointer!
- * C++: Stack unwind. Alle destruktører kalles

Stack (Forts.)

sp>

```
int i=7...
gcd(8);
int i=8;
gcd(9)
int i=9;
gcd(10)
int x=10; ...
is_prime(10)
main: is_prime(10)
```

- * OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
- * Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
- * Hvert funksjonskall dyster sp videre opp...
- * Blir det er fullt - Stack overflow!
- * Return...? Håper vi lagret forrige kalls stack pointer!
- * C++: Stack unwind. Alle destruktorer kalles

Stack (Forts.)

sp>

```
...  
gcd(7)  
int i=7;  
gcd(8);  
int i=8;  
gcd(9)  
int i=9;  
gcd(10)  
int x=10; ...  
is_prime(10)  
main: is_prime(10)
```

- * OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
- * Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
- * Hvert funksjonskall dyster sp videre opp...
- * Blir det er fullt - Stack overflow!
- * Return...? Håper vi lagret forrige kalls stack pointer!
- * C++: Stack unwind. Alle destruktorer kalles

sp>

Stack overflow!

Stack (Forts.)

```
...  
gcd(7)  
int i=7;  
gcd(8);  
int i=8;  
gcd(9)  
int i=9;  
gcd(10)  
int x=10; ...  
is_prime(10)  
main: is_prime(10)
```

- * OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
- * Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
- * Hvert funksjonskall dyster sp videre opp...
- * Blir det er fullt - Stack overflow!
- * Return...? Håper vi lagret forrige kalls stack pointer!
- * C++: Stack unwind. Alle destructorer kalles

Stack (Forts.)

sp>

gcd(0) : return!

...

gcd(7)

int i=7...

gcd(8);

int i=8;

gcd(9)

int i=9;

gcd(10)

int x=10; ...

is_prime(10)

main: is_prime(10)

- * OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
- * Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
- * Hvert funksjonskall dyster sp videre opp...
- * Blir det er fullt - Stack overflow!
- * Return...? Håper vi lagret forrige kalls stack pointer!
- * C++: Stack unwind. Alle destruktører kalles

Stack (Forts.)

sp>

```
gcd(7)
int i=7...
gcd(8);
int i=8;
gcd(9)
int i=9;
gcd(10)
int x=10; ...
is_prime(10)
main: is_prime(10)
```

- * OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
- * Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
- * Hvert funksjonskall dyster sp videre opp...
- * Blir det er fullt - Stack overflow!
- * Return...? Håper vi lagret forrige kalls stack pointer!
- * C++: Stack unwind. Alle destruktorer kalles

Stack (Forts.)

sp>

```
int x=10; ...
is_prime(10)
main: is_prime(10)
```

- * OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
- * Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
- * Hvert funksjonskall dyster sp videre opp...
- * Blir det er fullt - Stack overflow!
- * Return...? Håper vi lagret forrige kalls stack pointer!
- * C++: Stack unwind. Alle destruktører kalles

Stack (Forts.)

sp>

main: is_prime(10)

- * OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
- * Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
- * Hvert funksjonskall dyster sp videre opp...
- * Blir det er fullt - Stack overflow!
- * Return...? Håper vi lagret forrige kalls stack pointer!
- * C++: Stack unwind. Alle destructorer kalles

Stack (Forts.)

sp>

```
printf("Not a prime!")
```



```
main: is_prime(10)
```

- * OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
- * Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
- * Hvert funksjonskall dyster sp videre opp...
- * Blir det er fullt - Stack overflow!
- * Return...? Håper vi lagret forrige kalls stack pointer!
- * C++: Stack unwind. Alle destructorer kalles

Stack (Forts.)

sp>
printf("Not a prime!")
main: is_prime(10)

- * OS'et leverer en stack til prosessen, som den får skrive til. Main nederst.
- * Hver funksjon tar selv ansvar for å sette av nok plass på stack til sine egne "automatiske variabler" (Den flytter bare stack pointer). Kompilatoren fikser dette for oss.
- * Hvert funksjonskall dyster sp videre opp...
- * Blir det er fullt - Stack overflow!
- * Return...? Håper vi lagret forrige kalls stack pointer!
- * C++: Stack unwind. Alle destruktører kalles

Stack (Forts.)

0x...a6

"false"

int* res=&i; return res

sp>

printf("Prime?%i",*res)

main: is_prime(10)

- * Returnerer man pekere, til lokale variable...
- * kan man ha flaks

Stack (Forts.)

0x...a6

42

int* res=&i; return res

sp>

printf("Prime?%i",*res)

main: is_prime(10)

- * Returnerer man pekere, til lokale variable...
- * kan man ha flaks
- * ...med mindre det har skjedd en ny serie av funksjonskall i mellomtiden

Automatic Storage (forts.)

- * Kalles gjerne "Stack" - men det sier ikke alt
- * Inkluderer all minnehåndtering kompilatoren kan gjøre for oss, (som å flytte `sp`), men også å lagre mellomregninger, som `i=10*sqrt(2)+5`
- * Er svært begrenset i størrelse (feks. 10 MB). Hvor stor er den hos deg? Prøv!
- * Hvilken feilmelding vil jeg få hvis jeg skriver over stacken? Prøv!

Static Storage

- * Det som er deklarert i “global scope” og “namespace scope” ligger her
- * Kan ikke endre størrelse- må settes “link-time”, før kjøring
- * Variable i klasser og funksjoner som er eksplisitt definert “static” ligger også her

Dynamic memory / Free Store (heap)

- * Det som er allokeret med "new" (C++) legges i free store
 - * Må frigjøres med "delete"
- * Det som er allokeret med "malloc" (C) legges på heap. Kan være samme sted, kan være annet.
 - * Må frigjøres med "free"
- * Dette er eneste måten å gjøre "manuell" "dynamisk allokering" på, dvs. "runtime".

Free Store (forts.)

- * **malloc(n)** returnerer en **void*** til **n** bytes utenfor stack, i "trygt område"... bra?
 - * Det tvinger oss til å gjøre en eksplisitt cast (vi kan ikke bruke **void*** i beregninger)
 - * Vi må (noen ganger) huske hvor stor **n** var
- * **new** er typet, og returnerer peker av riktig type - og dermed riktig størrelse. (**int* x=new int(9)**)
- * Men: begge lagrer data i "free store", og DU må ta ansvar for at dette ryddes opp. Ellers?

Minnelekkasjer

- * ALDRI kall **malloc** (punktum) uten **free**
- * ALDRI kall **new** uten **delete**
- * I C++ trenger du (nesten) bare å bruke new, når du lager datastrukturer
(og når du bruker C-biblioteker)
 - * STL-containere kan opprettes fritt, uten å tenke på dette. De gjør det for deg.
- * Usikker på om du har en lekkasje? Installer og kjør programmet ditt med valgrind: <http://valgrind.org/>

Nå: Demo!

stackoverflow.cpp
memleak.cpp

Komposite typer

- * C har structer:

```
struct student{  
    int personnr;  
    string name;  
}; ...  
student s;  
s.personnr=10;  
s.name="Alfred";
```

- * C++ har klasser: Structer er klasser der alle medlemmer er public
- * Structer kan ha metoder i C++, men ikke i C
- * Structer og klasser er også typer; instansene er objekter.
- * OBS: Legg merke til at "new" ikke er nødvendig for å instansiere!

Egendefinerte typer

- * `typedef` lar deg lage et "alias" for en annen type
- * Feks: `typedef time_t long;` (i `clock.h`)
- * Hvorfor?
 - * Hvis man får lyst til å endre den underliggende typen, trenger man kun endre det ett sted
 - * ...forutsatt at alle operasjoner der typen inngår fortsatt er gyldige
 - * Moteksempler?

Konstanter

- * Kodeordet **const** lar oss definere en variabel, som ikke kan endres.
 - * Hvordan får den verdi da?
 - * Initialisering
- * **const int i=5; //OK**
i++; //Err.
- * Men hva med **const int* iptr=&i;**
- * Her er bare verdien konstant. Adressen kan endres.
- * Tenk "const int"-pointer
- * Løsning: **const int* const iptr=&i.**

Referanser

- * `int&` i er en referanse til en integer
- * Referanser er “automatically dereferenced immutable pointers”
 - * Immutable betyr “const”. Her er det adressen som er “const”.
- * `int i=10;`
`int& j=i; //Tilordning “by value”`
`j++ //i inkrementeres`
- * `const &int j=i; //Nå kan ikke i endres via j`
- * Når er dette nyttig? “Pass by reference”.

Parameteroverføring

- * Som "Default" overføres funksjonsargumenter "By value".
 - * Gjelder dette pekere?
- * For store objekter er dette tungt. Bedre:
`void myFunc (bigObject& obj)`
`void myFunc (const bigObject& obj)`
- * Kalles med "bigObject obj; myFunc(obj)"
 - * myFunc kan endre "obj" hvis ikke const
 - * Ulemper?

Parameteroverføring: Bjarne anbefaler

- * Pass by value, når det er lite data
- * Pass by const reference ellers, hvis du kan
- * Pass by reference, hvis funksjonen skal endre variablen
- * Pass by pointer, kun hvis du må få høyde for at pekeren kan være 0.
- * Mer i oppgavene og på onsdag!