Zaawansowane CPP/Wykład 12: Używanie funktorów

From Studia Informatyczne

< Zaawansowane CPP

Spis treści

Wstęp

W poprzednim wykładzie omawiałem pojęcie obiektu funkcyjnego, jako uogólnienia wskaźników do funkcji. I choć funktory można wywoływać bezpośrednio tak jak funkcje, to dużo częściej używa się ich jako parametrów przekazywanych do innych funkcji, w celu dostarczenia tej funkcji informacji koniecznych do wykonania swojego zadania. W tej kategorii zastosowań mieści się większość bibliotek, w tym STL, gdzie funktory służą jako argumenty do uogólnionych algorytmów. Przykłady takich zastosowań były już podawane w poprzednich wykładach, a na tym wykładzie omówię je trochę bardziej szczegółowo.

Inny popularny schemat użycia wskaźników funkcji (a więc i funktorów) to rozdzielenie miejsca i czasu definicji funkcji od miejsca i czasu jej wykonania. Wygląda to zwykle następująco: Klient definiuje funkcje i przekazuje ich wskaźniki do aplikacji, aplikacja wywołuje te funkcje w wybranym przez siebie czasie. Klient nie ma pojęcia kiedy zostaną wywołane przekazane przez niego funkcje, a aplikacja nie wie jakie one mają działanie. Taka technika funkcji zwrotnych (callbacks) jest podstawą implementacji wielu szkieletów graficznych interfejsów użytkownika i w E. Gamma, R. Helm, R. Johnson, J. Vlissides *"Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku"* wymieniona jest jako wzorzec polecenie. Aby używać funktorów definiowanych na poprzednim wykładzie, we wzorcu polecenie trzeba będzie je trochę dostosować. Zostanie to opisane w drugiej części tego wykładu.

Algorytmy uogólnione i programowanie funkcyjne

Algorytmy stanowią, jak już to opisałem w wykładzie 2 (http://osilek.mimuw.edu.pl/index.php?title=Zaawansowane_CPP/Wyk%C5%82ad_2:_Programowanie_uog%C3%B3lnione), jedną z części biblioteki STL, większośc z tych algorytmów posiada wersje przyjmujące funktor jako jeden z argumentów. Nie jest moim celem przedstawianie tutaj wszystkich algorytmów, jest ich zresztą blisko 100, choć gorąco zachęcam Państwa do zapoznania się z nimi. W tym celu polecam książke N.M. Josuttis *"C++ Biblioteka Standardowa. Podręcznik programisty"* i stronę http://www.sgi.com/tech/stl. Znakomita jest też pozycja S. Meyers *"STL w praktyce. 50 sposobów efektywnego wykorzystania"*, jak zresztą wszystkie książki tego autora.

W tym wykładzie chciałbym tylko zwrócić uwagę, że biblioteka algorytmów wprowadza do C++ elementy programowania funkcyjnego. Programowanie funkcyjne polega, z grubsza rzecz biorąc, na zastępowaniu pętli poleceniami, które potrafią wywołać daną funkcję na każdym elemencie danej kolekcji. Taki styl programowania jest często spotykany w językach interpretowanych, np. używa go pakiet `Mathematica`,

podobnie pakiet do obliczeń numerycznych w Perlu: Perl Data Language i wiele innych. Biblioteka STL oferuje tylko namiastkę tego stylu, ale właśnie ją chciałbym przedstawić w tej części wykładu.

Podstawą programowania funkcyjnego są funkcje, które wywołują inne funkcje na każdym obiekcie z kolekcji. STL oferuje kilka takich algorytmów, podstawowym z nich jest `for_each`.

```
template <class InputIterator, class UnaryFunction>
UnaryFunction
for_each(InputIterator first,InputIterator last,
         UnaryFunction op);
```

Działanie tego algorytmu polega na wywołaniu podanego funktora `op` na każdym elemencie zakresu `[first,last)`. Funktor może modyfikować wywoływany obiekt i może powodowac inne skutki uboczne. Jego kopia jest zwracana po wykonaniu wszystkich operacji. Wartość zwracana przez funktor `op` jest ignorowana. Algorytm zwraca kopię funktora `op`.

Podobnym algorytmem jest `transform`. W swojej pierwszej wersji

```
template <class InputIterator, class OutputIterator, class UnaryFunction>
OutputIterator transform(InputIterator first, InputIterator last,
                         OutputIterator result, UnaryFunction op);
```

działa podobnie jak `for_each`, z tą różnicą, że wyniki wywołania operacji `op` na wartościach zakresu `[firts,last)` są zapisywane poprzez iterator `result`. Ważną cechą tego algorytmu jest to, że może on operować na dwu wejściowych zakresach. Jego druga wersja

```
template <class InputIterator1, class InputIterator2,
          class OutputIterator,class BinaryFunction>
OutputIterator
transform(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, OutputIterator result,
          BinaryFunction op);
```

wywołuje operację binarną `op` na parach wartości wziętych po jednej z każdego zakresu wejściowego. Wynik zapisywany jest poprzez iterator wyjściowy `result`.

Warto też zwrócić uwagę na algorytmy numeryczne, które pomimo ich nazwy mogą spokojnie zostać użyte do innych ogólnych zastosowań. Są to:

```
template <class InputIterator, class T, class BinaryFunction>
T accumulate(InputIterator first, InputIterator last, T init,
             BinaryFunction op);
```

który oblicza uogólnioną sumę podanego zakresu

$$init \operatorname{op} a_1 \operatorname{op} a_2 \operatorname{op} \cdots \operatorname{op} a_n \equiv \operatorname{op}(\operatorname{op}(\operatorname{op}(init, a_1), a2)...a_n)$$

iloczyn skalarny `inner_product`:

```
template <class InputIterator1, class InputIterator2, class T,
          class BinaryFunction1, class BinaryFunction2>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init, BinaryFunction1 op1,
                BinaryFunction2 op2);
```

który oblicza uogólniony iloczyn skalarny

$$(a_1 \operatorname{op}_1 b_1) \operatorname{op}_2 (a_2 \operatorname{op}_1 b_2) \operatorname{op}_2 \cdots \operatorname{op}_2 (a_n \operatorname{op}_1 b_n) \cdots$$

oraz sumy i różnice częściowe:

```
template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator
partial_sum(InputIterator first,   InputIterator last,
            OutputIterator result, BinaryOperation binary_op);

template <class InputIterator, class OutputIterator, class BinaryFunction>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                                   OutputIterator result,
                                   BinaryFunction op);
```

zwracające poprzez iterator `result` odpowiednio:

$$a_1, a_1 \operatorname{op} a_2, a_1 \operatorname{op} a_2 \operatorname{op} a3, \ldots, a_1 \operatorname{op} \cdots \operatorname{op} a_n$$

i

$$a_1, a_2 \operatorname{op} a_1, a_3 \operatorname{op} a2, \ldots, a_n \operatorname{op} a_{n-1}$$

Zwłaszcza algorytm `adjacent_difference` jest ciekawy, umożliwia bowiem zastosowanie dwuargumentowej operacji do kolejnych par elementów:

```
int print(int i,int j) {
  cout<<"("<<i<<":"<<j<<")";
  return 0;
}
main() {
  list<int> li;
  list<int> res;
  generate_n(back_insert_iterator<list<int> >(li),10,SequenceGen<int>(1,2));

  adjacent_difference(li.begin(),li.end(),back_insert_iterator<list<int> >(res),print);
```

(Źródło: sums.cpp)

Ten przykładzik ilustruje niedogodność posługiwania się algorytmami przyjmującymi zakres wyjściowy, takimi jak `adjacent_difference` czy `transform`, do prostego działania funkcją, która nie zwraca żadnych wartości. Do tego przeznaczony jest algorytm `for_each`. Niestety, ten algorytm nie pozwala bezpośrednio operować na parach kolejnych elementów, tak jak `adjacent_difference`. To by jeszcze można zasymulować używając odpowiedniego funktora, ale `for_each` nie potrafi operować na parach elementów pochodzących z dwu zakresów, tak jak potrafi to `transpose`.

Jest kilka możliwości rozwiązania tego problemu. Możemy np. napisać własne algorytmy (zob. zadania). Możemy też dalej korzystać np. z `adjacent_difference` ale napisać narzędzia pomagające adoptować takie algorytmy do naszych celów. Jak by to mogło wyglądać?

Patrzac na nasz przykład i deklarację `adjacent_difference` widzimy, że są dwa problemy: po pierwsze wartość zwracana z funkcji `op` musi być tego samego typu jak typ elementów w zakresie wejściowym, po drugie musimy jakoś "zjeść" te zwracane wartości. Potrzebny więc będzie funktor opakowujący dowolną funkcję i zwracający wartość zadanego typu, oraz iterator pełniący rolę `/dev/null`, czyli "czarnej dziury" połykającej wszystko co się do niej zapisze. Mając takie obiekty możemy powyższy kod zapisać następująco:

```
void print(int i,int j) {
  cout<<"("<<i<<":"<<j<<")";
}
main() {
  list<int> li;
```

```
  generate_n(back_insert_iterator<list<int> >(li),10,SequenceGen<int>(1,2));

  adjacent_difference(li.begin(),li.end(),dev_null<char>(),dummy<char>(print));
```

Obiekt klasy `dev_null<T>` to iterator typu `output_iterator`, którego `value_type` jest równy `T`. Iterator ten ignoruje wszelkie operacje na nim wykonywane, w tym przypisanie do niego. Funckja `dummy<T>(F f, T val = T())` zwraca funktor, który wywołuje funkcję `f`, a następnie zwraca wartośc `val` typu `T`. Implementacje tych szablonów pozostawiam jako ćwiczenie (zob. zadania).

Pomysły można mnożyć. Jako ostatni zaprezentuję iterator, który nie wstawia nic do żądanego kontenera, ale wywołuje na przypisywanym do niego obiekcie jakąś zadaną funkcję:

```
void p(double x) {
  std::cout<<"printing  "<<x<<"";
};

double  frac(int i) {
  return i/10.0;
}
  list<int> li;
  generate_n(back_insert_iterator<list<int> >(li),10,SequenceGen<int>(1,2));

std::transform(li.begin(),li.end(),function_iterator<double>(p),frac);
```

Taki iterator łatwo zaimplementować przy pomocy klas proxy (zob. zadania).

Wzorzec Polecenie

Wzorzec polecenie najlepiej omówić na przykładzie. Jak już wspomniałem, typowe zastosowanie to graficzny interfejs użytkownika (GUI). Taki interfejs musi reagować na różne zdarzenia: kliknięcie myszą, naciśnięcie przycisku, wybranie polecenia z menu. Ewidentnie twórca biblioteki GUI nie może wiedziec jakie działanie ma pociągnąć dane zdarzenie. Nawet jednak gdybyśmy sami pisali cały kod, to "zaszycie" polecenia na stałe w kodzie danego elementu interfejsu jest bardzo złą praktyką programistyczną. Dlatego używa się w tym celu funkcji zwrotnych. I tak w jakiejś hipotetycznej klasie

```
class Window {
```

znajdziemy na pewno funkcję w rodzaju

```
on_mouse_click(void (*)(int,int))
```

służącą do przekazania wskaźnika do funkcji, która zostanie wywołana po naciśnięciu myszką na oknie. Podobnie dla innych komponentów:

```
class Button {
void (*_f)() ; /*wskaźnik do funkcji */
void when_pressed(void (*f)()) {_f=f;};
...
}
```

Ponieważ typ funkcji określony jest poprzez typ jej parametrów i typ wartości zwracanej to możemy w ten sposób ustawić dowolną fukcję o odpowiedniej sygnaturze. Niestety, jeżeli chcemy wykorzystać dobrodziejstwa jakie daje nam zastosowanie obiektów funkcyjnych zamiast funkcji, musimy je dziedziczyć ze wspólnej klasy bazowej, ponieważ jedną z podstawowych cech funktorów jest to, że posiadają swój typ. Żeby więc móc przekazywać różne funktory za pomocą tej samej sygnatury, musimy je rzutować w górę na wspólny

typ. To jest cała esencja wzorca polecenie (zob. E. Gamma, R. Helm, R. Johnson, J. Vlissides *"Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku"*).

Oznacza to jednak, że nie możemy skorzystać z całej technologii wyprowadzonej na poprzednim wykładzie. Szablony też nam nie pomogą, ponieważ wenątrz elementów GUI musimy jakoś przechować funktor zwrotny. Aby przechowywać różne funktory musielibyśmy parametryzować nasz element typem funktora, co doprowadziłoby do tego, że typ elementu interfejsu zależałby od konkretnego polecenia, które wywołuje! To uniemożliwiłoby w praktyce jakiekolwiek funkcjonowanie szkieletu.

Żeby więc wykorzystać już istniejące funkcje i funktory musimy je opakować w typ, który będzie zależał tylko od typów wartości zwracanej i argumentów, podobnie jak w przypadku zwykłych funkcji. Poniżej przedstawię zarys konstrukcji szablonu, który umożliwia robienie tego automatycznie. Będę się opierał na implementacji zamieszczonej w A. Alexandrescu: *"Nowoczesne projektowanie w C++"*, ale ograniczę się do funktorów zero-, jedno- i dwuargumentowych.

Uniwersalny funktor

Uniwersalny funktor (nazwiemy go `Function`) ma z założenia odpowiadać typowi funkcyjnemu, czyli zależeć jedynie od typu wartości zwracanej i typów argumentów. Musi więc być zdefiniowany jako szablon, paremetryzowany właśnie tymi typami. Tu napotykamy pierwszy problem: C++ nie dopuszcza zmiennej liczby parametrów szablonu. Najprostszym nasuwającym się rozwiązaniem jest stworzenie trzech różnych szablonów `Function0`, `Function1` i `Function2`, każdy z odpowiednią liczbą parametrów. Takie rozwiązanie jest jednak niegodne prawdziwego uogólnionego programisty :). Poszukajmy więc typów, które zawierają w sobie inne typy. Możliwości mamy całkiem sporo, moglibyśmy np. wykorzystać klasy ze zdefiniowanymi odpowiednimi typami stowarzyszonymi, można wykorzystać typy funkcyjne lub typy wskaźników do funkcji, no i listy typów wprowadzonych w wykładzie 6.3 (http://osilek.mimuw.edu.pl/index.php?title=Zaawansowane_CPP/Wyk%C5%82ad_6:_Funkcje_typ%C3%B3w_i_inne_sztuczki#6.3_Listy_typ.C3.B3w) właśnie do takich celów. Orginalne rozwiązanie w A. Alexandrescu *"Nowoczesne projektowanie w C++"* wykorzystuje listy typów (autor tej pozycji jest ich twórcą), ja użyję typów wskaźnikow do funkcji. To, że bedę używał wskaźników, a nie samych typów funkcji, podyktowane jest względami czysto technicznymi: moja klasa cech `functor_traits` rozpoznaje typy wskaźników do funkcji, a same typy funkcyjne już nie (choć jest to tylko kwestia dodania odpowiednich specjalizacji). Deklaracja szablonu `Function` wyglądała będzie więc następująco:

```
template<typename FT> class Function:
public  functor_traits<FT>::f_type {

  typedef typename functor_traits<FT>::result_type res_type;
  typedef typename functor_traits<FT>::arg1_type   arg1_type;
  typedef typename functor_traits<FT>::arg2_type   arg2_type;
 public:
...
};
```

(Źródło: universal.h)

Dziedziczenie z `functor_traits<FT>::f_type` zapewnia nam, że `Function` będzie posiadał odpowiednie typy stowarzyszone zgodnie z konwencją STL.

Jako klasa opakowująca `Function` musi zawierać opakowywany funktor. Nie może jednak tego robić bezpośrednio, bo nie znamy typu tego funktora (a raczej nie chcemy go znać). `Function` będzie więc zawierał wskaźnik do abstrakcyjnej klasy `AbstractFunctionHolder<FT>` parametryzowanej tym samym typem.

```
std::auto_ptr<AbstractFunctionHolder<FT> > _ptr_fun;
```

Z tej klasy będą dziedziczyć klasy opakowujące konkretne typy funktorów:

```
template<typename FT,typename F>
class FunctionHolder: public AbstractFunctionHolder<FT> {

  typedef typename functor_traits<FT>::result_type res_type;
  typedef typename functor_traits<FT>::arg1_type arg1_type;
  typedef typename functor_traits<FT>::arg2_type arg2_type;
...

private:
   F _fun;
};
```

(Źródło: universal.h)

Konstruktor klasy `FunctionHolder` będzie inicjalizował pole `_fun`:

```
FunctionHolder(F fun):_fun(fun) {};
```

Korzystać z niego będzie konstruktor klasy `Function`, który musi być szablonem aby pozwolić na inicjalizowanie `Function` dowolnym typem funktora:

```
template<typename F>  Function(F fun):
  _fun(new FunctionHolder<FT,F>(fun)){};
```

(Źródło: universal.h)

Ten konstruktor zamienia statyczną informację o typie `F` na informację dynamiczną zawartą we wskaźniku `_ptr_fun`. Tę informację odzyskujemy za pomocą polimorfizmu. W celu uzyskania polimorficznego zachowania przy kopiowaniu i przypisywaniu obiektów `Function` skorzystamy z klonowania:

```
Function(const Function& f):_fun(f._fun->Clone()) {};
Function &operator=(const Function &f) {
  _fun.reset(f._fun->Clone());
}
```

(Źródło: universal.h)

Wymaga to dodania do klasy `AbstractFunctionHolder` wirtualnych funkcji:

```
virtual  AbstractFunctionHolder* Clone()=0;
virtual  AbstractFunctionHolder() {};
```

(Źródło: universal.h)

Klasa `FunctionHolder` implementuje funkcję `Clone()` następująco:

```
FunctionHolder*Clone() {return new FunctionHolder(*this);}
```

(Źródło: universal.h)

Proszę zwrócić uwage na typ argumentu konstruktorów. Zgodnie z tym co napisałem w wykładzie 11.2 (http://osilek.mimuw.edu.pl/index.php?title=Zaawansowane_CPP/Wyk%C5%82ad_11:_Funktory#Funkcje.2C_wska.C5.BAniki_i_referencje__do_funkcji), użyłem wywołania przez wartość `(F fun)`, inaczej nie możnaby inicjalizować pola `F _fun` w przypadku gdyby `F` był typem funkcyjnym. Należy nadmienić, że implementacja

opisana w A. Alexandrescu: *"Nowoczesne projektowanie w C++"* zawiera właśnie taki błąd, który nie występuje w kodzie biblioteki `loki` dostępnym w Internecie.

Wywoływanie: operator()

W ten sposób mamy już wszystko poza najważniejszym: operatorem `operator()(...)`, który czyni funktor funktorem.

Zacznijmy od operatora nawiasów w klasie `Function`. Problem polega na tym, że nie ma możliwości zdefiniowania operatora odpowiadającego przekazanemu typowi funkcyjnemu (można by ewentulanie go zadeklarować), bo wymagałoby to definicji dopuszczającej zmienną liczbę argumengtów (tak, tak, wiem w C jest taka możliwość, ale lepiej z niej nie korzystać, zresztą nie ma potrzeby). Możliwe rozwiązanie to specjalizacja szablonu dla funktorów bez argumentów, z jednym lub z dwoma argumentami i w każdej specjalizacji zdefiniowanie operatora `operator()(...)` z odpowiednią liczbą argumentów. Okazuje się, że nie ma takiej potrzeby, możemy zdefiniować wszystkie wersje operatora wywołania funkcji w tej samej klasie i polegać na mechaniźmie opóźnionej konkretyzacji: dopóki nie wywołamy operatora ze złą ilością argumentów, to nie będzie on konkretyzowany. Dodajemy więc do klasy `Function` następujące linijki:

```
res_type operator()() {return (*_ptr_fun)();};
res_type operator()(arg1_type x) {return (*_ptr_fun)(x);};
res_type operator()(arg1_type x,arg2_type y) {
  return (*_ptr_fun)(x,y);
};
```

(Źródło: universal.h)

Wskaźnik `_ptr_fun` jest typu `AbstractFunctionHolder*`, musimy więc zaimplementować w tej klasie operator nawiasów. Klasa `AbstractFunctionHolder*` nie posiada wystarczającej w tym celu informacji, więc deklarujemy te funkcje jako czyste funkcje wirtualne, które zostaną zdefiniowane dopiero w klasie pochodnej `FunctionHolder`. Niestety deklaracja trzech różnych wariantów operatora

```
 virtual res_type operator()() = 0;
 virtual res_type operator()(arg1_type x) = 0;
 virtual res_type operator()(arg1_type x,arg2_type y)=0;
```

i odpowiadające im definicje w klasie `FunctionHolder`

```
 res_type operator()() {return _fun();};
 res_type operator()(arg1_type x) {return _fun(x);};
 res_type operator()(arg1_type x,arg2_type y) {return _fun(x,y);};
```

spowodują błędy w kompilacji już przy konstrukcji klasy. Dzieje się tak dlatego, że funkcje wirtualne mogą niepodlegać konkretyzacji opóźnionej i w implementacji kompilatora `g++` jej nie podlegają. Tzn. kiedy kompilator dokonuje konkretyzacji klasy `FunctionHolder<FT,F>`, wymaganej przez konstruktor inicjalizujący klasy `Function`, konkretyzuje wszystkie funkcje wirtualne, a tylko jedna wersja operatora wywołania jest prawidłowa. W tym przypadku nie unikniemy więc konieczności specjalizowania szablonów dla każdej ilości agumentów. Na szczęście wystarczy wyspecjalizować tylko klasę `AbstractFunctionHolder`:

```
template<typename FT> class AbstractFunctionHolder ;

template<typename R,typename A1,typename A2>
class AbstractFunctionHolder<R (*)(A1,A2)> {
 public:
  virtual R operator()(A1 x, A2 y) = 0;
  virtual  AbstractFunctionHolder* Clone()=0;
```

```
  virtual  AbstractFunctionHolder() {};
} ;
```

(Źródło: universal.h)

i tak samo dla funkcji z jednym i bez argumentów.

Szablon `FunctionHolder` może dalej definiować wszystkie warianty operatora wywołania. Prześledźmy teraz konkretyzację tych szablonów na przykładzie:

```
double f(double x) {return x;}

Function<double(*)(double)> Func1;
```

Konstrutor klasy `Function<double (*)(double)>` wywoła konstruktor klasy `FunctionHandler<double (*)(double),double (*)(double)>`, co pociągnie za sobą konieczną konkretyzację tego szablonu. Ten szablon dziedziczy z

```
AbstractFunctionHandler<double (*)(double)>
```

która ma zdefiniowany tylko jeden operator

```
virtual double operator(double x) = 0
```

Ale to oznacza, że z trzech wariantów operatora wywołania zdefiniowanych w `FunctionHandler<double (*)(double),double (*)(double)>` tylko jeden jest wirtualny: ten dobry! I właśnie ten zostanie skonkretyzowany, pozostałe dwa są zwykłymi funkcjami składowymi i nie zostaną utworzone jesli ich nie użyjemy. Jeśli jednak spróbujemy ich użyć dostaniemy błąd kompilacji, co jest w tej sytuacji zachowaniem pożądanym.

Używając szablonu `Function` możemy teraz napisać:

```
#include"universal.h"
  Cov f(1.0,2.0,2.0);

  Function<double (*)(double,double)>  fc;
  fc=compose_f_gxy(
                           __gnu_cxx::compose1(std::ptr_fun(exp),
                                        std::negate<double>()),
                           f);

std::cout<<fc(1.0,1.0)<<"";

Function<double (*)(double)> my_exp(exp);

std::cout<<my_exp(1.0)<<"";
```

(Źródło: test_universal.cpp)

boost::functions

Podobną funkcjonalność dostarcza biblioteka `functions` z repozytorium `boost`. Zresztą stamtąd wziąłem pomysł parametryzowania funktorów typem funkcyjnym. Kod używający tej biblioteki będzie wyglądał bardzo podobnie:

```
#include<boost/function.hpp>
using namespace boost;
```

```
  Cov f(1.0,2.0,2.0);

  boost::function<double (double,double)>  fc;
  fc=compose_f_gxy(
                           __gnu_cxx::compose1(std::ptr_fun(exp),
                                        std::negate<double>()),
                           f);

std::cout<<fc(1.0,1.0)<<"";
```

(Źródło: boost_function.cpp)

Źródło: "http://wazniak.mimuw.edu.pl/index.php?title=Zaawansowane_CPP/Wyk%C5%82ad_12:_U%C5%BCywanie_funktor%C3%B3w"

- Tę stronę ostatnio zmodyfikowano o 13:49, 11 gru 2007;