Templates: Basics

Forelesning 8, vår 2015

Alfred Bratterud

Generic Programming

* Motivasjon: Vi ønsker å skrive en funksjon "print(...)", som skriver ut argumentet
* Hva er argumentet i et strengt typet språk?
  * Int? Heksadesimal? Eller desimal?
  * Char? Eller bare en byte?
  * String? Eller et char-array?
* For hver type trenger vi da en egen print-funksjon
* ...Med mindre vi har noe generisk

Templates

* En template er en "midlertidig type"
* For funksjoner, eller for klasser
* Det avgjøres "compile time" hvilken type som vil brukes
* For hver type som bruker funksjonen/klassen, lages en kopi

Generisk print()

* Vi begynner med å definere en generisk type:
  ```
  template<typename myType>
  void generic_function(myType t)
  ```
* Dette er å betrakte som en ny type, myType i funksjonsdefinisjonen:

```
template<typename myType>
void print(myType t){
    cout << t << endl;
}
```

* Hva skjer nå: print(5), print('á'), print("øy")?

Generisk print()

```
template<typename myType>
void print(myType t){
    cout << t << endl;
}
```

- Kalles print(5)? kompilator oppretter en funksjon print, som tar int som arg.
- Kalles print('c')? Kompilator oppretter en funksjon som tar char som arg.
- Men, innholdet er likt - ingen garanti for at det virker: class myClass{}; ...print(myClass c)...?

Templates forts.

* Flere template-argumenter gir flere kombinasjoner av typer

```
template<typename type1,typename type2>
void print2(type1 t1, type2 t2){
    cout << t1 << " and " << t2 << endl;
}
```

* Skopet gjelder kun for klassen eller funksjonen direkte etter
* Samme navn, som feks. "T" kan gjenbrukes i flere funksjoner i samme namespace, men må da oppgis på nytt hvert sted.

Demo:

templates1.cpp
template_mini.cpp

Template klasser

* template<typename key,typename value>
  class myHash{
      key k;
      value v;
    public:
      void myHash(key nKey, value nValue);
      void printValue(); printKey();
  }

* Kompilator vil nå opprette klassene ettersom de kalles - men type må spesifiseres:

  myHash <string, int> h2("age",12);

  myHash <int, char> h2(5, '5');

Template parametre

* Når vi skriver:
  `template<typename key,typename value>`
  `class myHash{`
  Har vi introdusert to generisk typer som "template parametre" til klassen
* Vi bruker ordet "typename" for å si at "key" og "value" er "typer", (vi kan også bruke ordet "class")
* Men, vi kan også bruke tall som template parametre:
  `template<typename T, int SIZE>`
  `struct myArr{ T elements[SIZE]; }`
* Nå har vi både en type, og en "int" som template-parametre
* For hver "myArr" vi lager, med ulik "int", oppretter kompilatoren en egen struct.

Demo:

key_value.hpp
int_template_params.hpp
templates2.cpp

Mye mer å si om templates...

Kanskje vi får til en video til, men sitter dette skikkelig, er det nok til å bestå kurset.