

Szablony

Zagnieżdżanie

212

Szablony – rozszerzenie

Zagnieżdżanie

Przyjmijmy, że mamy szablony klasy, który ma jeden parametr:

```
template<typename T>
class MojaKlasa
```

Przyjmijmy, że wewnątrz tej klasy, która została podana jako parametr szablonu, jest zdefiniowany inny, abstrakcyjny typ danych, tj. klasa, np.:

```
class Pierwsza {
public:
    class Druga {
public:
        void g() {}
    };
};
```

Problem: w definicji szablonu klasy potrzebujemy odwołać się do typu zdefiniowanego w klasie podanej jako parametr (np. utworzyć pole, a potem skorzystać z jednej z metod).

© UKSW, WMP, SNS, Warszawa 213

213

Szablony – rozszerzenie

```
class Pierwsza {
public:
    class Druga {
public:
        void lekka() {}
    };
};
template<typename T>
class MojaKlasa {
    T::Druga i;
public:
    void mocna() { i.lekka(); };
};

int main() {
    MojaKlasa<Pierwsza> MK;
    MK.mocna();
}
```

error C2146: syntax error : missing ';' before identifier 'i'

© UKSW, WMP, SNS, Warszawa 214

214

Szablony – rozszerzenie

Zagnieżdżanie – problem

Jeżeli odwołamy się do typu **Druga** używając go zgodnie z przeznaczeniem, kompilator nie będzie w stanie rozstrzygnąć, czy to faktycznie typ zadeklarowany w klasie **Pierwsza**, czy też składowa tej klasy.

Aby uczynić sytuację jednoznaczną, używamy dodatkowo słowa kluczowego **typename** wewnątrz definicji szablonu klasy.

© UKSW, WMP, SNS, Warszawa 215

215

Szablony – rozszerzenie

```
class Pierwsza {
public:
    class Druga {
public:
        void lekka() {}
    };
};
template<typename T>
class MojaKlasa {
    T::Druga i;
public:
    void mocna() { i.lekka(); };
};

int main() {
    MojaKlasa<Pierwsza> MK;
    MK.mocna();
}
```

Celem użycia słowa **typename** nie jest zadeklarowanie nowego typu, ale wskazanie kompilatorowi, że tak zakwalifikowany identyfikator ma być traktowany jako typ.

© UKSW, WMP, SNS, Warszawa 216

216

Szablony – rozszerzenie

Zagnieżdżanie – szablony składowe

Szablony składowe to szablony zadeklarowane wewnątrz innego szablonu

Przykład – szablon klasy w której konstruktor jest w postaci szablonu zagnieżdżonego:

```
template<class T> class ABC {
    template<typename X>
    ABC(const ABC<X>& x); //deklaracja konstr. konwertującego
    ...
};
```

Powyższy szablon pozwala na tworzenie np. mechanizmu konwersji typów w których obydwa zostały zbudowane z wykorzystaniem szablonu **ABC**:

```
ABC<float> z(); // ,z' to obiekt typu A<float>
ABC<double> w(z); // obiekt typu ABC<double> z ABC<float>
```

© UKSW, WMP, SNS, Warszawa 217

217

Szablony – rozszerzenie

Zagnieżdżanie – szablony składowe

Aby zdefiniować zagnieżdżony szablon konstruktora, destruktora lub metody *poza* definicją szablonu klasy zewnętrznej, musimy wyrazić fakt zagnieżdżenia w sposobie deklaracji:

```
template<typename T>
template<typename X>
Abc<T>::Abc(const Abc<X>& x) { // nagłówek konstruktora
    ...                       // kod konstruktora
};
```

© UKSW, WMP, SNS, Warszawa

218

218

Szablony – rozszerzenie

Zagnieżdżanie – szablony składowe

Szablony składowe mogą być także klasami:

```
template<typename T>
class Zewnetrzna {
public:
    template<typename R>
    class Wewnetrzna {
    public:
        void f();
    };
    ...
};
```

© UKSW, WMP, SNS, Warszawa

219

219

Szablony – rozszerzenie

Zagnieżdżanie – szablony składowe

Aby zdefiniować zagnieżdżony szablon konstruktora, destruktora lub metody *poza* definicją szablonu klasy zewnętrznej, musimy wyrazić fakt zagnieżdżenia w sposobie deklaracji:

```
template<typename T>
template<typename R>
void Zewnetrzna<T>::Wewnetrzna<R>::f() {
    ... // tutaj ciało metody f
};
```

© UKSW, WMP, SNS, Warszawa

220

220

Szablony – rozszerzenie

```
template<typename T>
class Zewnetrzna {
public:
    template<typename R>
    class Wewnetrzna {
    public:
        void f();
    };
};

template<typename T>
template<typename R>
void Zewnetrzna<T>::Wewnetrzna<R>::f() {
    cout << "Zew: " << typeid(T).name() << endl;
    cout << "Wew: " << typeid(R).name() << endl;
    cout << "Razem Wew: " << typeid(*this).name()
        << endl;
};

int main() {
    Zewnetrzna<int>::Wewnetrzna<bool> W;
    W.f();
};

W oknie konsoli:
Zew: int
Wew: bool
Razem Wew: Zewnetrzna<int>::Wewnetrzna<bool>
```

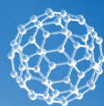
© UKSW, WMP, SNS, Warszawa

221

221

Szablony

Specjalizacja jawna



Szablony – rozszerzenie

Specjalizacja jawna szablonu funkcji

Czasem istnieje potrzeba, aby obok szablonu istniała wersja kodu dla szczególnych typów danych, różniąca się od wersji kodu w szablonie.

Należy wtedy jawnie zadeklarować specjalizację szablonu dla tego konkretnego typu (lub listy typów).

© UKSW, WMP, SNS, Warszawa

223

223

222

Szablony – rozszerzenie

Specjalizacja jawna szablonu funkcji:

```
template<typename T>
const T& min(const T& a, const T& b) {
    return (a < b) ? a : b;
};

template<>
const char* const& min<const char*>(const char* const& a,
                                     const char* const& b) {
    return (strcmp(a, b) < 0) ? a : b;
};

int main() {
    const char *s2 = "ali", *s1 = "baba";
    cout << ::min(s1, s2) << endl;
    cout << ::min<>(s1, s2) << endl;
} // obydwa odwołania funkcji min odnoszą się do tej samej specjalizacji
```

© UKSW, WMP, SNS, Warszawa

224

224

Szablony – rozszerzenie

Uwagi dotyczące specjalizacji w podanym przykładzie:

```
const char* const& min<const char*>(const char* const& a,
                                     const char* const& b)
```

- W specjalizacji zastępujemy typ **T** typem **const char***
- W pierwotnym szablonie mamy jako definicję typu wyrażenie **const T**, dlatego dokonując specjalizacji musimy słowem kluczowym **const** objąć cały typ **const char***
- .. czyli musimy wyrazić fakt, że to wskaźnik na **const char*** jest **const**, stąd piszemy **const char* const**

© UKSW, WMP, SNS, Warszawa

225

225

Szablony – rozszerzenie

Uwagi dotyczące specjalizacji w podanym przykładzie:

W funkcji main:

```
cout << ::min(s1, s2) << endl;
cout << ::min<>(s1, s2) << endl;
```

Dwa dwukropki reprezentują operator rozpoznawania zakresu. Zapis powyżej oznacza pochodzenie z zakresu globalnego, tj. funkcja nie jest zdefiniowana w żadnej **namespace**.

© UKSW, WMP, SNS, Warszawa

226

226



Szablony

Częściowe uporządkowanie szablonów klas

227

Szablony – rozszerzenie

Niejednoznaczność funkcji

Zawsze wywoływany jest szablon najlepiej wyspecjalizowany

```
template<typename T>
void f(T) { // 1.
    cout << "T" << endl;
};

template<typename T>
void f(T*) { // 2.
    cout << "T*" << endl;
};

template<typename T>
void f(const T*) { // 3.
    cout << "const T*" << endl;
};
```

© UKSW, WMP, SNS, Warszawa

228

228

```
int main() {
    f(0); // 1. T
    int i = 0;
    f(&i); // 2. T*
    const int j = 0;
    f(&j); // 3. const T*
```

Jeżeli wśród dostępnych szablonów nie da się jednoznacznie określić tego najlepiej wyspecjalizowanego, powstaje **niejednoznaczność** i kompilator zwraca błąd – stąd nazwa „częściowe uporządkowanie”

Szablony – rozszerzenie

Częściowe uporządkowanie szablonów klas

Zasady decydujące, które szablony klas będą wybierane w poszczególnych sytuacjach są podobne do zasad uporządkowania szablonów funkcji – wybierany jest szablon najbardziej wyspecjalizowany.

Przykładowe zasady:

1. Dopasowana specjalizacja jawna szablonu jest bardziej preferowana,
2. jeśli szablon klasy jest typu **T** i dostępny jest inny szablon klasy, który przyjmuje **T***, twierdzi się, że wersja **T*** jest bardziej specjalizowana i preferowana od wersji generycznej **T** zawsze wtedy, gdy argument jest typem wskaźnika, nawet gdy oba są dostępnymi dopasowaniami.

Uporządkowanie kolejności dopasowania jest **częściowe**, ponieważ może istnieć wiele szablonów, które są uważane za jednakowo specjalistyczne.

© UKSW, WMP, SNS, Warszawa

229

229

Szablony – rozszerzenie

Rozważmy różne specjalizacje jawne szablonu klasy C oraz szablonny klas:

```
template<typename T, typename U>
class C {
public:
    void f() { cout << " Główny\n"; }
};
```

```
template<typename U>
class C<int, U> {
public:
    void f() { cout << "T == int\n"; }
};
```

```
template<typename T>
class C<T, double> {
public:
    void f() { cout << "U == double\n"; }
};
```

© UKSW, WMP, SNS, Warszawa

```
template<typename T, typename U>
class C<T*, U> {
public:
    void f() { cout << "użyto T*\n"; }
};
template<typename T, typename U>
class C<T, U*> {
public:
    void f() { cout << "użyto U*\n"; }
};
template<typename T, typename U>
class C<T*, U*> {
public:
    void f() { cout << "użyto T* i U*\n"; }
};
template<typename T>
class C<T, T> {
public:
    void f() { cout << "T == U\n"; }
};
```

230

230

Szablony – rozszerzenie

```
template<typename T, typename U>
class C {
public:
    void f() { cout << " Główny\n"; }
};
template<typename U>
class C<int, U> {
public:
    void f() { cout << "T == int\n"; }
};
template<typename T>
class C<T, double> {
public:
    void f() { cout << "U == double\n"; }
};
```

© UKSW, WMP, SNS, Warszawa

```
int main() {
    C<float, int>().f(); // 1: Główny
    C<int, float>().f(); // 2: T == int
    C<float, double>().f(); // 3: U == double
}
```

231

231

Szablony – rozszerzenie

```
template<typename T, typename U>
class C<T*, U> {
public:
    void f() { cout << "użyto T*\n"; }
};
```

```
template<typename T, typename U>
class C<T, U*> {
public:
    void f() { cout << "użyto U*\n"; }
};
```

```
template<typename T, typename U>
class C<T*, U*> {
public:
    void f() { cout << "użyto T* i U*\n"; }
};
```

© UKSW, WMP, SNS, Warszawa

```
template<typename T>
class C<T, T> {
public:
    void f() { cout << "T == U\n"; }
};
C<float*, float>().f(); // 4: użyto T* [T to float]
C<float, float*>().f(); // 5: użyto U* [U to float]
C<float*, int*>().f(); // 6: użyto T* i U* [float,int]
C<float, float>().f(); // 7: T == U
```

232

232

Szablony – rozszerzenie

1. template<typename T, typename U> class C
2. template<typename U> class C<int, U>
3. template<typename T> class C<T, double>
4. template<typename T, typename U> class C<T*, U>
5. template<typename T, typename U> class C<T, U*>
6. template<typename T, typename U> class C<T*, U*>
7. template<typename T> class C<T, T>

Poniższe są niejednoznaczne:

- 8: C<int, int>().f();
2 czy 7
- 9: C<double, double>().f();
3 czy 7
- 10: C<float*, float*>().f();
6 czy 7
- 11: C<int, int*>().f();
2 czy 5
- 12: C<int*, int*>().f();
6 czy 7

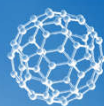
© UKSW, WMP, SNS, Warszawa

233

233

Szablony

Cechy charakterystyczne (trejty)



Szablony – rozszerzenie

Cechy charakterystyczne –
idiom* programowania za pomocą szablonów

Idiom – językown, związek wyrazowy (zwrot, wyrażenie) właściwy tylko danemu językowi, niedający się dosłownie przetłumaczyć;
np. pol. i. drzeć koty 'żyć w niesgodzie'

Właściwości szablonów pozwoliły na powstanie nowych technik programowania, zwanych *idiomami programowania*, których wykorzystanie nadaje programom specyficzny styl, sprawia że stają się łatwiejsze do zrozumienia i wykorzystania przez innych programistów, znających zastosowane idiomy

© UKSW, WMP, SNS, Warszawa

235

235

234

Szablony – rozszerzenie

Cechy charakterystyczne

Technika cech charakterystycznych (*traits*) jest idiomem i polega na grupowaniu deklaracji związanych z danym typem w obrębie jednej klasy – tworzenie *klasy cech charakterystycznych* lub po prostu klasy cech.

© UKSW, WMP, SNS, Warszawa

236

236

Szablony – rozszerzenie

Cechy charakterystyczne

pozwalają mieszać i dobrać pewne typy i wartości w zależności od bieżącego kontekstu pozwalającego używać ich w elastyczny sposób, dzięki czemu poprawia się czytelność i łatwość w utrzymaniu kodu.

Przykładem cech charakterystycznych jest szablon klasy `numeric_limits` z pliku nagłówkowego `<limits>`

W jednym z poprzednich slajdów korzystaliśmy z metody:

```
numeric_limits<double>::epsilon()
```

Było to odwołanie do składowej specjalizacji szablonu `numeric_limits` dla typu `double`. Takie specjalizacje dla wszystkich typów wbudowanych są – obok szablonu – zawarte w pliku `<limits>`

© UKSW, WMP, SNS, Warszawa

237

237

Szablony – rozszerzenie

Cechy charakterystyczne

Przykład #1:

```
#include <iostream>
#include <limits>
using namespace std;

int main()
{
    cout << "int is signed: " << numeric_limits<int>::is_signed;
    cout << "max values\n";
    cout << "int: " << numeric_limits<int>::max() << endl;
    cout << "double: " << numeric_limits<double>::max() << endl;
    cout << "float: " << numeric_limits<float>::max() << endl;
    cout << "unsigned int: " << numeric_limits<unsigned int>::max() << endl;

    return 0;
}
```

© UKSW, WMP, SNS, Warszawa

238

238

Szablony – rozszerzenie

Cechy charakterystyczne

Zaczynamy od utworzenia klasy bazowej, np.:

```
__numeric_limits_base
```

i zadeklarowania w niej pól reprezentujących cechy.

W Visual Studio 2008 wygląda to tak:

© UKSW, WMP, SNS, Warszawa

239

239

Szablony – rozszerzenie

```
struct __numeric_limits_base
{
    /** This will be true for all fundamental types (which have
        specialisations), and false for everything else. */
    static const bool is_specialized = false;

    /** The number of 80 radix digits that be represented without change: for
        integer types, the number of non-sign bits in the mantissa; for
        floating types, the number of 80 radix digits in the mantissa. */
    static const int digits = 0;

    /** The number of base 10 digits that can be represented without change.
        */
    static const int digits10 = 0;

    /** True if the type is signed. */
    static const bool is_signed = false;

    /** True if the type is integer.
        */
    static const bool is_integer = false;

    /** True if the type uses an exact representation. "All integer types are
        exact, but not all exact types are integer. For example, rational and
        fixed-exponent representations are exact but not integer."
        (8.3.1.2)/15
        */
    static const bool is_exact = false;

    /** For integer types, specifies the base of the representation. For
        floating types, specifies the base of the exponent representation. */
    static const int radix = 0;

    itd.
}
```

© UKSW, WMP, SNS, Warszawa

240

A w Visual Studio 2010 – tak:

240

Szablony – rozszerzenie

```
struct _CRTIMP2_PURE_Num_base
{
    /** base for all types, with common
        defaults
        */
    _STCONS(float_round_style, round_style,
            round_toward_zero);
    _STCONS(int, digits, 0);
    _STCONS(float_denorm_style, has_denorm, _STCONS(int, digits10, 0);
            denorm_absent);
    _STCONS(bool, has_denorm_loss, false);
    _STCONS(bool, has_infinity, false);
    _STCONS(bool, has_quiet_NaN, false);
    _STCONS(bool, has_signaling_NaN, false);
    _STCONS(bool, is_bounded, false);
    _STCONS(bool, is_exact, false);
    _STCONS(bool, is_iec559, false);
    _STCONS(bool, is_integer, false);
    _STCONS(bool, is_modulo, false);
    _STCONS(bool, is_signed, false);
    _STCONS(bool, is_specialized, false);
    _STCONS(bool, tinyness_before, false);
    _STCONS(bool, traps, false);

    #if _HAS_CFPFX
    _STCONS(int, max_digits10, 0);
    #endif /* _HAS_CFPFX */
    _STCONS(int, max_exponent, 0);
    _STCONS(int, max_exponent10, 0);
    _STCONS(int, min_exponent, 0);
    _STCONS(int, min_exponent10, 0);
    _STCONS(int, radix, 0);
};

Przy czym:
#define _STCONS(ty, name, val)
static const ty name = (ty)(val)
```

© UKSW, WMP, SNS, Warszawa

241

241

Szablony – rozszerzenie

Cechy charakterystyczne

Teraz, mając klasę bazową z listą pól reprezentujących cechy oraz ustalonymi dla nich wartościami domyślnymi, projektujemy szablon klasy, która będzie dziedziczyła po bazowej (wersja VS 2010):

```
template<class _Ty>
class numeric_limits: public _Num_base
{ // numeric limits for arbitrary type _Ty (say little or nothing)
  ...
}
```

W szablonie klasy należy wpisać ewentualne niezbędne metody dla tej klasy.

© UKSW, WMP, SNS, Warszawa

242

242

Szablony – rozszerzenie

Cechy charakterystyczne

A następnie możemy już deklarować specjalizacje dla różnych typów danych (wersja VS 2010):

```
// CLASS numeric_limits<char>
template<
class _CRTIMP2_PURE numeric_limits<char>: public _Num_int_base
{ // limits for type char
public:
  typedef char _Ty;
  // tutaj zaktualizowane wersje odziedziczonych metod
  ...
  // tutaj zaktualizowane wartości stałych
  _STCONS(bool, is_signed, CHAR_MIN != 0);
  _STCONS(int, digits, CHAR_BIT - (CHAR_MIN != 0 ? 1 : 0));
  ...
}
```

© UKSW, WMP, SNS, Warszawa

243

243

Szablony – rozszerzenie

Cechy charakterystyczne

Aby ustalić, czy **double** jest typem całkowitoliczbowym:

```
numeric_limits<double>::is_integer
```

Aby znaleźć najmniejszą liczbę typu **int** używa się metody

```
numeric_limits<int>::min()
```

Nie wszystkie składniki mają zastosowanie dla wszystkich typów wbudowanych – np. **epsilon()** odnosi się jedynie do typów wbudowanych, dla których:

```
numeric_limits<T>::is_integer == false.
```

Uwaga techniczna:

- wartości zwracane, które zawsze będą całkowite, deklarowane są jako pola statyczne w klasie, natomiast
- te które mogą nie być całkowite, implementowane są jako metody statyczne (tylko pola statyczne z wartościami całkowitoliczbowymi mogą być inicjalizowane (ukw@natrz definicji klasy).

© UKSW, WMP, SNS, Warszawa

244

244

Szablony – rozszerzenie

Cechy charakterystyczne – zastosowanie

Przyjmijmy teraz, że pewien nasz szablon klasy będziemy konkretyzować z różnymi typami danych, które mogą mieć różne wartości tych samych cech (np. najmniejsza wartość dla **int** jest inna niż dla **double**)

Ogólna zasada wprowadzania cech charakterystycznych do definicji naszej klasy:

- Definiujemy szablon klasy bazowej ze zbiorem atrybutów, takich jakie nasza klasa mogłaby wykorzystywać w swoim działaniu
- Definiujemy specjalizacje tego szablonu dla poszczególnych, różnych typów danych, jakie potencjalnie mogą być użyte
- Wykorzystujemy ten szablon jako domyślną wartość dla drugiego (kolejnego) parametru szablonu naszej klasy, gdzie pierwszym parametrem jest zawsze typ danych, dla którego nasza klasa będzie konkretyzowana.

© UKSW, WMP, SNS, Warszawa

245

245

Szablony – rozszerzenie

Cechy charakterystyczne

Przykład #2:

Nieco bardziej złożony przykład cech charakterystycznych - klasa **string**.

Jest to specjalizacja szablonu **basic_string**:

```
template< class _Elem,
          class _Traits = char_traits<_Elem>,
          class _Ax = allocator<_Elem> >
class basic_string: public _String_val<_Elem, _Ax>;
```

- Parametr **_Elem** reprezentuje typ znaków przechowywanych w obiekcie typu **basic_string<...>**
- Dla każdego z rozpatrywanych typów istnieją też specjalizacje z wykorzystaniem szablonów **char_traits** i **allocator**

© UKSW, WMP, SNS, Warszawa

246

246

Szablony – rozszerzenie

Cechy charakterystyczne

Deklarując obiekt typu **string** deklarujemy faktycznie obiekt typu **basic_string<char>**, co z uwagi na istnienie domyślnych argumentów daje ostatecznie typ:

```
basic_string<char, char_traits<char>, allocator<char>>
```

W ten sposób, przez oddzielenie cech charakterystycznych znaków od szablonu **basic_string** czynimy ten szablon bardziej uniwersalnym – tworzony jest rodzaj standardu specyfikacyjnego, z którego korzystamy w jednakowy sposób mimo istotnie różnych właściwości typów danych, z których korzystamy.

© UKSW, WMP, SNS, Warszawa

247

247

Szablony – rozszerzenie

Cechy charakterystyczne

Przykład #3:

Przyjmijmy, że mamy kilka klas reprezentujących smakołyki oraz kilka klas reprezentujących różnych bohaterów bajek. Mamy też klasę reprezentującą cechy smakołyków wybieranych przez bohaterów bajek. Prezentując bohaterów chcemy w uniwersalny sposób prezentować też ich smakołyki, ale odpowiednio do poszczególnych bohaterów.

W tym celu definiujemy szablon cech charakterystycznych smakołyków i specjalizujemy go dla poszczególnych bohaterów.

Prezentując bohaterów i ich smakołyki korzystamy z istniejących specjalizacji, które mają uniwersalny, jednorodny format (ponieważ powstały na podstawie tego samego szablonu).

© UKSW, WMP, SNS, Warszawa

248

248

Szablony – rozszerzenie

```
class Milk {
public:
    friend ostream& operator<<(ostream& os, const Milk&) {
        return os << "Mleko"; }
};
class CondensedMilk {
public:
    friend ostream& operator<<(ostream& os, const CondensedMilk&) {
        return os << "Skondensowane Mleko"; }
};
class Honey {
public:
    friend ostream& operator<<(ostream& os, const Honey&) {
        return os << "Miodek"; }
};
class Cookies {
public:
    friend ostream& operator<<(ostream& os, const Cookies&) {
        return os << "Ciasteczka"; }
};
```

© UKSW, WMP, SNS, Warszawa

249

249

Szablony – rozszerzenie

```
class Bear {
public:
    friend ostream& operator<<(ostream& os, const Bear&) {
        return os << "Puchatek";
    }
};
class Boy {
public:
    friend ostream& operator<<(ostream& os, const Boy&) {
        return os << "Krzys";
    }
};
```

© UKSW, WMP, SNS, Warszawa

250

250

Szablony – rozszerzenie

```
// główny szablon cech (pusty, może opisywać cechy wspólne)
template<class Guest>
class GuestTraits { };

// Cechy charakterystyczne - specjalizacje typu Guest:
// Specjalizacja dla misia
template<> class GuestTraits<Bear> {
public:
    typedef CondensedMilk beverage_type;
    typedef Honey snack_type;
};
// specjalizacja dla chłopca
template<> class GuestTraits<Boy> {
public:
    typedef Milk beverage_type;
    typedef Cookies snack_type;
};
```

© UKSW, WMP, SNS, Warszawa

251

251

Szablony – rozszerzenie

```
// Szablon Guest - korzysta z klasy cech
template<class Guest, class traits = GuestTraits<Guest> >
class BearCorner {
    Guest theGuest;
    typedef typename traits::beverage_type beverage_type;
    typedef typename traits::snack_type snack_type;
    beverage_type bev;
    snack_type snack;

public:
    BearCorner(const Guest& g) : theGuest(g) {}
    void entertain() {
        cout << „Wchodzi " << theGuest
            << " podajemy " << bev
            << " i " << snack << endl;
    }
};
```

© UKSW, WMP, SNS, Warszawa

252

252

Szablony – rozszerzenie

Zastosowanie

```
int main() {
    Boy cr;
    BearCorner<Boy> pc1(cr);
    pc1.entertain();
    Bear pb;
    BearCorner<Bear> pc2(pb);
    pc2.entertain();
} // co pojawi się w oknie konsoli?
```

© UKSW, WMP, SNS, Warszawa

253

253

Szablony – rozszerzenie

```
Wyjście: Wchodzi Krzys podajemy Mleko i Ciasteczka
         Wchodzi Puchatek podajemy Skondensowane Mleko i Miodek
```

W programie odpowiednie klasy gości otrzymują poczęstunek zgodnie ze swoimi gustami.

Powiązanie gości z przedmiotami wykonywane jest przez specjalizację podstawowego (pustego) szablonu klasy z cechami charakterystycznymi poszczególnych gości.

Domyślny argument drugiego parametru klasy `BearCorner`:

```
class traits = GuestTraits<Guest>
```

gwarantuje, że zostaną przypisane właściwe specjalizacje.

© UKSW, WMP, SNS, Warszawa

254

254

Szablony – rozszerzenie

Nowe, specjalizowane klasy cech dodane przez nas:

```
class MixedUpTraits {
public:
    typedef Milk beverage_type;
    typedef Honey snack_type;
};
```

Teraz już możemy korzystać nie z domyślnej klasy cech, ale z własnej:

```
int main() {
    Boy cr;
    BearCorner<Boy> pc1(cr);
    pc1.entertain();
    Bear pb;
    BearCorner<Bear> pc2(pb);
    pc2.entertain();
    BearCorner<Bear, MixedUpTraits> pc3(pb);
    pc3.entertain();
} // co pojawi się w oknie konsoli?
```

© UKSW, WMP, SNS, Warszawa

255

255

Szablony – rozszerzenie

Cechy charakterystyczne – przykład:

Domyślny argument drugiego parametru klasy `BearCorner` gwarantuje, że zostaną przypisane właściwe specjalizacje, ale ...

jednocześnie istnieje możliwość zmiany cech charakterystycznych poprzez jawne podanie w deklaracji wykorzystującej szablon nazwy klasy reprezentującej zdefiniowane przez nas inne cechy charakterystyczne.

© UKSW, WMP, SNS, Warszawa

256

256