Abusing Templates

(More like using them for non-obvious things)

Vladimír Štill

paradise.fi.muni.cz/~xstill/darcs/AbusingTemplates

4. 11. 2013

C++ šablony

- parametrizované datové typy a funkce
- parametrizace:
  - typovou proměnnou (type template parameter)
  - hodnotou (non-type template parameter), jen primitivní hodnoty
  - typový konstruktorem (template template parameter)
- u šablonovaných typů nutné explicitně uvést parametry
- u funkcí se parametry mohou (částečně) odvodit

```
template< typename TypeParameter >
void f( TypeParameter x ) { x.doSomething(); }

template< int nonTypeParameter, typename AnotherParam >
int g( AnotherParam ) { return nonTypeParameter + 1; }

int gg() { return g< 1 >( 2 ); } // calls g< 1, int >( 2 )

template< template< typename > class TemplateParam >
void h( TemplateParameter< int > intContainer ) { }
```

Instanciace šablon

- šablona je jen předpis, skutečný kód se generuje až pro konkrétní hodnoty parametrů (pro každou použitou hodnotu šablonového parametru zvlášť), při kompilaci
- tato instanciace je typicky implicitní, tedy nastává při použití šablony
- u šablon typů se implicitně instancijují jen potřebné metody, ne všechny

```
template< typename T > struct Test {
    T t;
    int f() { return 1; }
    int g() { return t.g(); }
};

struct A { int g() { return 2; } };
struct B { };

int main( void ) {
    Test< A > a; Test< B > b;
    a.f(); a.g();
    b.f();
    return 0;
}
```

Přetěžování funkcí

Nešablonovaný parametr podléhá implicitním konverzím:

```
#include <iostream>
void f( int ) { std::cout << "int, "; }
void f( void * ) { std::cout << "void *, "; }

int main() {
    int a; void *b; long c;
    f( a ), f( b ), f( c );
    // outputs: int, void *, int
}
```

V přítomnosti šablonované verze mají nešablonované přednost, pokud typy přesně odpovídají. Šablonovaná funkce má přednost před konverzí:

```
#include <iostream>
void f( int ) { std::cout << "int, "; }
void f( void * ) { std::cout << "void *, "; }
template< typename T >
void f( T ) { std::cout << "T, "; }

int main() {
    int a; void *b; long c;
    f( a ), f( b ), f( c );
    // outputs: int, void *, T
}
```

Zakázání konverze

Tyto vlastnosti šablon můžeme použít k zakázání implicitní konverze

```
#include <iostream>
void f( int ) { std::cout << "int, "; }
void f( void * ) { std::cout << "void *, "; }
template< typename T >
void f( T ) = delete;

int main() {
    int a; void *b; long c;
    f( a ), f( b ), f( c );
}
```

Nezkompiluje se:

```
overload2.cpp:9:21: error: call to deleted function 'f'
    f( a ), f( b ), f( c );
                    ^

overload2.cpp:5:6: note: candidate function [with T = long]
    has been explicitly deleted
void f( T ) = delete;
     ^

overload2.cpp:2:6: note: candidate function
void f( int ) { std::cout << "int, "; }
     ^

...
```

SFINAE: příklad

Šablonované varianty funkcí lze zakázat v závislosti na parametru, nevstupují pak vůbec do rozhodování:

```
#include <iostream>
#include <type_traits>
void f( int ) { std::cout << "int, "; }

template< typename T >
auto f( T ) -> typename
    std::enable_if< !std::is_same< T, long >::value >::type
{ std::cout << "T, "; }


int main() {
    int a; bool b; long c;
    f( a ), f( b ), f( c );
}
```

Výstup: `int, T, int,`

SFINAE: motivace, fungování

Motivace

- Chceme volit různé verze funkcí v závislosti na daných podmínkách
- Podmínky v závislosti na typových i netypových parametrech funkce (vnořený typ, typová ekvivalence, existence metody...)

Fungování

- **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror
- pokud při substituci konkrétních parametrů dojde k tomu, že v typu funkce vznikne špatně utvořený výraz, funkce se deaktivuje
- pokud existuje více verzí funkce a zůstane právě jedna nejspecifičtější aktivní pak se tato zvolí
- pokud je více aktivních nejspecifičtějších funkcí, je to chyba
- podpora v C++11 knihovně, hlavičkový soubor `<type_traits>`
- díky `decltype` je možné v typu použít libovolný výraz, ne jen typový
- typicky se podmínky zapisují do návratového typu, pro přehlednost je lepší nový (C++11) způsob zápisu typu funkce
- tělo funkce se v úvahu nebere!

SFINAE: příklad II

```
template< typename T >
auto doSomething( T &t ) -> decltype( t.doSomething() )
{ return t.doSomething(); }
template< typename T >
int doSomething( T &t ) { return 0; }

struct A { long doSomething() { return 1; } };
int main() {
    A a;
    doSomething( a );
    return 0;
}
```

Není možné zkompilovat:

```
sfinae0.cpp:15:5: error: call to 'doSomething' is ambiguous
    doSomething( a );
    ^~~~~~~~~~~
sfinae0.cpp:3:6: note: candidate function [with T = A]
auto doSomething( T &t ) -> decltype( t.doSomething() ) {
     ^
sfinae0.cpp:7:5: note: candidate function [with T = A]
int doSomething( T &t ) {
    ^

1 error generated.
```

SFINAE: problém

Problém

- pokud existuje víc nejspecifičtějších funkcí, které zůstanou při řešení přetěžování, volání je nejednoznačné a způsobí chybu při kompilaci
- řešením by mohlo být uvádět podmínky tak, aby vždy platila nejvýše jedna
- to by však způsobovalo opakování kódu → nepřehlednost, špatná udržovatelnost
- navíc některé podmínky nelze negovat (`decltype`)

Řešení

- Prioritizace funkcí s využitím následujícího pravidla: *Pokud existují 2 verze funkce takové, že se liší počtem implicitních konverzí, které jsou nutné k dosazení parametrů, pak má předost funkce kde je počet konverzí nižší.*
- vyžaduje přidat další parametr/parametry
- umožňuje zavést uspořádání na funkcích podle preference → vždy se zavolá první funkce, která není deaktivovaná, v daném pořadí
- typické řešení z internetu používá implicitní konverze mezi číselnými typy, to však působí nejasně a magicky

SFINAE: příklad III

```
#include <iostream>
// types used to prioritize function overloads
struct Preferred { };
struct NotPreferred { NotPreferred( Preferred ) { } };

template< typename T >
auto doSomething( T &t, Preferred )
    -> decltype( t.doSomething() )
{
    return t.doSomething();
}

template< typename T >
void doSomething( T &t, NotPreferred ) {
    std::cout << "general";
}

struct A { void doSomething() { std::cout << "A, "; } };
int main() {
    A a; int b;
    doSomething( a, Preferred() );
    doSomething( b, Preferred() );
    return 0; // outputs: A, general
}
```

SFINAE: konkrétní problém

Problém

- Máme hierarchii tříd, které plní danou funkčnost (reportování výsledků v DIVINE). Tato hierarchie je postavená na virtuálních voláních. Kořenem hierarchie je třída `Report`, která udává rozhraní.
- `std::shared_ptr< Report > report`
- report chceme vytvářet pomocí statické factory metody
  - parametrizovaná typem požadovaného reportu
  - typicky bude konstruovat `shared_ptr` na objekt požadovaného typu
  - ve specifických případech chceme provést nějakou akci nad rámec konstruktoru daného typu
  - například chceme umožnit, že v určitých případech nepůjde daný typ reportu zkonstruovat – chceme vracet nulový ukazatel

Naivní řešení

- Do každé třídy implementující `Report` dáme statickou metodu `get`. To nám zároveň zajistí parametrizaci.
- Voláme něco jako `TextReport::get()`.
- **Nevýhoda:** většina get metod je téměř stejná, jen volá dynamickou alokaci příslušné třídy.

SFINAE: konkrétní problém a řešení

Řešení pomocí šablon a SFINAE

- factory metodu umístíme do bázové třídy `Report`
- parametrizovaná požadovaným typem reportu
- pokud budeme chtít překrýt, umístíme do příslušné třídy statickou metodu `get`, která bude konstruovat objekt této konkrétní třídy
- bázová faktory metoda bude volat specifickou `get` metodu požadovaného potomka pokud tato existuje
- jinak provede alokaci a konstrukci potomka

SFINAE: konkrétní problém a řešení (zjednodušené)

```
struct Report {
    template< typename Rep, typename T >
    static std::shared_ptr< Report > get( T t ) {
        return _get< Rep >( wibble::Preferred(), t );
    }

    template< typename Rep, typename T >
    static auto _get( wibble::Preferred, T t ) ->
        decltype( Rep::get( t ) )
    { return Rep::get( t ); }

    template< typename Rep, typename T >
    static auto _get( wibble::NotPreferred, T t )
        -> std::shared_ptr< Report >
    { return std::make_shared< Rep >( t ); }
};

struct SqlReport : Report {
#ifndef O_SQL_REPORT
    template< typename X >
    static std::shared_ptr< Report > get( X )
    { return nullptr; }
#endif
};
```

Variadické šablony

Variadické funkce v C

- C umožňuje mít funkce s volitelným počtem argumentů, jako například `printf`
- nulová typová bezpečnost – kontrola při kompilaci naprosto chybí, nebo vyžaduje aby kompilátor znal příslušnou funkci
- možné používat i v C++98, ale jen v omezené míře, na jednoduché datové typy (bez konstruktorů, virtuálních metod,...)

Variadické šablony (C++11)

- více parametrů můžeme zastoupit šablonou pro *argument pack*
- typicky buď přímo předáváme všechny argumenty do jiné funkce, nebo postupně rekurzivně rozbalujeme
- odvození typů z použitých argumentů
- statická typová kontrola a volba funkce

Variadické šablony – příklad

```
MDNode *node( MDNode *root ) { return root; }

template< typename N, typename... Args >
MDNode *node( N *root, int n, Args... args ) {
    if ( root )
        return node(
            cast< MDNode >( root->getOperand(n) ),
            args... );
    return nullptr;
}
```

- z LLVM interpretru
- `MDNode` tvoří (n-ární) stromovou strukturu, pomocí funkce `node` můžeme snadno získat uzel v tomto stromě
- `node( root, 1, 4, 3, 0 )`
- při kompilaci se ověří, že všechny argumenty jsou typu `int` (nebo konvertovatelné)

Perfect forwarding

Motivace

- umožňuje nám předat parametry funkce do jiné funkce tak jak jsou, bez kopírování
- typicky s použitím variadických šablon
- předávání parametrů konstruktoru přes factory metodu,...
- `std::vector< T >::emplace_back()`
- využívá rvalue reference, `std::forward< T >()`
- rvalue reference = reference na dočasnou hodnotu

```
template< typename Rep, typename... Ts >
static std::shared_ptr< Report > get( Ts &&... ts ) {
    return _get< Rep >( wibble::Preferred(),
                        std::forward< Ts >( ts )... );
}
```

declcheck

- někdy chceme uvnitř `decltype` ověřit platnost více výrazů
- nebo chceme ověřit jen jejich platnost, ale návratový typ nás nezajímá
- definujeme speciální funkci, řekněme `declcheck` – bere libovolné parametry libovolných typů, vrací požadovaný typ
- **pozor:** pokud se nám v `declcheck` výrazu objeví výraz vracející void, je celý výraz neplatný! → můžeme používat typ, který nese jedinou hodnotu namísto `void`, například `Unit` v příkladě

```
struct Unit { };

template< typename... X >
static Unit declcheck( X... ) { return Unit(); }

template< typename X = int >
auto doSomeArithmetic( X &r, X &a, X &b )
    -> decltype( declcheck( a + b, a % b, std::fmod( a, b ) ) )
{ /* ... */ }
```

Singleton typy

- šablonované funkce dokáží odvodit šablonové parametry
- jen parametry, které lze odvodit z typů parametrů funkce
- tedy nefunguje přímo pro hodnotové šablonové parametry, typové parametry nepoužité v parametrech funkce
- někdy nemůžeme explicitně funkci předat typové parametry
  - konstruktor
  - více přetížených variant s různými typovými parametry
- je možné používat *type witness*
- často typ, který nenese žádnou významnou run-time hodnotu, má smysl jen při odvození typu – singleton typ
- pro číselné konstanty lze použít `std::integral_constant`
- type witness lze snadno předávat (typicky má triviální bezparametrický konstruktor, copy-konstruktor)

Singleton typy – příklad

```
#include <type_traits>
template< typename T > struct Witness { };

struct A {
    template< typename T >
    A( Witness< T > ) { /* ... */ }
};

template< int val >
int getX() { return val; }

template< int val >
int getY( std::integral_constant< int, val > ) {
    return val;
}
template< int val >
int getZ( std::integral_constant< int, val > singleton ) {
    return getY( singleton );
}
int main( void ) {
    A a{ Witness< long >() }; // new ctor call syntax needed
    getX< 1 >();
    return getZ( std::integral_constant< int, 5 >() );
}
```

The Dark Side

Instanciace komponent v DIVINE

Problém

- v DIVINE je mnoho komponent, které jsou z důvodu výkonu staticky spojované při kompilaci pomocí šablon
- je třeba vybrat správnou kombinaci v závislosti na vstupu
- existuje mnoho kombinací (cca 3000)
- ne všechny kombinace dávají smysl (jen cca 900 smysluplných, některé nesmyslné kombinace nelze ani zkompilovat)
- počet kombinací značně roste s tím, jak se DIVINE rozšiřuje

Původní řešení

- instanciace probíhala od algoritmu, ve fixním pořadí a pro každý typ komponenty byla samostatná funkce kódující výběr
- špatně se řešili nesmyslné volby, jedině ad-hoc řešení
  $\rightarrow$ spousta zbytečných instancí
- instanciace startuje z fixního množství souborů, odpovídá počtu algoritmů
  $\rightarrow$ kompilace jednoho souboru začala vyžadovat mnoho paměti ($> 8GB$)

současné řešení

- založené na automaticky generovaném kódu pro kompilaci instancí
- každá komponenta popsána strukturou obsahující:
  - funkci pro výběr
  - funkci provedenou po výběru
  - hlavičkový soubor
  - podmínky aktivace (s ohledem na ostatní komponenty)
- selekční mechanismus vytvořený pomocí šablon
  - umožňuje projít celý strom selekce (pro generování)
  - nebo vybrat jedinou položku
- podmínky aktivace vyhodnoceny při kompilaci selekčního algoritmu
  - v závislosti na již vybraných typech
- pořadí typů komponent k selekci a pořadí komponent v rámci typu dáno seznamem typů

- `divine/toolkit/typelist.h`
- `divine/instances/definitions.h`
- `divine/instances/select-impl.h`

TypeList

- singleton typ, seznam libovolného počtu typů
- zná svou délku, snadná iterace
- základní funkce pro manipulaci s typovými seznamy
- kóduje komponenty
- kóduje rovněž podmínky instanciace

```
#include <type_traits>

template< typename... >
struct TypeList {
    using Empty = wibble::Unit;
    static constexpr size_t length = 0;
};

template< typename _T, typename... _Ts >
struct TypeList< _T, _Ts... > : private TypeList< _Ts... > {
    using Head = _T;
    using Tail = TypeList< _Ts... >;
    static constexpr size_t length = 1 + Tail::length;
};
```

```
template< typename X, typename EmptyList >
struct Append { using T = TypeList< X >; };

template< typename _X, typename _T, typename... _Ts >
struct Append< _X, TypeList< _T, _Ts... > > {
    using T = TypeList< _X, _T, _Ts... >;
};

template< template< typename > class Cond, typename List >
struct Filter : public
    std::conditional< Cond< typename List::Head >::value,
        Filter< Cond, typename List::Tail >,
        Append< typename List::Head,
            typename Filter< Cond, typename List::Tail >::T >
    >::type { };

template< template< typename > class Cond >
struct Filter< Cond, TypeList<> > { using T = TypeList<>; };

template< typename List >
struct Last : public Last< typename List::Tail > { };

template< typename _T >
struct Last< TypeList< _T > > {
    using T = _T;
};
```

Definice komponent

- pomocí C maker jsou zadefinovány struktury pro příslušné komponenty
- seznam komponent je v `TypeList`

```
#define STORE( STOR, SELECTOR, SUPPORTED_BY ) struct STOR { \
    using IsStore = IsStoreT; \
    static constexpr const char *symbol = \
        "template < typename Provider , typename Generator , typenam
        "using _Store = ::divine::visitor::" #STOR "< Provider , Ge
    static constexpr const char *key = #STOR; \
    using SupportedBy = SUPPORTED_BY; \
    static bool select( Meta &meta ) { \
        return SELECTOR; \
        static_cast< void >( meta ); \
    } \
}

using ForCompressions =
    Not< Or< algorithm::Info, algorithm::Simulate > >;
STORE( NTreeStore, meta.algorithm.compression == Tree,
        ForCompressions );
STORE( DefaultStore, true, Any );
// ...
using Stores = TypeList< NTreeStore, HcStore, DefaultStore >;
```

Selektor

- jádro systému
- postupně prochází seznamy komponent a z nich vybírá dostupné
- příliš dlouhé na ukázku, pro příklad přetížená varianta funkce, která se volá pokud nedošlo k chybě instanciace ale komponenta je povolená

```
// one more overload before ...
template< typename Selector, typename Head, typename Tail,
    typename Default, typename ToSelect, typename Selected >
auto runIfValid( Selector &sel, Head, Tail, Default,
        ToSelect, Selected, Preferred, NotPreferred )
    -> typename std::enable_if<
        ! std::is_base_of< InstantiationError, Head >::value
        && EvalBoolExpr< ContainsP< Selected >::template Elem,
            typename Head::SupportedBy >::value,
        typename Selector::ReturnType >::type
{
    return sel.ifSelect( Head(), [ &sel ]() {
            return runSelector( sel, ToSelect(),
                typename Append< Head, Selected >::T() );
        }, [ &sel ]() {
            return runOne( sel, Tail(), Default(),
                ToSelect(), Selected() );
        } );
} // and one after ...
```

zhodnocení současného řešení

Klady

- je možné specifikovat snadno kolik instancí se kompiluje najednou
- menší nároky na paměť při kompilaci
- přesné vyjádření požadavků pro platnou kombinaci
- snadno rozšířitelné o další komponenty
- lepší využití paralelní kompilace

Zápory

- s počtem komponent roste doba kompilace samotného generátoru
- kompilace je pomalejší na stroji s dostatkem paměti pro původní verzi
- vyžaduje samostatný krok v buildu který zkompiluje a spustí generátor

Závěr

Děkuji za pozornost

Pokud vás problematika zaujala...

- cppreference.com je vhodným zdrojem pro další studium
- Type Support a hlavičkový soubor `<type_traits>`
- Typové šablony a jejich specializace + částečná specializace
- Funkční šablony
- Proč nespecializovat funkční šablony
- zdrojové soubory DIVINE :-)