Metaprogramowanie w C++

Na podstawie książki A. Alexandrescu “Nowoczesne projektowanie w C++”

Marcin Świderski
sfider@students.mimuw.edu.pl

Metaprogramowanie

Metaprogramowanie to pisanie programów komputerowych, które modyfikują lub generują kod innego programu, bądź też wykonują część pracy w trakcie kompilacji.

W C++ metaprogramowanie jest możliwe do zrealizowania za pomocą mechanizmu programowania uogólnionego, jakim są szablony.

O czym będzie mowa?

- Klasy parametryzowane wytycznymi
- Inteligentne wskaźniki
- Programowanie statyczne
- Funktory uogólnione
- Implementacja singletonu
- Fabryki obiektów
- Fabryki abstrakcyjne

Wytyczna

- Implementuje fragment funkcjonalności projektowanej klasy
- Jest oparta na składni, a nie na sygnaturze
- Potrafi wzbogacić projektowaną klasę o dodatkowe funkcjonalności
- Umożliwia modyfikację struktury klasy
- W połączeniu z innymi wytycznymi tworzy wykładniczą ilość wersji projektowanej klasy

Wytyczna `Creator`

Implementacje wytycznej **`Creator`**:

```
template< class T > struct OpNewCreator {
  static T* create() {
    return new T;
  }
};


template< class T > struct MallocCreator {
  static T* create() {
    void* buf = std::malloc(sizeof(T));
    if (!buf) return 0;
    return new(buf) T;
  }
};
```

Wzbogacona wytyczna Creator

```
template< class T > struct PrototypeCreator {
  PrototypeCreator(T* obj = 0) : prototype_(obj) {}

  T* create() {
    return prototype_ ? prototype_->clone() : 0;
  }

  T* getPrototype() {
    return prototype_;
  }

  void setPrototype(T* obj) {
    prototype_ = obj;
  }

private:
  T* prototype_;
};
```

Korzystanie z wytycznych

```
// Kod biblioteki
template<
  template <class> class CreationPolicy = OpNewCreator
> class WidgetManager : public CreationPolicy< Widget > {
  ...
};


// Kod użytkownika
typedef WidgetManager< MallocCreator > MyWidgetManager;
```

Destruktory wytycznych

```
template< class T > struct OpNewCreator {
  static T* create() {
    return new T;
  }

protected:
  ~OpNewCreator() {...}
};
```

- Chroniony, aby jedynie klasa-gospodarz mogła niszczyć obiekt wytycznej
- Niewirtualny, aby zminimalizować narzut czasowy i pamięciowy

Inteligentne wskaźniki

- Spełniają rolę analogiczną do zwykłych wskaźnikiów
- W C++ mogą udawać zwykłe wskaźniki syntaktycznie
- Potrafią zniszczyć wskazywany obiekt
- Wiedzą kiedy zniszczyć wskazywany obiekt
- Potrafią pilnować swojej integralności
- Mogą, ale nie muszą, być konwertowane do zwykłych wskaźników

Inteligentne wskaźniki

Rozkład na wytyczne ortogonalne:

- **`Storage`**:
  - Opisuje strukturę inteligentnego wskaźnika
  - **`DefaultSPStorage, ArrayStorage, LockedStorage, HeapStorage`**
- **`OwnerShip`**:
  - Definiuje moment niszczenia obiektu
  - **`DeepCopy, RefCounted, RefCountedMT, COMRefCounted, RefLinked, DestructiveCopy, NoCopy`**

Inteligentne wkaźniki

Rozkład na wytyczne ortogonalne c.d.:

- **`Conversion`**:
  - Definiuje, czy dozwolona jest konwersja inteligentnego do zwykłego wskaźnika
  - **`AllowConverion, DisallowConversion`**
- **`Checking`**:
  - Mechanizm kontroli integralności
  - **`AssertCheck, AssertStrictCheck, RejectNullStatic, RejectNull, RejectNullStrict, NoCheck`**

Programowanie statyczne

Jest to termin, którego A. Alexandrescu używa w swojej książce w odniesieniu do programowania za pomocą szablonów.

Programowanie statyczne wykorzystuje szablony do tworzenia pewnego rodzaju funkcji oraz funkcji rekurencyjnych, których argumentami oraz wartościami zwracanymi mogą być statycznie wyliczane stałe (np. liczby całkowite) oraz typy.

Stała całkowita -> Typ

```
template< int v > struct Int2Type {
    enum { value = v }
};


template< class T, bool isPolymorphic >
class NiftyContainer {
private:
    void doSth(T* obj, Int2Type< true >) {
        T* newObj = obj->clone();
    }
    void doSth(T* obj, Int2Type< false >) {
        T* newObj = new T(*obj);
    }
public:
    void doSomething(T* obj) {
        doSth(obj, Int2Type< isPolimorphic >());
    }
};
```

Typ -> Typ

```
template< class T > struct Type2Type {
    typedef T OriginalType;
};


template< class T, class U >
T* create(const U& arg, Type2Type< T >) {
    return new T(arg);
}


template< class U >
Widget* create(const U& arg, Type2Type< Widget >) {
    return new T(arg, -1);
}
```

Wybór typu

```
template< bool flag, class T, class U >
struct Select {
  typedef T Result;
};

template< class T, class U >
struct Select< false, T, U > {
  typedef U Result;
};

template< class T, bool isPolymorphic >
class NiftyContainer {
  ...
  typedef
    typename Select< isPolymorphic, T*, T >::Result
    ValueType;
  ...
};
```

Ciąg Fibonacciego

```
template<> struct Fib;

template<> struct Fib< 0 > {
  enum { value = 1 };
};

template<> struct Fib< 1 > {
  enum { value = 1 };
};

template< int f > struct Fib {
  enum { value = Fib< f - 1 >::value
               + Fib< f - 2 >::value; };
};
```

Inne sztuczki

- Statyczne wykrywanie dziedziczenia i możliwości konwersji
- Opakowanie dla **`type_info`** – nadaje semantykę wartości
- **`NullType`** – wartość kompletnie nieinteresująca
- **`EmptyType`** – dobry domyślny argument szablonu
- Cechowanie typów

Listy typów

- Pozwalają na powielenie kodu dla wielu typów
- Umożliwiają implementację np. Fabryki Abstrakcyjnej

```
template< class T, class U > struct TypeList {
  typedef T Head;
  typedef U Tail;
};


#define TYPELIST_1(T1) TypeList< T1, NullType >
#define TYPELIST_2(T1, T2) TypeList< T1, TYPELIST_1(T2) >
...
#define TYPELIST_50(...)...
```

Listy typów: obliczanie długości

```
template< class TList > struct Length;


template<> struct Length< NullType > {
  enum { value = 0 };
};


template< class T, class U >
struct Length< TypeList< T, U > > {
  enum { value = 1 + Length< U >::value };
};
```

Listy typów: indeksowanie

```
template< class Head, class Tail >
struct TypeAt< TypeList< Head, Tail >, 0> {
  typedef Head Result;
};

template< class Head, class Tail, unsigned int i >
struct TypeAt< TypeList< Head, Tail >, i > {
  typedef
    typename TypeAt< Tail, i - 1 >::Result
    Result;
};
```

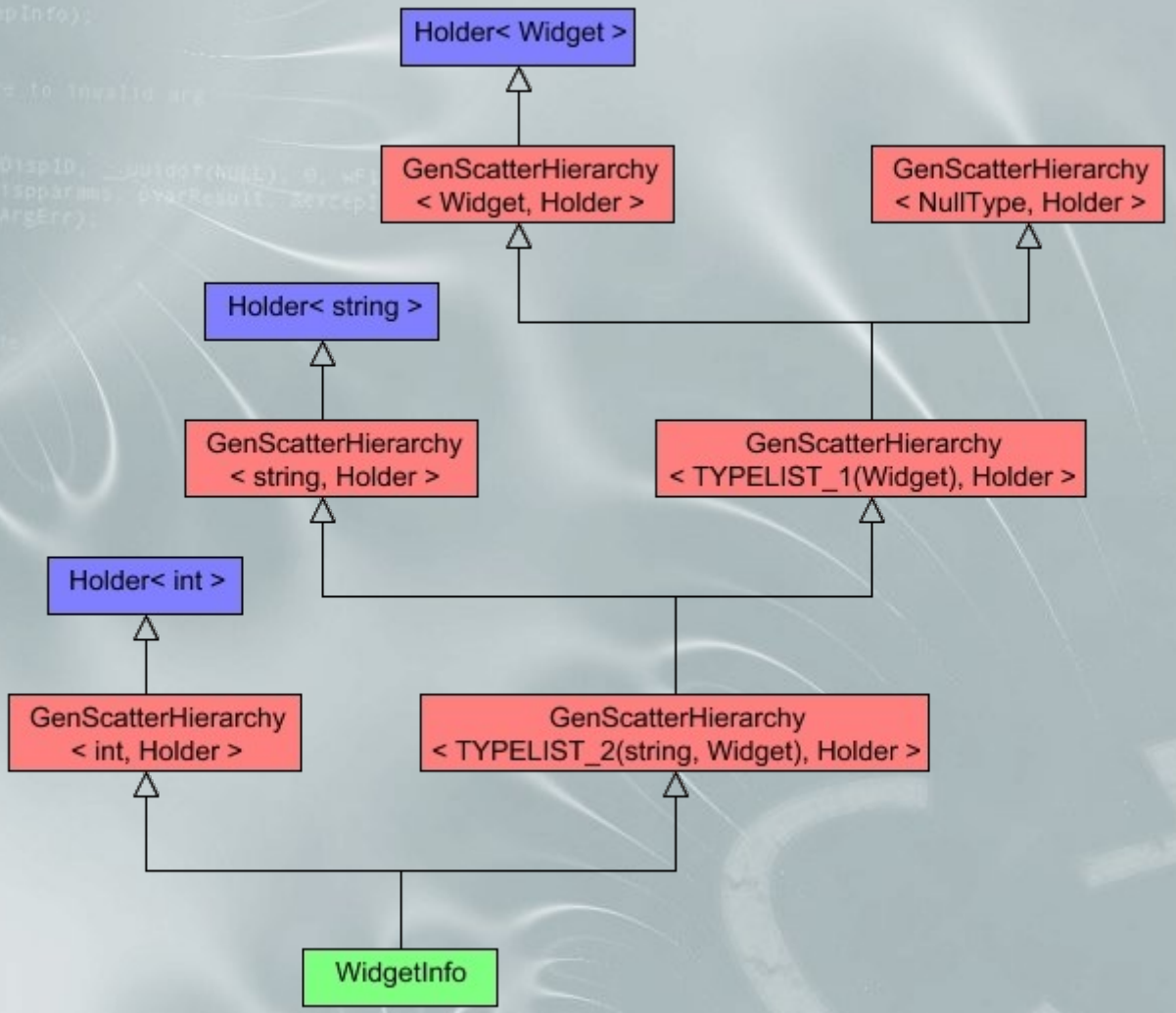
Operacje na listach typów

- `Length< TList >`
- `TypeAt< TList, idx >`
- `TypeAtNonStrict< TList, idx >`
- `IndexOf< TList, T >`
- `Append< TList, T >`
- `Erase< TList, T >`
- `EraseAll< TList, T >`
- `NoDuplicates< TList >`
- `Replace< TList, T, U >`
- `ReplaceAll< TList, T, U >`
- `MostDerived< TList, T >`
- `DerivedToFront< TList >`

Generowanie herarichii rozległych

```
template< class TList, template< class > class Unit >
class GenScatterHierarchy;


template< class T1, class T2, template< class > class Unit >
class GenScatterHierarchy< TYPELIST_2(T1, T2), Unit>
  : public GenScatterHierarchy< T1, Unit >
  , public GenScatterHierarchy< T2, Unit > {};


template< class AtomicType, template< class > class Unit >
class GenScatterHierarchy< AtomicType, Unit >
  : public Unit< AtomicType > {};


template< template< class > class Unit >
class GenScatterHierarchy< NullType, Unit > {};
```

Hierarchia rozległa



Generowanie krotek

```
template< class T > struct Holder {
  T value_;
};


typedef GenScatterHierarchy<
  TYPELIST_4(int, string, float, bool), Holder >
  MixedTuple;


MixedTuple tuple;
...
int i = Field< 0 >(tuple).value_;
string s = Field< 1 >(tuple).value_;
float f = Field< 2 >(tuple).value_;
bool b = Field< 3 >(tuple).value_;
```

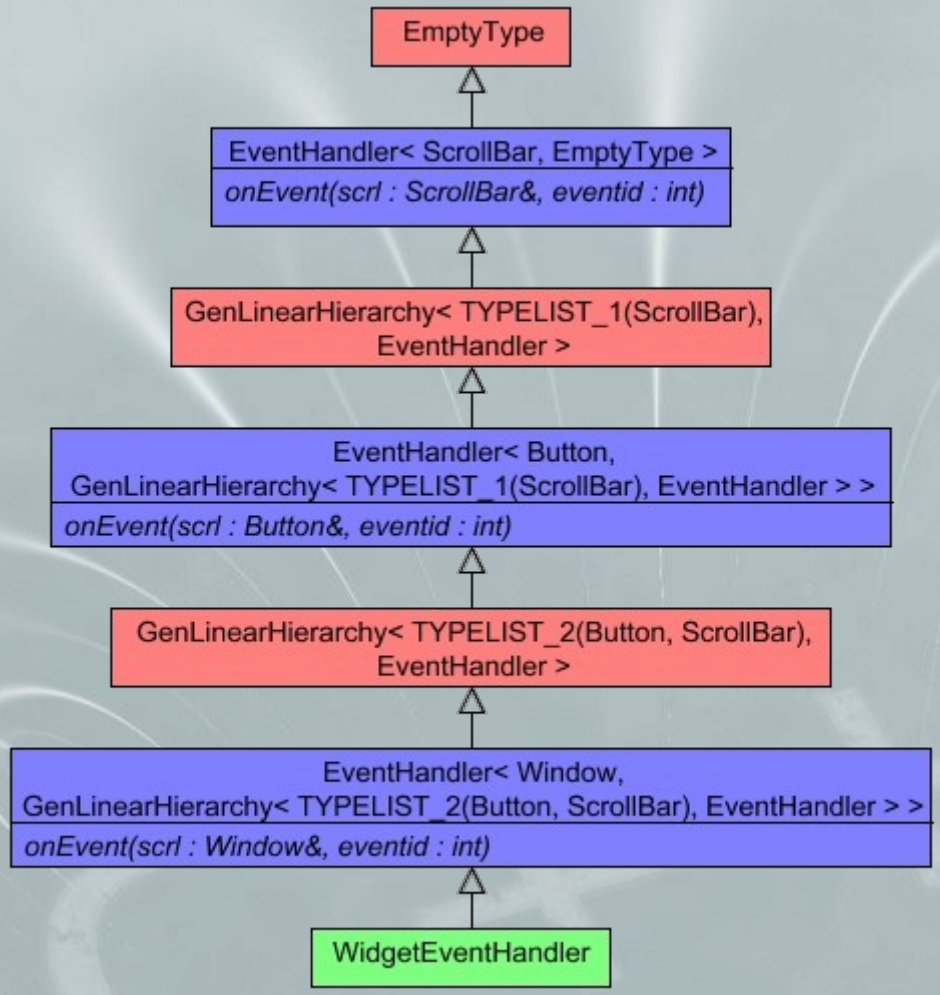
Generowanie hierarchii liniowych

```
template<
  class TList,
  template< class AtomicType, class Base > class Unit,
  class Root = EmptyType
> class GenLinearHierarchy;

template<
  class T1, class T2,
  template< class, class > class Unit,
  class Root
> class GenLinearHierarchy< TypeList< T1, T2 >, Unit, Root >
  : public Unit< T1, GenLinearHierarchy< T2, Unit, Root > >
  {};

template<
  class T, template< class, class > class Unit,
  class Root
> class GenLinearHierarchy< TYPELIST_1(T), Unit, Root >
  : public Unit< T, Root > {};
```

Hierarchia liniowa

```
template< class T, class Base >
class EevntHandler : public Base {
public:
  virtual void onEvent(
       T& obj, int eventId);
};
```

```
typedef GenLinearHierarchy<
    TYPELIST_3(Window, Button,
        ScrollBar),
    EventHandler >
  WidgetEventHandler;
```



Funktory uogólnione

- Kapsułkują dowolne polecenie przetwarzania (funkcję, funkcję składową, funktor, funktor uogólniony)
- Bezpieczne typologicznie
- Semantyka wartości
- Ułatwiają implementację wzorca projektowego Polecenie
- Pozwalają na wiązanie argumentów oraz łączenie wielu funktorów sekwencyjnie (domknięcia?)

Funktory uogólnione - przykłady

```
void function(int);
struct SomeFunctor { void operator()(int); };
struct SomeClass { void memberFunction(int); };

// inicjowanie funkcją
Functor< void, TYPELIST_1(int) > cmd1(function);
// inicjowanie funktorem
SomeFunctor fn;
Functor< void, TYPELIST_1(int) > cmd2(fn);
// inicjowanie wskaźnikiem obiektu wraz ze wskaźnikiem do
// funkcji składowej
SomeClass obj;
Functor< void, TYPELIST_1(int) > cmd3(&obj,
  &SomeClass::memberFunction);
// inicjowanie innym funktrem uogólnionym
Functor< void, TYPELIST_1(int) > cmd4(cmd3);
```

Funktory uogólnione - przykłady

Wiązanie argumentów:

```
// definiujemy funktor trójargumentowy
Functor< void, TYPELIST_3(int, int, double) > cmd1(f);
// wiążemy 10 z pierwszym argumentem
Functor< void, TYPELIST_2(int, double) > cmd2(
  BindFirst(cmd1, 10));
// to samo, co cmd1(10, 20, 5.6)
cmd2(20, 5.6);
```

Łączenie sekwencyjne funktorów:

```
Functor<> cmd1(f1);
Functor<> cmd2(f2);
// połącz cmd1 i cmd2
Functor<> cmd3(Chain(cmd1, cmd2);
```

Implementacja singletonu

```
template<
  class T,
  template< class > class CreationPolicy = CreateUsingNew,
  template< class > class LifetimePolicy = DefaultLifetime,
  template< class > class ThreadingModel = SingleThreaded
> class SingletonHolder;
```

- Szablon **`SingletonHolder`** zarządza jedynie cyklem życia obiektu, który sam musi być singletonem
- Tworzenie: **`CreateUsingNew`**, **`CreateStatic`**, **`CreateUsingMalloc`**,
- Cykl życia: **`DefaultLifetime`**, **`NoDestroy`**, **`PhoenixSingleton`**, **`SingletonWithLongevity`**
- Wielowątkowość: **`SingleThreaded`**, **`ClassLevelLockable`**

Fabryka obiektów

```
template<
  class AbstractProduct,
  class IdentifierType,
  class ProductCreator = AbstractProduct*(*)(),
  template< class, class >
    class FactoryErrorPolicy = DefaultFactoryError
> class Factory;
```

- Fabryka tworzy obiekty na podstawie identyfikatorów typów (np. w celu serializacji)
- Każda klasa w hierarchii, dla której utworzona jest fabryka musi sama się w fabryce zarejestrować
- Odmianą fabryki obiektów jest fabryka klonów, w której kluczem jest typ obiektu (RTTI)

Fabryka abstrakcyjna

- Fabryka abstrakcyjna służy do tworzenia rodzin powiązanych lub zależnych od siebie obiektów polimorficznych
- Implementacja fabryki abstrakcyjnej korzysta z list typów do przekazania listy produktów
- Fabrykę abstrakcyjną implementują dwa szablony: **`AbstractFactory`**, **`ConcreteFactory`**

Szablon `AbstractFactory`

```
template<
    class TList,
    template< class > class Unit = AbstractFactoryUnit
> class AbstractFactory;
```

- Konkretyzacja **`AbstractFactoryUnit< T >`** definiuje funkcję czysto wirtualną

  `T* DoCreate(Type2Type< T >)`

- **`AbstractFactory`** udostępnia szablon szablon funkcyjny składowy **`Create`**, który można skonkretyzować dla dowolnego z typów produktów fabryki abstrakcyjnej

```
Soldier *soldier = factory->Create< Soldier >();
```

Szablon ConcreteFactory

```
template<
  class AbstractFact,
  template< class, class >
    class FactoryUnit = OpNewFactoryUnit,
  class TList = AbstractFactory::ProductList
> class ConcreteFactory;


typedef ConcreteFactory<
  AbstractEnemyFactory,
  OpNewFactoryUnit,
  TYPELIST_3(SillySoldier, SillyMonster, SillySuperMonster)
> EasyLevelEnemyFactory;


typedef ConcreteFactory<
  AbstractFactory,
  PrototypeFactoryUnit
> EnemyFactory;
```

Pominięte

- Odwiedzający – wariant cykliczny i acykliczny
- Wielometody – implementacja statyczna liniowa, oraz dwie dynamiczne: logarytmiczna nieintruzyjna, stała intruzyjna

Źródła

- “Nowoczesne projektowanie w C++” A.Alexandrescu
- Strona A.Alexandrescu: http://erdani.org
- Strona biblioteki Loki: http://loki-lib.sourceforge.net
- Wikipedia :P

Dziękuję i zapraszam do dyskusji