

Norges teknisk-naturvitenskapelige
universitet
Institutt for datateknologi og
informatikk

TDT4102 Prosedyre-
og objektorientert
programmering
Vår 2024

Øving 11

Frist: 2024-04-19

Mål for denne øvingen:

- Lære å måle ytelsen til et program
- Benytte ulike strategier for å optimalisere et program
- Lære om templates og bruke disse til å gjøre funksjoner og klasser mer generelle
- Bruke beholderne som finnes i standardbiblioteket
- Implementere egne utgaver av noen av disse beholderne

Generelle krav:

- Bruk de eksakte navn og spesifikasjoner gitt i oppgaven.
- Teorioppgaver besvares med kommentarer i kildekoden slik at læringsassistenten enkelt finner svaret ved godkjenning.
- Denne øvingen skal implementeres uten hjelp fra `std_lib_facilities.h`.
- 70% av øvingen må godkjennes for at den skal vurderes som bestått.
- Øvingen skal godkjennes av stud.ass. på sal.
- Det anbefales å benytte en programmeringsomgivelse(IDE) slik som Visual Studio Code.

Anbefalt lesestoff:

- Kapittel 19.2 og 19.3 i PPP.
- Beskrivelsen av beholderne i standardbiblioteket i boka og ekstra detaljer på [cppreference.com](https://en.cppreference.com)

I denne øvingen skal vi ikke bruke `std_lib_facilities.h` så du må passe på å inkludere eventuelle biblioteker du trenger og holde orden på navnerom selv.

1 Optimalisering: Måle ytelse (15%)

Optimalisering handler om å få et program til å kjøre mer effektivt, uten å endre på funksjonalitet. Dette er viktig for programmer som krever høy ytelse, som for eksempel dataspill. En viktig grunn til at man velger å skrive i C++ er at det er svært effektivt, men selv C++ kode kan optimaliseres. Optimalisering er ikke dekket av læreboka, se slides og eksempler fra forelesning.

For å optimalisere kode må vi kunne finne ut om en endring vi gjør faktisk fører til at koden kjører raskere. Den eneste måten vi kan vite det på er å måle tiden programmet bruker på å kjøre, før og etter endringen. I denne oppgaven skal vi se på en metode for å måle ytelse. Dette er ikke alltid så enkelt da mange operasjoner tar mindre enn et nanosekund, og klokker i datamaskiner er ofte ikke nøyaktige nok til å måle dette. I tillegg vil operativsystemet og andre prosesser som kjører i bakgrunnen føre til støy som påvirker resultatet. For å klare å måle ytelsen til små programmer kan vi gjenta programmet mange ganger og dele totaltiden på antall repetisjoner.

a) **Lag et program som kan måle ytelsen til korte operasjoner.** Dette gjøres ved å lage en løkke som gjentar operasjonen mange ganger. Ta tiden på kjøringen av hele løkka og del totaltiden på antall iterasjoner for å finne gjennomsnittlig kjøretid for operasjonen. Benytt klassen `Stopwatch` som ligger i den utdelte koden. Lag et `Stopwatch` objekt og kall på medlemsfunksjonen `start()` og `stop()` for å ta tiden på programmet.

b) **Mål kjøretiden til ulike operasjoner.** Benytt programmet du lagde i oppgave a).

- Mål tiden det tar å initialisere en `std::unique_ptr` og en `std::shared_ptr`.
- Sammenlign tiden det tar å allokere minne på stack-en vs. på heap-en. Dette kan du for eksempel gjøre ved å ta tiden på hver av kodene under.

```
#include <array>
constexpr int size = 10000;

// Stack allocation
std::array<int, size> arr1;

//Heap allocation
std::array<int, size>* arr2 = new std::array<int, size>;
delete arr2;
```

Tips: Prøv deg frem med ulikt antall kjøring. Dersom programet ikke blir ferdig inne 20 sekunder kan du stoppe det og prøve med litt færre repetisjoner. Dersom du får ut 0 sekunder må du øke antall repetisjoner. Prøv også å kjøre det samme programmet flere ganger for å se på mengden støy i målingene.

2 Optimalisering av `std::vector` (20%)

I denne oppgaven skal vi se på en metode for å optimalisere standardbiblioteket sin mest brukte beholder `std::vector`. Da er det først viktig å skjønne hvordan den fungerer.

- Undersøk hvordan `std::vector` er implementert.** `std::vector` benytter seg av et dynamisk allokert `array` for å lagre data. Kapasiteten til et `array` kan ikke endres, men med en `std::vector` kan vi legge til så mange elementer vi vil etter vi har opprettet det. Undersøk hvordan dette er implementert ved å først opprette en tom `vector`, og deretter legge til tallene 0-19 ved hjelp av en løkke. For hver iterasjon skriver du ut lengden til vektoren med medlemsfunksjonen `size()`, samt kapasiteten til vektorens `array` med `capacity()`. Hva skjer når vektorens lengde overskrider kapasiteten til `array`et?
- Det tar tid å allokere minne og kopiere data. Ved å reservere plass i minne til alle elementene vi har tenkt til å legge inn i vektoren slipper vi å bruke unødvendig tid på minneallokasjon. **Kall på medlemsfunksjonen `reserve(20)` før du legger inn elementene.** Hva skjer nå med `size()` og `capacity()`?
- Hva skjer hvis vi istedenfor benytter medlemsfunksjonen `resize(20)`?** Skriv ut størrelsen og kapasiteten til vektoren før du legger inn noen elementer. Må du endre noe på innholdet i løkka for å få samme vektor som i a)?
- Mål tiden det tar å legge inn 1 000 000 elementer i en `std::vector`.** Benytt medlemsfunksjonen `push_back()`. Her er det nok å ta tiden på en kjøring.
- Optimaliser operasjonen fra forrige deloppgave.** Prøv å optimaliser både ved å bruke `reserve()` og ved å bruke `resize()`. Hvilken går raskest?

Nyttig å vite: Optimaliseringsstrategier

Algoritmeoptimalisering

Å benytte en mer effektiv algoritme kan gi en betydelig økning i ytelsen til et program. Denne typen optimalisering går ut på å redusere unødvendige beregninger og minnebruk. Det finnes for eksempel mange ulike sorterings algoritmer som kan gi ulik effektivitet både med tanke på tidsbruk og minnebruk.

Minneoptimalisering

Det tar tid å allokere minne, derfor kan man optimalisere ved å redusere minne allokering og kopiering av objekter. Det tar også lengere tid å allokere minne på `heap`-en enn på `stack`-en, derfor bør man holde seg til allokering på `stack`-en der det er mulig.

Cache-optimalisering

Cache er et lite, men ekstremt raskt minne i prosessoren som effektiviserer dataaksessering. Det er vesentlig raskere å hente data fra cache enn fra RAM (hovedminnet). Derfor kopieres data som er sannsynlig at vil bli aksessert, over på cache. Dette gjøres enkelt forklart ved at hver gang man henter ut data fra en minneadresse i RAM vil en hel blokk med minne, som ligger rundt den aksesserte minneadressen, bli kopiert over på cache. Dette gjør at data som ligger i nærheten av minneadressen du har aksessert, og data du nylig har aksessert vil ligge raskt tilgjengelig på cache.

Vi kan optimalisere programmet vårt ved å skrive kode som best mulig utnytter cache (kalles ofte "cache friendly code"). Dette kan for eksempel gjøres ved å:

- Benytte beholdere som lagrer data lineært etter hverandre, som `array` og `vector`.

- Lage array med verdier og ikke array med pekere til verdier, siden disse ikke vil ligge på samme sted i minne.
- Iterere gjennom beholdere i den rekkefølgen de er definert.

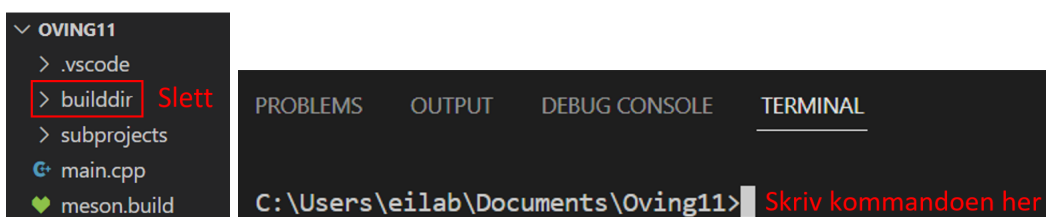
Kompilatoroptimalisering

Kompilatoren kan optimalisere kode når den bygger programmet. Den kan få det ferdig bygde programmet til å kjøre betydelig fortere ved å optimalisere algoritmene, og fjerne unødvendige kode. I faget til nå har vi benyttet oss av debug mode når vi kompilerer programmet. Denne har ikke fokus på å optimalisere, og legger faktisk til ekstra informasjon slik at det skal være lettest mulig å debugge. Det gjør at man kan bruke debuggeren og man får mer forklarende feilmeldinger med hvor og hva feilen er. Derfor brukes debug mode under utviklingen av et program. Når programmet er ferdig skrevet og alt kjører som det skal, kan man bytte over til release mode som aktiverer optimalisering.

3 Optimalisering av utdelt kode (25%)

I denne oppgaven skal du optimalisere et lite program som ligger i den utdelte filen *optimizationTask.cpp*. For å kjøre programmet må funksjonen `optimizationTask()` kalles fra `main()`.

- Ta tiden på hele programmet før optimalisering.** Noter ned tiden slik at du kan sammenligne med tiden etter optimalisering.
- Optimaliser koden så mye du klarer.** Skriv en liten kommentar til hver av endringene du gjør om hvorfor du gjør de. Prøv å skjønne hvorfor endringen gjør at koden kjører raskere.
- Skru på kompilatoroptimalisering.** Dette aktiveres ved å bytte til release mode. På oppsettet vårt kan dette gjøres ved å slette mappen som heter `builddir` og deretter skriv følgende kommando i terminalen og trykk Enter
`meson setup builddir -Dbuildtype=release`



Dersom du har gjort dette riktig skal det blant annet skrives dette i terminaler

```
User defined options
buildtype      : release
```

For å endre tilbake til debug mode, slett `builddir` mappen igjen og kjør kommandoen:
`meson setup builddir -Dbuildtype=debug`

Nyttig å vite: Template

Noen ganger ønsker vi at en funksjon skal fungere for flere datatyper. Hittil har vi gjort dette ved å overlaste funksjonen. For eksempel kan man definere funksjonen `add()` for både `int` og `double`.

```
int add(int lhs, int rhs){
    return lhs + rhs;
}
double add(double lhs, double rhs){
    return lhs + rhs;
}
int main() {
    add(1, 2); // Gir 3
    add(1.0, 2.0); // Gir 3.0
}
```

Du legger kanskje merke til at definisjonen av de to funksjonene er helt like. Eneste forskjellen er typene til parameterne og returverdien. Med templates kan disse kombineres, og man slipper duplisering av koden.

```
template<typename T>
T add(T lhs, T rhs){
    return lhs + rhs;
}
int main() {
    add(1, 2); // Gir 3
    add(1.1, 2.2); // Gir 3.3
    add(1, 2.2) // Feil, parameterne må ha samme datatype

    // Eksplisitt template-argument
    add<int>(1, 2); // Gir 3
    add<double>(1.1, 2.2); // Gir 3.3
    add<int>(1, 2.2); // Gir 3, parameterne konverteres til int
    add<double>(1, 2.2); // Gir 3.2, parameterne konverteres til double
    return 0;
}
```

Dere er allerede kjent med å bruke templates og syntaksen for å spesifisere template-argument fra bruk av `std::vector`.

```
std::vector<int> i; // int er template-argument, gir en vektor med heltall
std::vector<double> d; // double er template-argument, gir en vektor med flyttall
```

Templates kan også benyttes til å generalisere klasser. Eksempler på dette finner man blant annet i standardbiblioteket: `std::vector`, `std::set` og alle de andre beholderne i standardbiblioteket er implementert ved hjelp av templates.

Klasse-templates defineres slik:

```
template<typename T, typename U>
struct Pair{
    T first;
    U second;
};
```

Her har vi definert en type med to medlemsvariabler som kan være av hvilken som helst type. Dette er slik `std::pair` klassen er implementert, som brukes i blant annet `std::map`.

4 Templates for funksjoner (20%)

Templates er en form for *generisk programmering* som lar oss skrive generelle funksjoner som fungerer for mer enn én datatype uten å tvinge oss til å lage separate implementasjoner for hver enkelt datatype. I denne oppgaven skal vi skrive noen slike. Templates er beskrevet i kapittel 19.3 i boken.

- a) **Skriv template-funksjonen `maximum` som tar inn to verdier av samme type som argument og returnerer den største verdien av de to.** Funksjonen skal være skrevet slik at følgende kode skal compilere uten feil og gi forventede resultater ved kjøring.

```
int a = 1;
int b = 2;
int c = maximum<int>(a, b);
// c er nå 2.

double d = 2.4;
double e = 3.2;
double f = maximum<double>(d,e);
// f er nå 3.2
```

Denne funksjonen vil fungere for alle grunnleggende datatyper (for eksempel `int`, `char` og `double`), men hvis du bruker argumenter av en egendefinert type, som en `Person`- eller `Circle`-klasse, er sjansen stor for at koden din ikke vil compilere. Hvorfor? Hva kreves av objektene for at denne funksjonen skal kunne brukes?

- b) **Skriv template-funksjonen `shuffle` som stokker om på elementene i en `vector` slik at rekkefølgen på elementene i `vector`-en blir tilfeldig.** Du kan benytte `std::swap` for å bytte plass på to elementer.

Funksjonen skal være skrevet slik at følgende kode skal compilere uten feil og gi forventede resultater ved kjøring.

```
std::vector<int> a{1, 2, 3, 4, 5, 6, 7};
shuffle(a); // Resultat, rekkefølgen i a er endret.

std::vector<double> b{1.2, 2.2, 3.2, 4.2};
shuffle(b); // Resultat, rekkefølgen i b er endret.

std::vector<std::string> c{"one", "two", "three", "four"};
shuffle(c); // Resultat, rekkefølgen i c er endret.
```

5 Templates for klasser (20%)

I denne oppgaven skal vi lage vår egen versjon av template beholderen `std::array`. Når vi skal lage en klasse som benytter templates er det viktig at all koden for klassen, inkludert implementasjonen av medlemsfunksjonene, begynner seg i en headerfil. Dette er fordi kompilatoren

må kjenne til hele klassen (både deklarasjon og implementasjon) for å kunne generere riktig spesialisering av klassen hver gang den benyttes.

a) **Deklarer en klasse `MyArray` som kan holde et vilkårlig antall elementer av vilkårlig datatype.** Klassen skal ha to template parametere, en `typename Type` og en `int Size`. Den skal ha en privat medlemsvariabel `elements` som skal være et C-array som kan holde `Size` elementer av typen `Type`.

b) **Definer følgende medlemsfunksjoner**

- `getSize()` som returnerer størrelsen til arrayet
- `at()` som tar inn en indeks, og sjekker om den ligger i arrayet. Dersom den gjør det skal funksjonen returnere en referanse til elementet på den plassen. Hvis ikke skal det kastes et beskrivende unntak.
- `fill()` som tar inn en verdi og fyller hele arrayet med den verdien.

Du kan godt definere funksjonene i klasse scope. Dersom du ønsker å skrive definisjonen utenfor klasse-scope må du igjen spesifisere template argumentene slik:

```
template<typename Type, int Size>
void MyArray<Type, Size>::fill(){
    // implementasjon
}
```

Husk at definisjonen også må ligge i `.h` filen når vi benytter templates.

c) **Test klassen `MyArray`.** Opprett flere `MyArray` objekter med ulik lengde og datatype. Bruk medlemsfunksjonene til å fylle beholderne med elementer og skriv dem ut til konsollen.

Prøv også å aksessere elementer som ligger både inni og utenfor beholderen. Plasser koden som kan kaste unntak i en `try`-blokk, og lag en tilhørende `catch`-blokk som håndterer og informerer brukeren om feilen.