Avdeling for informatikk og e-læring, Høgskolen i Sør-Trøndelag

4. Kontainere, auto og anonyme funksjoner

Else Lervik, Ole Christian Eidheim

Lærestoffet er utviklet for faget IFUD1048 C++ for programmerere
med grunnlag i lærestoff fra LV195D Objektorientert programmering i C++

4. Kontainere, auto og anonyme funksjoner

Resymé: STL er C++ sitt bibliotek for å håndtere data-kontainere. I denne leksjonen konsentrerer vi oss om vektorer som er den mest brukte kontaineren. Andre kontainere brukes omtrent på samme måte. Bruken avhenger imidlertid til en viss grad av hva vi putter inn i kontaineren. Vi ser på to eksempler: data av en primitiv datatype (tall), og objekter av en klasse (klassen Flate fra en tidligere leksjon). Vi skal også se på STL algoritmer, nøkkelordet auto, og anonyme funksjoner.

Innhold

1.1 HVA ER STL? 1
1.2 STL-DOKUMENTASJON 2
1.3 KONTAINEREN VEKTOR 2
1.3.1 En vektor fylt med tall 2
1.3.2 En vektor fylt med objekter 3
1.3.3 Iterasjon av vektor elementene, og nøkkelordet auto 4
1.4 ITERATORER 5
1.5 ALGORITMER OG ANONYME FUNKSJONER 5
1.5.1 Algoritmen sort 5
1.5.2 Anonyme funksjoner 6

P. Deitel & H. Deitel: C++ 11 for Programmers, 2Ed.: vector template, kap. 7.10. Generelt om STL-kontainere og iteratorer, se kap. 15.1-15.5.1.

1.1 Hva er STL?

Standard Template Library (STL) er en del av C++ sitt standardbibliotek. Som navnet sier er det et template bibliotek, det vil si at du angir datatypen når du skal bruke de enkelte delene av biblioteket.

Biblioteket består av tre deler:

Kontainere

Dette er beholdere for data. Typen container sier hvordan data er organisert. Eksempler er de kjente datastrukturer som vektor (dynamisk tabell), sett og kart (assosiativ tabell). Du finner en oversikt over STL-kontainerne i boka side 476-477. På engelsk er de viktigste kontainerne som følger: vector, unordered_set og unordered_map. Disse tre er pensum, og etter du har lest gjennom denne leksjonen, kan du se på eksempel bruk av unordered_set og unordered_map her: http://www.cplusplus.com/reference/unordered_set/unordered_set/find/ og http://www.cplusplus.com/reference/unordered_map/unordered_map/find/.

En vektor (tenk ikke matematikk!) ligner på en tabell, men den er laget slik at vi ikke trenger å tenke på at den kan bli overfylt. Dersom den blir full, utvider den seg selv. Vi skal bruke vektorer både på enkle tall og på objekter.

Java-programmerere:

C++ **vector** svarer til Java-klassen **ArrayList** (og **Vector**). Men i Java blir kjøretiden mye dårligere når en går fra en lavnivå tabell til ArrayList. I C++ er det ingen forskjell om en bruker en lavnivå tabell eller vector. Bruk derfor vector i stedet for en tabell når du programmerer i C++.

Iteratorer

En iterator lar oss forflytte oss fra element til element i kontaineren.

Algoritmer

En algoritme gjør noe med datainnholdet i en kontainer. Vi har algoritmer for sortering, søking, kopiering, med mer. Algoritmene er i utgangspunktet fristilt fra kontainerne. Iteratorene danner bindeleddet mellom kontainere og algoritmer. Vi sender inn iteratorer som argumenter til algoritmene. Dermed vet algoritmen hvilke data den skal jobbe med.

1.2 STL-dokumentasjon

Læreboka viser en del (Deitel & Deitel, kap. 7, 15-16). Det fins også mye på Internett, se for eksempel http://www.cplusplus.com/reference/stl/.

1.3 Kontaineren vektor

1.3.1 En vektor fylt med tall

Følgende program lager en vektor, fyller den opp med positive tall lest inn fra brukeren, sorterer dem og skriver dem ut.

```
// Eksempel1.cpp,  se vedlagt zip-fil
#include <iostream>
#include <vector>   // header-fil for vector
#include <algorithm>  // header-fil for algoritmer

using namespace  std;

int main() {
  vector<int> posTall;  // oppretter en vektor av heltall
  int tall;
  cout << "Skriv positive tall (avslutt med 0): ";
```

```
  cin >> tall;
  while (tall > 0) {
    posTall.emplace_back(tall); // legger inn det nye tallet bakerst
    cin >> tall;
  }
  cout << "Du har skrevet " << posTall.size() << " tall.\n";
  for (size_t i = 0; i < posTall.size(); i++) {
    cout << posTall[i] << " ";  // henter ut et og et tall ved indeksering
  }
  cout << endl;
  return 0;
}
```

Medlemsfunksjonen **emplace_back()** legger inn et tall bakerst i vektoren. Vektorer fungerer mest effektivt hvis vi kan legge inn nye data bakerst.

Det fins også en **emplace()**-funksjon som kan brukes. Den bruker iteratorer for å fortelle hvor det nye elementet skal plasseres. Dette er en tyngre funksjon enn **emplace_back()**. (Se øvingen.)

size() (*størrelsen*) gir oss antall tall lagt inn i vektoren. Vektoren er implementert som en tabell, og lagringsplassene ligger ved siden av hverandre i minnet. Vektoren har til enhver tid en *kapasitet* som er større eller lik **size()**. Kapasiteten begynner med 1, deretter utvides den ved behov, og det i større og større bolker (for eksempel dobling, men det er kompilatoravhengig).

Du kan finne vektorens kapasitet ved å sende meldingen **capacity()** til **posTall**-objektet. Det fins funksjoner som lar brukeren bestemme kapasiteten og kapasitetsøkningen.

Også for vektorer er []-operatoren overloadet. Vi kan bruke den både på venstre og på høyre side av tilordningstegnet. På venstre side må du imidlertid bare bruke indeksering for elementer som allerede er lagt inn i vektoren, eksempel (jamfør **string**-objekter, der det samme gjelder):

```
for (size_t i = 0; i < posTall.size(); ++i) {
  posTall[i] += 100;
}
```

Du kan altså ikke bruke indeksering til å legge inn nye elementer bakerst i vektoren.

1.3.2 En vektor fylt med objekter

Vi skal legge **Flate**-objekter inn i vektoren. Koden kan se slik ut:

```
#include <iostream>
#include <vector>

using namespace std;

class Flate {
public:
  string navn;
  double lengde;
  double bredde;
```

```
  Flate(const string &startNavn, double startLengde, double startBredde) :
navn(startNavn), lengde(startLengde), bredde(startBredde) {}

  double finnAreal() const {
    return lengde * bredde;
  }
};

int main() {
  vector<Flate> flater;
  flater.emplace_back("aaa", 3, 3);
  flater.emplace_back("bbb", 1, 1);
  flater.emplace_back("ccc", 2, 2);

  cout << "Antall flater: " << flater.size() << endl;

  for (size_t i = 0; i < flater.size(); ++i) {
    cout << flater[i].navn << " areal: " << flater[i].finnAreal() << endl;
  }
}

/* Kjøring:
Antall flater: 3
aaa areal: 9
bbb areal: 1
ccc areal: 4
*/
```

Her er både definisjonen og implementasjonen av klassen **Flate** lagt i samme fil som main-funksjonen for å gjøre eksempelet enklere. I tillegg har vi gått bort i fra bruk av enkle hent- og sett-funksjoner, og i stedet satt datamedlemmene som public. Ofte er dette greit, og samtidig fører det til at klassene er enklere å lese. Derimot er ikke dette alltid lurt å gjøre, og en bør tenke seg om før en velger å lage en klasse slik. **Dere kan likevel trygt skrive øvinger og levere eksamen med denne forenklet måten å skrive klasser på**. Men om dere skal skrive et bibliotek eller et større program senere, etter dette kurset, så kan det være lurt å dele opp klassene (.h/.hpp og .cpp filer) og være ekstra forsiktig med hvilke datamedlemmer dere har som public.

Vektor-metoden emplace_back tar konstruktør-parameterne til Flate som parametere. Dette gjør at vi slipper å skrive for eksempel `flater.emplace_back(Flate(”aaa”, 3, 3))`.

1.3.3 Iterasjon av vektor elementene, og nøkkelordet auto

Å gå gjennom elementene i en vektor kan gjøres på flere måter. Vi har allerede sett denne måten:

```
for (size_t i = 0; i < flater.size(); ++i) {
  cout << flater[i].navn << " areal: " << flater[i].finnAreal() << endl;
}
```

En kan også gjøre det slik:

```
for (Flate &flate : flater) {
  cout << flate.navn << " areal: " << flate.finnAreal() << endl;
}
```

Faktisk så kan vi gjøre løkken over enda litt enklere ved å bruke nøkkelordet **auto**. Vi lar da kompilatoren finne ut hvilken klasse objektet flate tilhører:

```
for (auto &flate : flater) {
  cout << flate.navn << " areal: " << flate.finnAreal() << endl;
}
```

En kan bruke **auto** også i andre tilfeller, for eksempel:

```
auto areal = flate.finnAreal();
```

Variabelen **areal** blir her satt til å være samme typen som **finnAreal()** returnerer. Bruken av **auto** gjør at en ofte slipper å skrive datatypen mer enn en gang.

Vi kommer til å se på enda en måte å gå gjennom elementene i en vektor når vi kommer til iteratorer.

1.4 Iteratorer

Iteratorer fungerer først og fremst som et bindeledd mellom kontainerne og algoritmene. (Nesten) alle kontainere har medlemsfunksjonene **begin()** og **end()**. Funksjonen **begin()** returnerer en iterator til første element i kontaineren, **end()** returnerer en iterator til elementet etter det siste(!). Dette gjør det mulig å skrive løkker slik:

```
for (auto iterator = flater.begin(); iterator != flater.end(); ++iterator)
```

Dersom vi ønsker å iterere bare en del av vektoren skriver vi for eksempel:

```
for (auto iterator = flater.begin() + 2; iterator != flater.end() - 1; ++iterator)
```

Her går vi fra og med element med indeks 2, og opp til og med det nest siste elementet.

En iterator fungerer som en slags peker til de enkelte elementene i en kontainer. Vi kan for eksempel bruke dereferanseoperatoren (*) til å hente ut det aktuelle elementet.

Følgende løkke løper gjennom flate-vektoren fra eksemplet foran, og skriver ut hvert enkelt element:

```
for (auto it = flater.begin(); it != flater.end(); ++it)
  cout << (*it).navn << " areal: " << (*it).finnAreal() << endl;
```

I stedet for å skrive (*it).navn kan vi bruke operatoren -> slik:

```
for (auto it = flater.begin(); it != flater.end(); ++it)
  cout << it->navn << " areal: " << it->finnAreal() << endl;
```

Vi bruker forresten ++it i stedet for it++. Grunnen til dette er at it++ returnerer den gamle verdien til **it** i tillegg til at **it** blir økt. Kompilatoren må da passe på å ha to verdier av **it** lagret; den gamle og den nye verdien Når vi skriver ++it så trenger kompilatoren kun en verdi av iteratoren **it**, altså den nye verdien av **it** blir returnert. C++ burde egentlig hete ++C!

1.5 Algoritmer og anonyme funksjoner

1.5.1 Algoritmen sort

Som nevnt tidligere i leksjonen, så er algoritmer en del av STL. Det finnes mange slike STL-funksjoner. Noen av dem er tema i en senere leksjon. Før det, skal vi se nærmere på STL-funksjonen **sort**:

```
#include <iostream>
#include <vector>
```

```
using namespace std;

int main() {
  vector<int> tall;
  tall.emplace_back(3);
  tall.emplace_back(1);
  tall.emplace_back(2);
  sort(tall.begin(), tall.end());
  for (auto &et_tall : tall)
    cout << et_tall << endl;
}
/* Kjøring:
1
2
3
*/
```

På grunn av at algoritmen **sort** tar iteratorer som parametere, kan algoritmen brukes på flere ulike kontainerne på lignende måte.

Vi kan også bruke algoritmen på objektet flater av typen vector<Flate> på følgende måte:

```
...
bool sammenlign(const Flate &a, const Flate &b) {
  return a.finnAreal() < b.finnAreal();
}

int main() {
  ...
  sort(flater.begin(), flater.end(), sammenlign);
}
```

Siden kompilatoren ikke vet hvordan vi vil sortere flatene, må vi oppgi en funksjon som definerer hvordan en sammenligner to flater. I tilfellet over så blir flatene sortert etter stigende areal. Vi kunne også sortert etter for eksempel **bredde** ved å skrive **sammenlign** funksjonen slik:

```
bool sammenlign(const Flate &a, const Flate &b) {
  return a.bredde < b.bredde;
}
```

1.5.2 Anonyme funksjoner

I stedet for å skrive en egen **sammenlign**-funksjon, kan vi bruke en anonym funksjon som parameter til sort-algoritmen:

```
sort(flater.begin(), flater.end(), [](const Flate &a, const Flate &b) {
  return a.finnAreal() < b.finnAreal();
});
```

Den anonyme funksjonen `[](const Flate &a, const Flate &b) { … }` har samme parametere som **sammenlign** funksjonen. Kompilatoren finner også ut at den returnerer bool. Derimot kan vi spesifisere retur type ved å for eksempel skrive:

```
sort(flater.begin(), flater.end(), [](const Flate &a, const Flate &b) -> bool {
  return a.finnAreal() < b.finnAreal();
});
```

Du lurer sikkert på hva [] betyr. Her kan en derimot spesifisere hva en ønsker å overføre til den anonyme funksjonen (på engelsk: capture-list) uavhengig av parametre funksjonen har. En kan overføre både referanser og kopier. Vi kan også kalle anonyme funksjoner som vanlige funksjoner ved hjelp av ()-operatoren. I følgende eksempel overfører vi en referanse og kaller funksjonen to ganger, før og etter variabelen **tall** får verdien 2:

```
#include <iostream>

using namespace std;

int main() {
  int tall=1;
  auto skriv_tall_variabelen=[&tall]() { //tall blir her tilgjengelig
                                         //i funksjonen gjennom en referanse
    cout << tall << endl;
  };
  skriv_tall_variabelen();
  tall=2;
  skriv_tall_variabelen();
}
/* Utskrift:
1
2
*/
```

Den anonyme funksjonen skriv_tall_variabelen endrer på denne måten oppførsel når tall blir endret. Dette kan av og til være ønskelig. Vi kan også overføre en kopi av **tall**:

```
#include <iostream>

using namespace std;

int main() {
  int tall=1;
  auto skriv_tall_variabelen=[tall]() { //tall blir her kopiert til funksjonen
    cout << tall << endl;
  };
  skriv_tall_variabelen();
  tall=2;
```

```
  skriv_tall_variabelen();
}
/* Utskrift:
1
1
*/
```

Et litt større eksempel med bruk av anonyme funksjoner sammen med GUI biblioteket **gtkmm** finner du på https://github.com/ntnu-ifud1048/gtkmm-example. Her blir klasse instansen **this** overført til anonyme funksjoner som definerer hva som skal utføres når de ulike GUI elementene blir brukt. Instansen **this** blir overført for å kunne bruke datamedlemmene i de anonyme funksjonene (følg instruksjonene i lenken over for å kjøre eksempelet):

```
#include <gtkmm.h>

class Window : public Gtk::Window {
public:
  Gtk::VBox vbox;
  Gtk::Entry entry;
  Gtk::Button button;
  Gtk::Label label;

  Window() {
    button.set_label("Click here");

    vbox.pack_start(entry);  //Add the widget entry to vbox
    vbox.pack_start(button); //Add the widget button to vbox
    vbox.pack_start(label);  //Add the widget label to vbox

    add(vbox);  //Add vbox to window
    show_all(); //Show all widgets

    entry.signal_changed().connect([this]() {
      label.set_text("Entry now contains: " + entry.get_text());
    });

    entry.signal_activate().connect([this]() {
      label.set_text("Entry activated");
    });

    button.signal_clicked().connect([this]() {
      label.set_text("Button clicked");
    });
```

```
  }
};

int main() {
  Gtk::Main gtk_main;
  Window window;
  gtk_main.run(window);
}
```