C++ за напреднали

Templates and Generic Programming

Йордан Зайков Димитър Трендафилов

Факултет по Математика и Информатика

Изборна дисциплина, зимен семестър 2009 / 2010 г.

Съдържание

Обща информация -

- Една от последно вмъкнатите в стандарта (C++98) възможности на езика
  - По-лоша поддръжка от страна на компилаторите и по-големи разлики между тях
    - keyword “export”
  - Липса на ортогоналност поради технически или исторически причини
    - параметри по подразбиране за шаблонни функции
    - floating-point стойности за шаблонни аргументи
  - Различни синтактични особености

Обща информация +

- Едно от най-мощните изразителни средства на C++
  - Шаблоните в C++ са turing-complete (т.е. са пълен език за програмиране)
  - metaprogramming - изчисления по време на компилация и “смятане” с типове
  - expression templates и други Domain Specific Languages вмъкнати в C++
  - lambda изрази и n-торки
  - type traits

Съдържание

Общ вид на декларация на шаблонна функция

```
template <"тип параметър1" "име на параметър1"
    {, "тип параметър2" "име на параметър2"}>
<тип на резултата>
f(<формални аргументи на функцията>);
```

Шаблонни параметри

- “тип параметър” - типът на параметъра
  - typename, class - този параметър означава име на тип
    - typename и class са едно и също в този контекст, но typename е за предпочитане
    - тук не можете да използвате struct вместо class
  - име на тип - този параметър ще бъде стойност от посочения тип
    - тук не можете да използвате произволен тип
  - декларация на друг шаблон !

Пример за шаблонна функция

```
template <typename T>
inline const T& max(const T& x, const T& y) {
    return (x < y)? y : x;
}
int main() {
    // using ::max to avoid std::max
    std::cout << ::max(24, 42) << std::endl;
    std::cout << ::max<double>(9.8, 3.14) <<
        std::endl;
    // явно задаване на T
    std::cout <<
        ::max(std::string("path"),
              std::string("pathology"));
    return 0;
}
```

инстанциране

- Шаблонната функция max не се компилира до една функция, която да работи с всички типове аргументи, а за всеки тип аргументи се създава отделна функция max.
- Това създаване се нарича инстанциране (instantiation)
- инстанцирането е автоматично, но може да бъде и явно.
- Ако се опитаме да инстанцираме шаблонна функция за тип аргументи, който не изпълнява всички изисквания на шаблона (в случая на max - operator<), ще се получи грешка при компилация

инстанциране

- Шаблонната функция max не се компилира до една функция, която да работи с всички типове аргументи, а за всеки тип аргументи се създава отделна функция max.
- Това създаване се нарича инстанциране (instantiation)
- инстанцирането е автоматично, но може да бъде и явно.
- Ако се опитаме да инстанцираме шаблонна функция за тип аргументи, който не изпълнява всички изисквания на шаблона (в случая на max - operator<), ще се получи грешка при компилация

инстанциране

- Шаблонната функция max не се компилира до една функция, която да работи с всички типове аргументи, а за всеки тип аргументи се създава отделна функция max.
- Това създаване се нарича инстанциране (instantiation)
- инстанцирането е автоматично, но може да бъде и явно.
- Ако се опитаме да инстанцираме шаблонна функция за тип аргументи, който не изпълнява всички изисквания на шаблона (в случая на max - operator<), ще се получи грешка при компилация

Компилиране на шаблони

- Шаблоните се компилират на два етапа:
  1. Преди инстанциране - компилатора прави само проверка на синтаксиса - например за липсващи ";"
  2. По време на инстанциране - компилатора компилира кода на шаблона и прави проверка за валидността на всички използвани функции, методи и оператори

- За да бъде инстанциран един шаблон, компилаторът трябва да вижда неговата дефиниция.
- Това налага повечето шаблони да бъдат дефинирани в header файлове, които се включват в ползващите шаблона файлове

Компилиране на шаблони

- Шаблоните се компилират на два етапа:
  1. Преди инстанциране - компилатора прави само проверка на синтаксиса - например за липсващи “;”
  2. По време на инстанциране - компилатора компилира кода на шаблона и прави проверка за валидността на всички използвани функции, методи и оператори
- За да бъде инстанциран един шаблон, компилаторът трябва да вижда неговата дефиниция.
- Това налага повечето шаблони да бъдат дефинирани в header файлове, които се включват в ползващите шаблона файлове

Бележки

- Шаблонните функции дефинират фамилия от функции за различните шаблонни аргументи.
- Когато подавате шаблонни аргументи, шаблонната функция се инстанцира за тези типове.
- Можете явно да задавате шаблонни аргументи.

Съдържание

Общ вид на декларация на шаблонен клас

```
template <"тип параметър1" "име на параметър1"
    {, "тип параметър2" "име на параметър2"}>
[class|struct] A;
```

Пример

```
template <typename T>
class Stack {
  public:
    void push(const T&);
    void pop();
    T top() const;
    bool empty() const;
  private:
    std::vector<T> _s;
};

template <typename T>
void Stack<T>::push(const T& x) { _s.push_back(
    x); }
```

Използване на Stack

```
int main()
{
    Stack<int> stackInts;
    // предизвиква инстанциране на Stack<int>,
       но не на всички функции
    Stack<string> stackStrings;
    // Stack<int> и Stack<string> са два
       напълно
    // различни типа !
    stackInts.push(42);
    // ...
    return 0;
}
```

Използване на Stack

```
void f(Stack &s);
// грешка, Stack е име на шаблон, а не тип

void f(Stack<int>& s);
// ok, Stack<int> е име на тип

template <typename T>
void f(Stack<T>& s);
// ok, f е шаблонна функция, Stack<T> ще
// бъде име на тип при инстанцирането на f
```

Подразбиращи се аргументи на шаблон

```
template <typename T,
          typename Cont = std::vector<T> >
class Stack {
  public:
    // ...
  private:
    Cont _s;
};

int main()
{
    Stack<int> stackIntsVector;
    Stack<int, deque<int> > stackIntsDeque;
}
```

Инстанциране на шаблонни класове

- Шаблоните се проверяват за синтактични грешки, а останалите проверки се правят при инстанцирането
- инстанцират се само методите, който се използват!
  - Някоя семантична грешка може да мине незабелязана от компилатора
  - За да се инстанцират всички методи на шаблона използвайте явно инстанциране (explicit instantiation)

Пример за инстанциране на шаблонен клас

```
template <typename T>
bool Stack<T>::empty() const { return _s == 0;
    }
// vector<T> няма operator==(int);

int
main() {
    Stack<int> s;
    s.push(42);
    return 0;
}
// Не използваме никъде Stack<int>::empty!
// Кода се компилира и изпълнява чудесно
```

Бележки

- инстанцират се само тези методи на шаблона, които се използват!
- Можете да дефинирате стойност по подразбиране за параметри на шаблона, те могат да използват предходните параметри.

Съдържание

Не-типови шаблонни параметри

- Шаблонните параметри освен типове могат да и обикновени стойности, но от типове отговарящи на определени условия

Пример за не-типови параметри

```
template <typename T, unsigned Size>
class Array {
  public:
    // ...
    T& operator[] (size_t i) { return _a[i]; }
    T& at(size_t i) {
        if (i > Size)
            throw std::out_of_range("i too
                large");
        return _a[i];
    }
  private:
    T[Size] _a;
};
```

Пример за не-типови параметри

```
template <typename T, T Value>
T increase(const T& x) { return x + Value; }
// съвсем не достатъчно generic

int main() {
    cout << increase<int, 10>(32) << endl;
    return 0;
}
```

Условия за не-типовите шаблонни параметри

- Могат да бъдат константни интегрални стойности (включително enum) или указатели към обекти с външно свързване
  - интегрална стойност - проста, неделима стойност
  - указател към обект с външно свързване - указател към глобален за цялата програма
- Не могат да бъдат floating-point числа и обекти от непримитивен тип
  - floating-point не се разрешава по исторически причини, в бъдеще може да се допусне
  - Вече трябва да е ясно защо increase не е достатъчно generic

Примери за не-типови параметри

```
template <const char* s>
class MyClass;

MyClass<"Hello"> e;
// грешка, "Hello" е string literal

const char* s = "hello";
MyClass<s> e2;
// грешка, s е указател към обект с internal
    linkage

extern const char[] s2 = "hello";
MyClass<s2> o; // ok
```

Съдържание

Компилиране на шаблони

- Шаблоните се компилират на два етапа:
  1. Преди инстанциране - компилатора прави само проверка на синтаксиса - например за липсващи ";"
  2. По време на инстанциране - компилатора компилира кода на шаблона и прави проверка за валидността на всички използвани функции, методи и оператори
- За да бъде инстанциран един шаблон, компилаторът трябва да вижда неговата дефиниция.
- Това налага повечето шаблони да бъдат дефинирани в header файлове, които се включват в ползващите шаблона файлове

Компилиране на шаблони

- Шаблоните се компилират на два етапа:
  1. Преди инстанциране - компилатора прави само проверка на синтаксиса - например за липсващи “;”
  2. По време на инстанциране - компилатора компилира кода на шаблона и прави проверка за валидността на всички използвани функции, методи и оператори
- За да бъде инстанциран един шаблон, компилаторът трябва да вижда неговата дефиниция.
- Това налага повечето шаблони да бъдат дефинирани в header файлове, които се включват в ползващите шаблона файлове

инстанциране на шаблони

- Не явно - автоматично при използване на шаблонна функция или клас
- Явно (explicit instantiation)
  - предизвиква инстанцирането на всички методи на класа
  - **template int** f<**double**, **double**>(**double**, **double**);
  - **template class** MyStack<**int**>;

инстанциране на шаблон

```
template <typename T> class C; // forward
C<int>* p; // ok
template <typename T>
class C {
  public:
    void f(); // декларация на C::f()
}; // дефиниция на C

void g(C<int>& c)
{ c.f(); } // използва дефиницията на C
// необходима е дефиниция на C::f()
```

инстанциране на шаблон

```
template <typename T>
class C {
  public:
    C(int); // може да се ползва за
        конвертирания
};
void f(C<double> &);
void f(int);

void g() { f(42); }
// компилаторът може да инстанцира C<double>
// за да провери дали коя f да извика
```

Мързеливо инстанциране

Каква част от един шаблонен клас се инстанцира при неявно инстанциране?

- Колкото се може по-малко
  - декларациите на всички членове на класа и на членовете на съдържащи се анонимни union
  - дефинициите на virtual член функции биха могли да бъдат или да не бъдат инстанцирани
    - повечето компилатори ще ги инстанцират, тъй като имплементацията им на virtual механизма изисква тези функции да съществуват
  - параметри по подразбиране на функции се инстанцират само ако име извикване на функцията, при което параметърът по подразбиране се използва

Мързеливо инстанциране

Каква част от един шаблонен клас се инстанцира при неявно инстанциране?

- Колкото се може по-малко
  - декларациите на всички членове на класа и на членовете на съдържащи се анонимни union
  - дефинициите на virtual член функции биха могли да бъдат или да не бъдат инстанцирани
    - повечето компилатори ще ги инстанцират, тъй като имплементацията им на virtual механизма изисква тези функции да съществуват
  - параметри по подразбиране на функции се инстанцират само ако име извикване на функцията, при което параметърът по подразбиране се използва

Съдържание

Name Lookup?

name lookup процесът, при който компилаторът свързва използваните имена с обектите, които те представляват

- в случая под обекти разбираме функции, оператори и типове

Видове имена в C++

- Разпределянето на имената в C++ в класове е доста сложно
- Най-основно (и сравнително достатъчно за нашите цели тук) е следното разделяне
  - квалифицирано име е име, чийто обхват е явно указан чрез оператора :: (scope resolution operator) или операторите . и -> (member access operators)
    - this->count и count не са съвсем едно и също, но са две имена на едно и също в рамките на дефиниция на клас
  - зависимо име е име, което зависи по някакъв начин от шаблонен параметър
    - std :: vector<T>::iterator е зависимо име ако T е шаблонен параметър и е независимо име ако T е име на известен тип
    - this->x - x е зависимо име, когато се среща в шаблон
    - f(x) е зависимо име, ако типът на x зависи от шаблонен параметър

Правила за търсене на имена

- Правилата за търсене на име са пълни с различни детайли за да работят интуитивно в общия случай и коректно в сложни частни случаи.

Основни правила

- квалифицирани имена се търсят в посоченото от квалификацията пространство от имена
- неквалифицираните имена се търсят в последователно разширяващи се пространства от имена
  - започва се от текущото пространство, след това се търси в пространството съдържащо текущото и така нататък докато не се намери обект определен от това име

Argument Dependent Lookup

- за неквалифицираните имена се прилага и Argument Dependent Lookup (ADL, Koenig Lookup)
  - когато се търси функция с определено име освен в текущото и обхващащите го пространства от имена се търси и в пространствата, асоциирани със типовете на аргументите
  - Базира се на идеята, че интефейсът на един клас е съставен от публичните му методи и всички функции дефинирани в същото пространство от имена, който приемат параметри от този клас

ADL в действие

```
namespace Lib {
    class A { // ... };
    ostream& operator<<(ostream& s,const A& a);
    bool operator==(const A& l, const A& r);
}

int main() {
Lib::A a1, a2;
    cout << a1 << "␣" << endl;
    cout << a2 == a1 << endl;
    return 0;
}
```

Асоциирани пространства от имена - 1

- за примитивните типове - празното множество
- за указатели и масиви - пространството асоциирано със съответния тип на сочения / съдържания обект
- за enum - пространството, в което е дефиниран enum-ът
- за членове на клас - пространството е самият клас

Асоциирани пространства от имена - 2

- за класове - множеството от асоциирани класове е самият клас, съдържащият го клас и всеки пряк или непряк родителски клас. Асоцииранoто пространство от имена са пространствата от имена, в които са деклариране асоциираните класове
  - при инстанциране на шаблон, класовете на типовите шаблонни параметри и класовете и пространствата, в които са декларирани шаблонните шаблонни аргументи също се включват
- за указател към член на клас X - освен асоциираните с X пространства се включват и пространствата асоциирани с типа на сочения член
  - за указатели към методи - типа на резултата и типовете на параметрите също допринасят с техните асоциирани пространства от имена

ADL

- ADL търси името във всяко от асоциираните пространства все едно името е било квалифицирано в това пространство, като игнорира using директивите

ADL в действие с повече детайли

```
namespace X {
    template <typename T> void f(T);
}
namespace N {
    using namespace X;
    enum E { e1 };
    void f(E) { cout << "N::f(E)\n"; }
}
void f(int) { cout << "::f()\n"; }
void g() {
    ::f(N::e1); // квалифицирано, без ADL
    f(N::e1); // неквалифицирано, ADL
    // X::f() изобщо не се разглежда
}
```

Two-Phase Lookup

- При първоначалното компилиране на шаблона, зависимите имена не могат да бъдат свързани с функции и типове.
- Затова се въвежда two-phase lookup
  1. Независимите имена се свързват със съответните функции и типове по време на компилация на шаблона
  2. Зависимите имена се свързват в момента на инстанциране - name lookup се прави в POI

Two-Phase Lookup - Template.hpp

```
void g(double);

template <typename T> class A {
  public:
    A() : x(0) { }
    void f() {
        g(3.14);
        g(this->x);
        g(x);
        g(2);
    }
    T x;
};
```

Two-Phase Lookup - main.cpp

```
#include "Template.hpp"
void g(int x) {
    cout << "f(int) " << x << endl;
}
void g(double x) {
    cout << "f(double) " << x << endl;
}
int main() {
    A<int> a;
    a.f();
    return 0;
}
```

Two-Phase Lookup - изход

```
// правилен
f(double) 3.14
f(int) 0
f(int) 0
f(double) 42

// ако компилаторът не поддържа
// two-phase lookup
f(double) 3.14
f(double) 0
f(double) 0
f(double) 42
```

Точка на инстанциране

Point of Instantiation (POI) точката на инстанциране на шаблона - на това място компилаторът слага дефиницията на инстанцирания шаблон

Point of instantiation за шаблонни функции

```
template <typaname T>
void f(T x) {
    if (x > 0) g(x);
}
// (1)
namespace Lib {
    // (2)
    void g(MyInt y) {
        // (3)
        f<MyInt>(42); // точка на извикване
        // (4)
    }
    // (5)
}
// (6)
```

Точка на инстанциране за функции

- Точката на инстанциране не може да съвпадне с точката на извикване, защото не можете да дефинирате функция в друга функция
  - съответно отпадат точки (3) и (4)
- в точки (1) и (2) g(MyInt x) не е видима, което не е особено интуитивно
- в точка (5) и (6) g(MyInt x) е видима
- Точката на инстанциране на шаблонни функции е веднага след най-близката декларация или дефиниция, която съдържа обръщение към шаблона (в подходящия namespace) - (6) в нашия пример

Как Ви се струва този код?

```
template <typaname T>
void f(T x) {
    if (x > 0)
        g(x);
}

namespace Lib {
    void g(int y) {
        // (2)
        f<int>(42);
        // (3)
    }
}
```

Отговор

Предишният пример не се компилира, защото:

- template void f(T) е глобален шаблон и инстнацията на шаблона съответно е в глобалния namespace
- няма глобална функция g
- ADL не може да намери g, понеже int е примитивен тип

Отговор

Предишният пример не се компилира, защото:

- template void f(T) е глобален шаблон и инстнацията на шаблона съответно е в глобалния namespace
- няма глобална функция g
- ADL не може да намери g, понеже int е примитивен тип

Отговор

Предишният пример не се компилира, защото:

- template void f(T) е глобален шаблон и инстнацията на шаблона съответно е в глобалния namespace
- няма глобална функция g
- ADL не може да намери g, понеже int е примитивен тип

Отговор

Предишният пример не се компилира, защото:

- template void f(T) е глобален шаблон и инстнацията на шаблона съответно е в глобалния namespace
- няма глобална функция g
- ADL не може да намери g, понеже int е примитивен тип

Point of instantiation за шаблонни класове

```
template <typename T> class S {
  public:
    T m;
};
// (7)
unsigned long
h() {
    // (8)
    return (unsigned long)sizeof(S<int>);
    // (9)
}
// (10)
```

Точка за инстанциране на класове

- (8) и (9) отпадат понеже не може дефиницията на namespace-видим клас не може да бъде вътре във функция (и не можем да имаме шаблони във функция)
- Ако изберем (10) изразът sizeof(S<int>) няма да бъде валиден
- Остава (7)
- Точка на инстанциране за шаблонни класове е веднага преди най-близката декларация или дефиниция, която съдържа обръщение към класа (в подходящия namespace)

Point of instantiation за шаблонни класове

```
template <typename T>
class S {
  public:
    typedef int I;
};
// (1)
template <typename T>
void f() {
  S<char>::I var1 = 41;
  typename S<T>::I var2 = 42;
}
int main() {
    f<double>();
}
// (2): (2a), (2b)
```

Точка за инстанциране на класове - 2

- (1) - S<char>
- (2a) - S<double>
- (2b) - f<double>

Модели на включване и експортиране

При инстанцирането на шаблона, компилаторът трябва да вижда неговата дефиниция

1. Ако дефиницията е в header файл, можем да я включим
2. Можем да експортираме дефиницията на шаблона и тя да бъде в друг source файл

Включване

```
// mytemplate.hpp
#ifndef MYTEMPLATE_HPP
#define MYTEMPLATE_HPP
template <typename T>
void f(T) { /*...*/ }
#endif

// main.cpp
#include "mytemplate.hpp"
void g() { f(42); }
```

Експортиране

```
// mytemplate.cpp
export template <typename T>
void f(T) { /*...*/ }

// mytemplate.hpp
export template <typename T>
void f(T);

// main.cpp
#include "mytemplate.hpp"
void g() { f(42); }
```

“export”

- “export” не се имплементира от никои от разпространените компилатори

Съдържание

Специализация на шаблон

Специализация на шаблон Промяна на дефиницията на шаблон за някакви “специални” аргументи на шаблона

Специализация на шаблонни функции

- Няма специализация на шаблонни функция
- Имаме function overloading

Специализация на шаблонни функции

- Няма специализация на шаблонни функция
- Имаме function overloading

Function Overloading на кратко

1. Формира се множеството от функции, имащи едно и също име с извикваната функция
2. От това множество се премахват всички, които няма шанс да са подходящи (различен брой параметри, липса на конвертиране на типовете на аргументите и параметрите)
3. От останалото множество се избира “най - доброто съвпадение” или се извежда съобщение, че има повече от едно такива
4. В случай че има “най - доброто съвпадение” се проверява неговата валидност (не е private метод, например)

Най-добро съвпадение

- Една функция е по-добро съвпадение от друга, ако всеки нейн параметър е по-близък до аргументите спрямо другата функция
- Близост на аргументите
  1. Точно съвпадение с евентуално добавяне на const/volatile
  2. Малки напасвания - array към указател
  3. Промотиране на тип. Преобразуване на типове без загуба на точност - char -> int, float -> double
  4. Стандартни конверсии
  5. Потребителски конверсии - operator int(), implicit ctor
  6. Съвпадение със ...

Function Overloading + Templates

- За да участва една шаблонна функция във множеството функции тя трябва да бъде точно съвпадение
- Не-шаблонна функция, която е точно съвпадение винаги се предпочита пред шаблонна функция

Function overloading and templates - 1

```
const int& max(const int& x, const int& y); // (1)
template <typename T>
const T& max(const T&, const T&); // (2)

int main () {
    ::max(4, 3); // (1)
    ::max(3.13, 3.14); // (2)
    ::max('a', 'b'); // (2)
    ::max<>(2, 4); // (2)
    ::max<double>(3, 4);
    ::max('a', 3.14); // (1)
}
```

Function overloading and templates - 2

```
const char* const&
max(const char* const&, const char* const&);
// const char* max(const char*, const char*);

template <typename T>
const T& max(const T& a, const T& b, const T& c
    ) {
     return max(max(a, b), c);
}
// не бихме могли да използваме overload-а за
// низове, ако предаваме аргументите по
    стойност
```

Специализация на шаблонни класове

- Специализация на класове позволява да оптимизирате шаблона за определени типове или да го “напаснете” за опредени аргументи
- Можете да специализирате целия клас или само един метод от него

Специализация на шаблон на клас

```
template <typename T> class Stack
{ /* ... */ }
// искаме стек от с-низове,
// като стекът пази копия на низовете
template <>
class Stack<const char*>
{
  // пълна специализация на Stack
};
```

Специализация на метод на шаблон на клас

```
template <typename T> class Stack
{ /* ... */ }

template <>
bool Stack<std::string>::empty() const
{
  // ...
}
```

Частична специализация

- Шаблоните за класове могат да бъдат специализирани от части
- При частична специализация трябва да специализирате целия шаблон

Частична Специализация на метод на шаблон на клас

```
template <typename T1, typename T2> class
    MyClass
{ /* ... */ }

template <typename T> class MyClass<T, T>
{ /* ... */ }

template <typename T> class MyClass<int, T>
{ /* ... */ }

template <typename T1, typename T2>
class MyClass<T1*, T2*>
{ /* ... */ }
```

Съдържание

Необходимостта от typename

```
template <typename T>
class MyClass {
    typename T::SubType * ptr; // static member
    // T::SubType * ptr; // multiplication
};
```

Ключова дума “typename”

- Квалифицирано зависимо име не се смята за тип, освен ако не е предшествано от typename
- typename е задължително да се постави пред едно име, ако
  1. името се появава в шаблон
  2. името е квалифицирано
  3. не е използвано в списък на базовите класове или в списък за инициализация на базовите класове
  4. името зависи от параметър на шаблона
- Ако едно от първите 3 не е изпълнено е грешка да се използва typename

Quiz time!

```
template<typename_1 T>
struct S: typename_2 X<T>::Base {
    S(): typename_3 X<T>::Base(typename_4 X<T
        >::Base(0)) {}
    typename_5 X<T> f() {
        typename_6 X<T>::C * p;
        X<T>::D * q;
    }
    typename_7 X<int>::C * s;
};
struct U {
    typename_8 X<int>::C * pc;
};
```

Отговор

1. декларация на параметър на шаблона
2. грешно - списък на базови класове
3. грешно - инициализиращ списък
4. необходимо - създаване на обект от X<T>::Base
5. грешно - X<T> не е квалифицирано
6. задължително, ако декларираме указател
7. опционално - това е независимо име
8. грешка - извън шаблон

Съдържание

Необходимостта от template

```
template<int N>
void printBitset (std::bitset<N> const& bs)
{
    cout << bs.template to_string<char,
        char_traits<char>,
                                    allocator<
                                        char>
                                    >();
}
// bs е зависимо име и компилаторът не знае
   дали < преди
// char е operator< или скоба за шаблонни
   аргументи
// разбира се има p->template to_string< ...
```

Съдържание

```
typedef vector<vector<int>> Matrix;
// >> operator>>
typedef vector<vector<int> > Matrix;
// някой компилатори вече се спряват с това
// формално нарушавайки стандарта
// token-ите трябва да са максимално дълги
// в новия стандарт ще се разрешава
```

<:

```
class A;
namespace B {
typedef vector<::A> Matrix;
// <: e [
}

namespace B {
typedef vector< ::A> Matrix;
}
```

References

- Inside C++ Object Model, chapter 7.1
- Effective C++, Items 42, 43
- More Exceptional C++, Items 5, 10
- Modern C++ Design: Generic Programming and Design Patterns Applied
  - НЕ трябва да знаете какво пише в книгата!
  - просто я прехвърлете, ако искате да видите наистина полезни приложения на шаблони