

TDT4102 Gjesteforelesning <Templates>

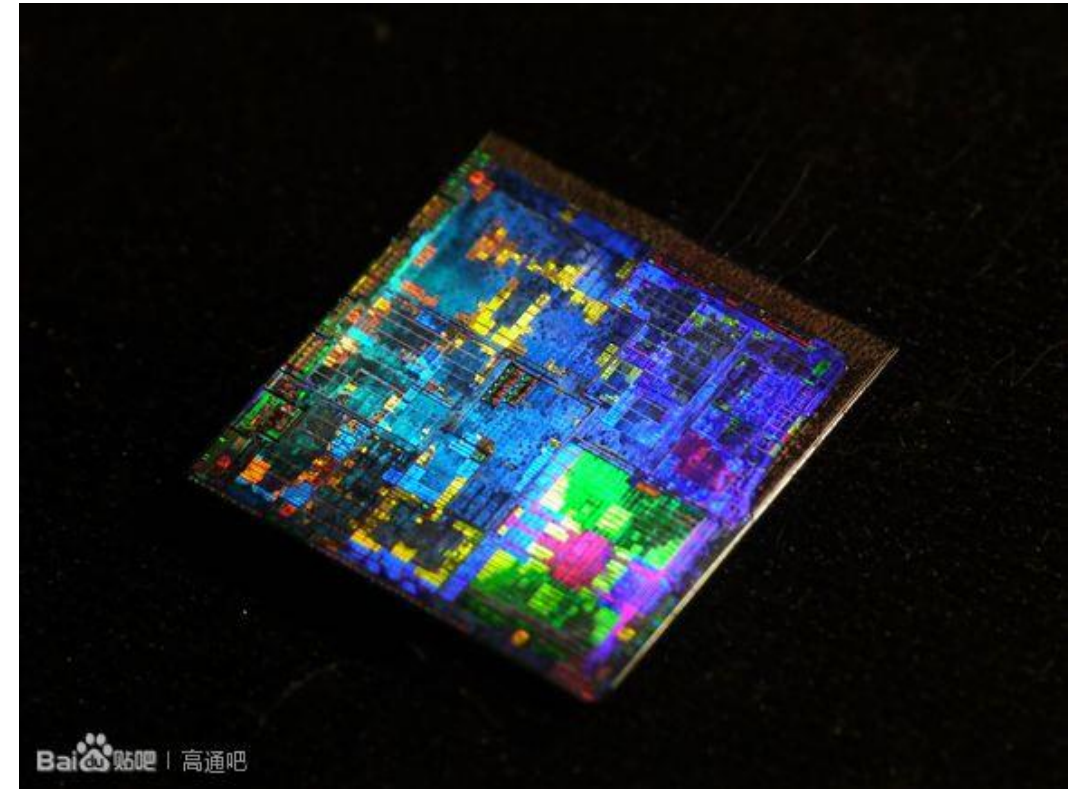
Stian Valentin Svedenborg – ARM

Hvem er jeg?

- Master i Fysikk og Matematikk (Industriell matematikk)
- Undass/Vitass i TDT4102 (4 år)
- Softwareutvikler i ARM Norge. (3 år)
 - HW Verifikasjon i C++
 - Grafikkdriverutvikling i C++ for Vulkan Grafikk API'et
 - Testing infrastruktur i Python

Hvem er ARM?

- Hardware «Intellectual Property» selskap
- Spesialisert på lav-energi systemer
 - Mobiltelefoner/Tablets
 - Mikrokontrollere
 - Datasentre
- I Norge:
 - Mali Grafikkprosessorer
 - Verdens mest solgte grafikkprosessor
 - Software- og hardwareutvikling
- Alle sommerjobber er besatt i år, men er du interessert for neste år kontakt HR:
ane.tamburstuen@arm.com



Dagens Tema: Templates

- Mer *generisk* (generell) kode.
 - Gjenbrukbar
 - Rask
 - Fleksibel
- Gjør dette ved å abstrahere *typene* i klassen/funksjonen.
- Dere har allerede brukt templates:
 - `std::vector<typename T>`
 - `std::map<typename K, typename V>`

STL – Standard Template Library (Kapittel 19)

- Et bibliotek med standard klasser designet for å kunne gjenbrukes.
- Klasser å merke seg:
 - `std::vector<T>`
 - `std::list<T>`
 - `std::deque<T>`
 - `std::set<T>`
 - `std::map<K,V>`
 - `unordered_set<T>`
 - `unordered_map<K,V>`

std::vector<typename T>

- Egner seg til å dynamisk lagre mange elementer.
- Egner seg til å legge til/fjerne elementer på slutten av vektoren.
- Egner seg til å aksessere elementer i vilkårlig rekkefølge.
- Egner seg ikke til å sette inn/fjerne elementer midt i listen.

Operasjon	Kompleksitet
T& operator[](int index)	O(1)
iterator insert(iterator pos,T value)	O(N)
iterator erase(iterator pos)	O(N)
void push_back(T value)	O(1)
void pop_back()	O(1)
int size()	O(1)
iterator find(T value)	O(N)


std::deque<typename T> - Double-Ended QUEue

- Som vector, men kan effektivt legge til/fjerne elementer både på starten og slutten av listen.

Operasjon	Kompleksitet
T& operator[](int index)	O(1)
iterator insert(iterator pos, T value)	O(N)
iterator erase(iterator pos)	O(N)
void push_back(T value)	O(1)
void pop_back()	O(1)
void push_front(T value)	O(1)
void pop_front()	O(1)
int size()	O(1)
iterator find(T value)	O(N)

std::list<typename T> - doubly linked list

- Egner seg når du trenger å sette inn/fjerne elementer midt i listen.
- Egner seg ikke når du må «hoppe rundt» i listen.

Operasjon	Kompleksitet
T& operator[](int index)	— 
iterator insert(iterator i, T value)	O(1)
iterator erase(iterator pos)	O(1)
void push_back(T value)	O(1)
void pop_back()	O(1)
void push_front(T value)	O(1)
void pop_front()	O(1)
int size()	O(N)/O(1)
iterator find(T value)	O(N)

Ikke støttet, må bruke iteratører.

std::set<typename T>

- Egenskaper:
 - Elementene i settet er alltid sortert.
 - Det lagres kun én forekomst av hver verdi.
 - Optimalisert for å sjekke om et element finnes i settet.

Operasjon	Kompleksitet
std::pair<iterator, bool> insert(T value)	$O(\log N)$
int erase(T value)	$O(\log N)$
int size()	$O(1)$
iterator find(T value)	$O(\log N)$

`std::map<typename KeyT, typename ValueT>`

- **Egenskaper:**

- Elementene i mappet er alltid sortert på nøkkel.
- Det lagres kun én forekomst av hver nøkkel.
- Optimalisert for å sjekke om en nøkkel finnes i mappet.

Operasjon	Kompleksitet
<code>ValueT& operator[] (KeyT key)</code>	$O(\log N)$
<code>std::pair<iterator, bool> insert(std::pair<KeyT, ValueT> element)</code>	$O(\log N)$
<code>void erase(KeyT key)</code>	$O(\log N)$
<code>int size()</code>	$O(1)$
<code>iterator find(KeyT key)</code>	$O(\log N)$

Gotcha! – Vanlige fallgruver for Set og Map

- For `set<T>` må `operator<` være definert for typen `T`.
- For `map` må `operator<` være definert for nøkkeltypen (`KeyT`).
 - Det er ingen slike krav for `operator<` på verditypen (`ValueT`).
- For `map`: Hvis du bruker `operator[]` med en nøkkel som ikke eksisterer, så blir det opprettet og default-initialisert.
 - Basistyper (`int`, `float`, `double`, `char`, pekere etc) blir initialisert til 0.
 - Klasser og strukter blir default initialisert.

Varianter av Set og Map:

- `unordered_set<T>`, `unordered_map<K,V>`
 - Som set/map, men er implementert som et *hashmap*.
 - `find(...)`, `insert(...)` og `remove(...)` blir derfor $O(1)$.
 - Ikke lenger sortert.
 - Krever ikke `operator<`, men krever en hash *functor* for egendefinerte key-typer. (avansert)
- `multiset<T>`, `multimap<K,V>`
 - Som set/map, men tillater duplikat elementer/nøkler.
- `unordered_multiset<T>`, `unordered_multimap<K,V>` finnes også.

Felles for alle containerklassene: Iteratorer

Operasjon	Beskrivelse
iterator begin()	Iterator til første element.
iterator end()	Iterator til elementet etter siste element.
iterator rbegin()	Reversiterator til siste element
iterator rend()	Reversiterator til elementet «før» første element.

Iteratorer – Det som oppfører seg som pekere

- Bruk av *iteratorer* er en teknikk for å aksessere elementer i en container.
- Se på det som en peker til et element i containeren.
- Men en iterator er ikke* en peker!

```
int main() {  
    std::vector<int> v = { 0, 1, 23, 4 };  
  
    for (std::vector<int>::iterator it = v.begin();  
        it != v.end();  
        ++it )  
    {  
        std::cout << *it << std::endl;  
    }  
}
```

* (nødvendigvis)

Forenklinger – auto og range-for-loop

```
int main() {  
    std::vector<int> v = { 0, 1, 23, 4 };  
  
    for (auto it = v.begin(); it != v.end(); ++it )  
    {  
        std::cout << *it << std::endl;  
    }  
  
    for (int & value : v )  
    {  
        std::cout << value << std::endl;  
    }  
  
    for (auto & value : v )  
    {  
        std::cout << value << std::endl;  
    }  
}
```

Eksempel:

- Oppgave:

- Les inn en liste med navn fra en fil.
- Sorter de på etternavn .
- Print ut navnene på en strukturert måte.

etternavn:

fornavn 1

fornavn 2

(...)

- I tillegg:

- Skriv ut sortert.
- Skriv også ut hvor mange forekomster av hvert navn.

Truls Nidarvoll
Knut Trostrup
Stian Nidarvoll
Knut Olsen
Ida Trostrup
Ola Olsen
Truls Hansen
Trude Nidarvoll
Stian Olsen
Olga Svedenborg
(...)

Skrive egne templates (Kapittel 16)

- Designet for generelle funksjoner/klasser
 - Viste seg å være turing-komplett.
- Unngå å gjenta seg selv.
 - Overlasting er ofte mulig, men det er vanskelig å forutse behovene for fremtiden.
- Optimalisere kode.



Syntaks: Template klasser

```
template<typename T1, typename T2>
struct Pair {
    Pair() = default;
    Pair(const T1 &first, const T2 &second) :
        first(first), second(second) {}

    T1 first;
    T2 second;
};

int main() {
    Pair<int, std::string> p(42, "Meaning of everything");
    std::cout << p.first << ": " << p.second << std::endl;
}
```

Template-prefiks

Bruker typene T1 og T2
som «place-holders»

T1 og T2 blir så bundet til to
eksplisitte typer når templateen
instansieres.

Mer syntaks

- `template<typename T> == template<class T>`
- Templateparametre er som oftest typer, men trenger ikke være det.
 - Kan også være konstanter.
 - Funksjonspekere.
 - Eller lignende.
- For bruk av template klasser må template parameterene alltid spesifiseres.
 - Unntak: Når en template klasse referererer til seg selv kan parametrene droppes helt.

Template Gotchas!

- Kompilatoren må ha hele templatens tilgjengelig når den instansieres.
 - Plasser derfor hele definisjonen i en header fil.
- Templates kan drastisk øke kompileringstiden til programmet.
- Uvettig bruk av templates kan lede til «code bloat» som gjør program store (og tregere).

Eksempel: Array

- Vi skal implementere en forenklet variant av `std::array<T, N>`.
- Krav:
 - Skal være en «wrapper» for en innebygget liste.
 - Skal lagre størrelsen på tabellen. (Og gjøre det mulig å få tak i denne.)
 - Skal støtte operator[].
- Hvis vi får tid:
 - Utvid klassen til å ha iteratorer og kunne brukes i range-for løkker.

Templatefunksjoner

- Noen ganger skriver vi funksjoner som er nyttige for mange forskjellige typer...

```
void Swap(int & v1, int & v2) {  
    int temp = v1;  
    v1 = v2;  
    v2 = temp;  
}
```

Templatefunksjoner

- Vi kan løse det med overlasting...

```
void Swap(int & v1, int & v2) {  
    int temp = v1;  
    v1 = v2;  
    v2 = temp;  
}  
  
void Swap(float & v1, float & v2) {  
    float temp = v1;  
    v1 = v2;  
    v2 = temp;  
}  
  
void Swap(std::string & v1, std::string & v2) {  
    (...)
```

Templatefunksjoner

- Da er det mye enklere med en templatefunksjon.

```
template<class T>
void Swap(T & v1, T & v2) {
    T temp = v1;
    v1 = v2;
    v2 = temp;
}
```


Templatefunksjoner

- Du kan også bruke templatefunksjoner i kombinasjon med overlasting:

```
template<class T>
void Swap(T & v1, T & v2) {
    T temp = v1;
    v1 = v2;
    v2 = temp;
}

void Swap(std::string & v1, std::string & v2) {
    v1.swap(v2);
}

template<class T>
void Swap(std::vector<T> & v1, std::vector<T> & v2) {
    v1.swap(v2);
}
```

Eksempel: Find

- Skriv templatefunksjonen:

```
iterator Find(iterator begin, iterator end, T value);
```

- Returner end hvis verdien ikke ble funnet.

- Pssst... Stian, ikke glem <algorithm>!



Helt på tampen (og utenfor pensum)

- Verdt å se på for alle:
 - [Boost](#) – Et bibliotek som utvider mulighetene til STL drastisk.
- For spesielt interesserte:
 - [Template Spesialisering](#) – Eksplisitt spesifisere varianter av en template.
 - [Meta-programmering](#) – Program som kjører i kompilatoren.
 - [SFINAE](#) – «Substitution Failure Is Not An Error»
 - [Variadic templates](#) – Template funksjoner/klasser med vilkårlig mange parametre.