Template

Vîlculescu Mihai-Bogdan

19 aprilie 2020

Contents

1 Introducere

1.1 Definiție

Template-ul este un instrument prin care se poate evita rescrierea de cod.

Presupune scrierea unei **singure clase / funcții** a cărei comportament este asemănător și se modifică doar dacă se modifică și un anumit **tip de date**.

Observație

Template este o **formă de polimorfism la compilare (compile-time)**.

1.2 Exemple

Cele mai bune exemple de **clase template** sunt cele din **STL (Standard Template Library)**.

Să discutăm despre **clasa vector**.

Clasa vector

Face aceleași operații (push, pop, get, ...) pentru mai multe tipuri de date (int, char, string, ...).

Fără template ar însemna să se creeze de către developer câte o clasă diferită pentru fiecare nou tip de date care se poate afla într-un vector.

Cu template se creează o singură clasă pentru care se implementează operațiile în funcție de un tip de date necunoscut.

2 Funcții template

2.1 Sintaxă

Când se declară o clasă / o funcție template, trebuie de fiecare dată specificat cuvântul cheie **template** înainte.

```cpp
template <typename T>
void f (T x) {
    // functie template
    cout << x;
}

void g (int x) {
    // functie obisnuita fara template
    cout << x;
}

int main () {
    f<int>(3);
    f<char>('c');
    g(2);
    return 0;
}
```

Template vs non-template functions

2.2 Compilare

Pentru a înțelege cu adevărat cum funcționează acest instrument numit *template*, este important de înțeles ce se întâmplă la compilare.

Să studiem pașii care au loc în momentul compilării unei funcții template:

Pași la compilare

1. Când e întâlnită o funcție template, e compilată, fără a se ține cont de tipul de date necunoscut.

2. În momentul în care se apelează o funcție template, compilatorul creează o nouă funcție obișnuită în care tipul de date necunoscut este înlocuit cu cel specificat în apel.

```
f<int>(3) ===> void f (int x) {...}
f<char>('c') ===> void f (char x) {...}
...
```

Acești pași sunt identici și pentru **clasele template**.

2.3 Specializarea funcțiilor template

Specializarea unei funcții template este o **particularizare** a acelei funcții template **pentru un anumit tip de date**.

```
template <typename T>
void f (T x) {
    cout << "Template func";
}

template <>
void f (int x) {
    cout << "Specialization func";
}

int main () {
    f(3); // Specialization func
    f('c'); // Template func
}
```

Specializarea unei funcții template

2.4 Prioritatea la supraîncărcare

O problemă importantă care apare de la folosirea funcțiilor template este **prioritatea** cu care sunt apelate acestea în momentul în care s-au **supraîncărcat** mai multe funcții.

Să urmărim exemplul de mai jos:

```
template <typename T>
void f (T x) {
    cout << "Template func";
}

template <>
void f (int x) {
    cout << "Specialization func";
}

void f (char c) {
    cout << "Regular func (char)";
}

void f (int c) {
    cout << "Regular func (int)";
}

int main () {
    f(3); // Regular func (int)
    f('c'); // Regular func (char)

    f<int>(3); // Specialization func
    f<char>('c'); // Template func
}
```

Supraîncărcare de funcții

Putem observa că prioritatea cea mai mare o au **funcțiile fără template**. Să urmărim cum procedează compilatorul când caută o funcție care se potrivește cu un apel:

```
1. Se caută o funcție obișnuită care să aibă parametri
potriviți.

2. Dacă nu s-a găsit la punctul 1., se caută o
specializare template cu parametri potriviți.

3. Dacă nici 2. nu a furnizat un rezultat, se caută
o funcție template cu numărul de parametri potriviți.

4. Dacă nici 3. nu a furnizat un rezultat, se întoarce
o eroare.
```

3 Clase template

3.1 Sintaxă

Sintaxa este asemănătoare cu cea de la funcții.

```
template <typename T>
class A {
    // clasa template
    T x;
    int y;
    // ...
};

class B {
    // clasa obisnuita fara template
    char x;
    int y;
    // ...
};

int main () {
    A<int> a1;
    A<char> a2;

    B b;
    return 0;
}
```

Clase template vs non-template

Spre deosebire de funcțiile template, aici este **obligatorie** specificarea tipului de date la declararea unui obiect (între <>).

3.2 Specializarea claselor template

Definiția este identică precum cea de la funcții și sintaxa este asemănătoare.

```
template <typename T>
class A {
    T x;

    public:
        A() { cout << "Template"; }
};

template <>
class A<int> {
    int x;

    public:
        A<int>() { cout << "Spec"; }
};

int main () {
    A<int> a1; // Spec
    A<char> a2; // Template
}
```

Clase template specializate

3.3 Metode

Când se dorește scrierea corpului metodelor unei clase template în afara clasei, sintaxa arată astfel:

```
template <typename T>
class A {
    T x;

    public:
        T getX () const;
        void setX (T);
};

template <typename T>
T A<T>::getX () const {
    return x;
}

template <typename T>
void A<T>::setX (T _x) {
    x = _x;
}
```

Metodele clasei template

4 Observații finale

- **typename** poate fi înlocuit începând cu standardul **C++17** cu **class** fără vreo diferență importantă. Am lăsat în resurse niște link-uri unde se dezbate puțin subiectul acesta dacă vă interesează.

5 Resurse

- https://www.geeksforgeeks.org/exception-handling-c/
- http://www.cplusplus.com/doc/tutorial/exceptions/
- TutorialsPoint
- Niste exemple mai dragute de pe site-ul Microsoft
- Typename vs Class - stack overflow
- Typename vs Class - microsoft