Effektiv Kode med C og C++

Forelesning 4, vår 2015

Alfred Bratterud

Agenda:

* Oblig 1!
* Litt repetisjon
  * Minne
  * Objekters levetid
* Klasser
* Initialisering og "default constructor"
* Destructor og memory leaks

Oblig 1

* Oblig 1 kommer på github i dag. Den har frist:
  10.feb kl. 23:59
* Levering via github:
  * Mappen 'oblig1' - kan være tom, men må ligge i ditt PRIVATE github-repo, senest om 1 uke
    * Link må være levert på fronter også da
* 2 deloppgaver:
  * oppvarming - datatyper og cæsar-kryptering
  * Blackjack!
* Du skal ikke ha noe GUI - kun terminal!
* Obligen skal løses individuelt

Litt repetisjon

Tre typer minne «Storage classes»

Fra boken §A.4.2 og §A.4.3 (Language summary)

* «Automatic storage»: «Local scope» + «statement scope»
  * > Allokeres - av ditt program - når CPU'en «går inn i deres skop».
  * < Frigjøres automatisk - av ditt program - når de går ut av skop.
    * Kompilator bygger det inn i din kode!
* «Static storage»: data i namespace- og globalt skop + «static»
  * > Allokeres (av OS'et) når prosessen starter
  * < Frigjøres (av OS'et) når prosessen avslutter
* Dynamic storage, "Free store (heap)": data i RAM du styrer
  * > Allokeres når du bruker ordet «new»
  * < Blir borte når DU gir beskjed (evt. når prosessen avslutter)

Wrap-up om datatyper

Hva er en datatype?

* Én av flere måter å lagre data i minne på
* Et antall bytes sett som en helhet
  - Primitive typer
* En sammensetting av primitive typer, sett som en helhet
  - Kompositte typer
* I tillegg til antall bytes har datatypene en del «implisitt semantikk»:
  * Operatorer har forskjellig oppførsel basert på typen(e) til parametrene
  * x + y ?
  * cout << x << endl; ?
  * Avhenger av typene til x og y!

Initialisering

- Initialisering er å gi en startverdi til en variabel
- Initialiserer man ikke, er innholdet "udefinert"
  - `int i; i=10;` // IKKE initialisering
  - `int i=10;` //OK
  - `int i(10);` //OK - default copy-constructor
  - `int i{10};` //C++11 - gir flere advarsler -best!
- Initialisering er rett frem for "vanlige typer", litt mer komplekst for medlemmer, ille for store arrayer.

Egendefinerte typer

* Structer og klasser er egendefinerte typer
* I tillegg: typedef lar deg lage et "alias" for en annen type
* Feks: typedef long time_t; (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?
    * ...Det kan bli vanskelig for utviklere å skjønne typene

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 "Derfault" 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

1. Pass by value, når det er lite nok data til at det er mulig.
2. Pass by const reference ellers, hvis du kan
3. Pass by reference, hvis funksjonen skal endre variablen
4. Pass by pointer, kun hvis du må ta høyde for at pekeren kan være 0.

* TIPS: Sjekk da ut shared_ptr og unique_ptr fra C++11 først! (Men, du må skjønne skop dynamisk minne skikkelig)

Kompositte typer og klasser

Objektorientering i C++

Kompositte typer

* C har structer:

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

* Structer er klasser der alle medlemmer er public - «Plain Old Data»(POD)
* 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!

Klasser

- Klasser er structer med medlemsfunksjoner, der medlemsvariabler er "private" som standard. Dette heter "enkapsulering"
- Medlemmer kan være av alle typer - inkludert egne
- Syntaksen er rett frem:

```
class student{
  int nr;
  string name;
  public:
    int get_nr();
    string get_name();
}
```

- Nå er studentklassen "read-only"
- Men hvor er "kroppene" til funksjonene?
  - Implementasjon (.cpp) og header-filer (.hpp / .h) legges hver for seg pga. gjenbruk ved linking
  - Header-filene blir som «interface» i java - et pent grensesnitt

Default konstruktorer

* En konstruktor er en medlemsfunksjon med samme navn som klassen
* Med C++ har også primitive typer en standard konstruktor, så `int()` er gyldig, også `int(5)`.
* En default konstruktor blir opprettet for deg, og tar ingen argumenter
* I egne konstruktorer er det egen notasjon for å initialisere medlemmer. Litt lettere i C++11.
* Man kan alltid ha flere konstruktorer, men bare i C++11 kan en konstruktor kalle en annen.

konstruktorer i klasser

* Konstruktoren har som jobb å initialisere alle medlemmer - ved å kalle deres konstruktorer.
* OBS: Lager du en konstruktor selv, uten argumenter, mister du default konstruktoren. Helt OK - men da har du ansvaret!
* I alle egne konstruktorer må alle medlemmer initialiseres eksplisitt - evt. ved å kalle standard-konstruktor.
* Man må da initialisere slik:

```
class student{
  int nr;
  string name_;
  public:
     student(string n) : nr{10}, name_{n} { ... }
     int get_nr();
     string name();
};
```

Instansiering av klasser

```
class Student{
  int _nr;
  string _name;

public:
  Student(string _n) : _name{_n} {}
  Student() : _name{"John Doe"}, _nr{42} {}

  string name(){ return _name; };

};

int main(){

  // Hvilken initialisering er riktig?

  Student s1;
  Student s2{"Alfred"};
  Student s3 = new Student("Alfred");
  Student s4 = Student("Alfred");
  Student* s5{new Student{"Alfred"}};
```

Instansiering av klasser

OK: Tom konstruktør og «automatic memory»

```
class Student{
  int _nr;
  string _name;

public:
  Student(string _n) : _name{_n} {}
  Student() : _name{"John Doe"}, _nr{42} {}

  string name(){ return _name; };

};

int main(){

  // Hvilken initialisering er riktig?

  Student s1;
  Student s2{"Alfred"};
  Student s3 = new Student("Alfred");
  Student s4 = Student("Alfred");
  Student* s5{new Student{"Alfred"}};
```

Instansiering av klasser

OK: Tom konstruktør og «automatic memory»

OK: Konstruktør m. String og «automatic memory»

```
class Student{
  int _nr;
  string _name;

public:
  Student(string _n) : _name{_n} {}
  Student() : _name{"John Doe"}, _nr{42} {}

  string name(){ return _name; };

};

int main(){

  // Hvilken initialisering er riktig?

  Student s1;
  Student s2{"Alfred"};
  Student s3 = new Student("Alfred");
  Student s4 = Student("Alfred");
  Student* s5{new Student{"Alfred"}};
```

Instansiering av klasser

OK: Tom konstruktør og «automatic memory»

OK: Konstruktør m. String og «automatic memory»

IKKE OK: new returnerer peker. Ellers OK.

```
class Student{
  int _nr;
  string _name;

public:
  Student(string _n) : _name{_n} {}
  Student() : _name{"John Doe"}, _nr{42} {}

  string name(){ return _name; };

};

int main(){

  // Hvilken initialisering er riktig?

  Student s1;
  Student s2{"Alfred"};
  Student s3 = new Student("Alfred");
  Student s4 = Student("Alfred");
  Student* s5{new Student{"Alfred"}};
```

Instansiering av klasser

OK: Tom konstruktør og «automatic memory»

OK: Konstruktør m. String og «automatic memory»

IKKE OK: new returnerer peker. Ellers OK.

OK: Konstruktør m. String og «automatic memory»

```
class Student{
  int _nr;
  string _name;

public:
  Student(string _n) : _name{_n} {}
  Student() : _name{"John Doe"}, _nr{42} {}

  string name(){ return _name; };

};

int main(){

  // Hvilken initialisering er riktig?

  Student s1;
  Student s2{"Alfred"};
  Student s3 = new Student("Alfred");
  Student s4 = Student("Alfred");
  Student* s5{new Student{"Alfred"}};
```

Instansiering av klasser

OK: Tom konstruktør og «automatic memory»

OK: Konstruktør m. String og «automatic memory»

IKKE OK: new returnerer peker. Ellers OK.

OK: Konstruktør m. String og «automatic memory»

OK: Konstruktør m. String og «dynamic memory»

```
class Student{
  int _nr;
  string _name;

public:
  Student(string _n) : _name{_n} {}
  Student() : _name{"John Doe"}, _nr{42} {}

  string name(){ return _name; };

};

int main(){

  // Hvilken initialisering er riktig?

  Student s1;
  Student s2{"Alfred"};
  Student s3 = new Student("Alfred");
  Student s4 = Student("Alfred");
  Student* s5{new Student{"Alfred"}};
```

Destruktorer

* Destruktor har ansvar for å rydde opp, dvs. frigjøre det minnet / evt. andre ressurser klassen har satt av.
* Destruktoren til en klasse "myClass" heter " ~myClass()
* Vi kan lage egne, som vanlig, men alle objekter har en "default destructor"
* ....Den kaller destructoren til medlemmene. Default destructor for peker?
* Hva vil vi typisk gjøre i en "destructor"?
    * Frigjøre alt klassen allokerte med new!
    * Evt. lukke filer!
* Fantastisk garanti: Destructor kalles alltid når variabel går ut av skop.
* Hvorfor har vi ikke destructorer i java/php?
    * Alle problemene i C++ skyldes pekere ;-)

Interface v.s. Implementation

* "Interfacet" til en klasse (eller et bibliotek) består av deklarasjoner av alle medlemmene, men kun med "signaturene" til funksjonene
    * Denne ligger gjerne i en egen "header-fil" (class_student.h)
* "Implementasjonen" ligger gjerne i en annen fil (class_student.cpp), som "inkluderer" header-fila
* Hvorfor?
* Fordi vi da kan «interface» med ferdig kompilerte klasser, kun ved å kjenne til header fila
* Dette krever «linking» mellom din binærfil og den ferdig kompilerte klassen
    * Gjøres av «linkeren» (GNU ld i Linux-vm'en)
    * Kan gjøres «Statisk» eller «dynamisk» - til og med «run time»
    * I Windows: .dll'er er delte, ferdigkompilerte biblioteker. I Linux: .so

Nå: Demo!

"construct_destruct.cpp"
+ memory leaks