Zaawansowane CPP/Wykład 11: Funktory

From Studia Informatyczne

< Zaawansowane CPP

Spis treści

Wstęp

Na poprzednim wykładzie prezentowane były inteligentne wskaźniki, czyli obiekty, które rozszerzają pojęcie zwykłego wskaźnika. Na tym wykładzie będę omawiał obiekty, które stanowią rozszerzenie pojęcia wskaźnika do funkcji. Motywacja wprowadzenia takich obiektów funkcyjnych jest jednak inna. Funkcje obiektami **nie** są i wskaźniki do nich nie mają kłoptów z prawami własności, współdzieleniem referencji, itp. Same wskaźniki do funkcji są obiektami ale typów wbudowanych (lub raczej typów złożonych). Możemy je kopiować, przepisywać, przekazywać do funkcji, ale nie mamy nad tym kontroli.

Obiekty funkcyjne posiadaja składnię wywołania funkcji, ale są też pełnoprawnymi obiektami, mogą więc posiadać stan, konstruktory, destruktory i inne metody, jak również typy stowarzyszone, no i oczywiście posiadają też swój własny typ. Te dodatkowe informacje pozwalają na implementowanie wielu ciekawych rozwiązań niedostępnych dla zwykłych funkcji i wskaźników do nich.

Funkcje, wskaźniki i referencje do funkcji

Typy funkcyjne

Zanim zajmiemy się bardziej skomplikowanymi obiektami jakimi są funktory, chciałbym najpierw wyjaśnić kilka faktów dotyczących funkcji i wskaźników do nich. Jak już wspomniałem w poprzednim podrozdziale funkcje nie są obiektami. Nie można ich kopiować ani przypisywać do siebie:

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

void  g(double) = f; niedozwolone
void h(double);
h=f; niedozwolone
```

Funkcje posiadają jednak typ. Typ funkcji (nazywany typem funkcyjnym) jest określony przez typ jej wartości zwracanej i typy jej argumentów. Typy funkcyjne możemy używać np. w poleceniu `typedef`. Wyrażenie:

```
typedef void f_type(double)
```

definiuje `f_type` jako typ funkcji o jednym argumencie typu `double` i nie zwracającej żadnej wartości. Taki typ ma jednak ograniczone zastosowanie, możemy go używać do deklarowania, ale nie definiowania innych funkcji:

```
f_type g;
```

Typ funkcyjny może też być użyty jako parametr szablonu:

```
template<typename F> Function {
F _fun;
};
Function<void (double)>
```

Niewiele jednak będziemy mieli pożytku z pola `_fun`, bo jak już widzieliśmy, nie będziemy w stanie nic do niego przypisać ani go zainicjalizować.

Możemy również używać typów funkcyjnych w deklaracjach argumentów funkcji. Wyrażenie:

```
double sum(double (double),...)
```

oznacza że funkcja `sum` oczekuje jako pierwszego argumentu funkcji zwracającej `double` o jednym argumencie typu `double`. Ten zapis jest jednak mylący! W rzeczywistości nie można przekazać funkcji jako argumentu wywołania i dlatego w deklaracjach argumentów typ funkcyjny jest automatycznie zamieniany na typ wskaźnika do funkcji i powyższa deklaracja jest równoważna deklaracji:

```
double sum(double (*)(double),...)
```

Wskaźniki do funkcji

Wskaźniki do funkcji są normalnymi obiektami i mogą być kopiowane i przypisywane:

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

void  (*g)(double) = &f;
void (*h)(double);
h=&f;
(*h)(0.0);
(*g)(1.0);
```

C++ posiada wygodną własność, która jednak zwiększa konfuzję pomiędzy funkcjami i wskaźnikami do nich. Otóż operatory `*` i `&` są aplikowane automatycznie do funkcji i powyższy kod można zapisać jako:

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

void  (*g)(double) = f;
void (*h)(double);
h=&f;
```

```
h(0.0);
g(1.0);
```

Jest to dość wygodne, ale powoduje, że część ludzi słabo rozróżnia funkcje od wskaźników do nich (Państwo oczywiście już do nich nie należą :).

Referencje do funkcji

Żeby już skończyć ten temat i zupełnie zamieszać Państwu w głowach napiszę, że można też definiować referencje do funkcji:

```
void f(double x) {};
typedef void f_type(double)

f_type  &g  = f;
f_type  &h  = g;
const f_type  &ch = g; równoważne z wyrażeniem f_type &ch  = g;
h=g; niedozwolone  h jest refencją do stałej
```

Należy dodać, że typ `const f_type &` nie jest obsługiwany przez kompilator `g++-3.3` ale przez `g++-3.4` już tak.

Dedukcja typów funcyjnych

Ponieważ funktory i funkcje często przekazywane są jako argumenty wywołania szablonów, których typ podlega dedukcji, warto wiedzieć jak ten mechanizn rozpoznaje typ przekazywanej funkcji. Rozważmy najpierw następującą definicję:

```
template<typename  F> test(F f) {
  F _fun(f);
}
```

Jeśli teraz wywołamy

```
 void (*g)(double) = f ;
 void (&h)(double) = f;

test(f);          F = void (*)(double)
test(g);          F = void (*)(double)
test(h);          F = void (*)(double)
```

to w każdym przypadku typ `F` zostanie wydedukowany jako wskaźnik do funkcji `void (*)(double)`.

Jeśli przypomnimy sobie, że argumenty do funkcji można dla oszczędności przekazywać jako referencje do stałych, to możemy napisać:

```
template<typename  F> test(const F &f) {
  F _fun(f);
}
```

czeka nas jednak niepospodzianka, kod

```
 void (*g)(double) = f ;
 void (&h)(double) = f;

test(f);          F = void ()(double), nie skompiluje  się
```

```
test(g);          F = void (*)(double)
test(h);          F = void ()(double) nie skompiluje się
```

zachowuje się już inaczej. W przypadku przekazania funkcji lub referencji do funkcji, wededukowanym typem będzie typ funkcyjny. Ponieważ nie można inicjalizować zmiennych typu funkcyjnego, wyrażenie `F _fun(f)` się nie skompiluje. Nie będzie kłopotów jeśli przekażemy wskaźnik do funkcji, ale tym razem musimy to zrobić jawnie, nie nastąpi automatyczna konwersja nazwy funkcji na wskaźnik do niej.

Można by przedefiniować funkcję `test` następująco:

```
template<typename  F> test(const F &f) {
  F *_fun(f);
}
```

Teraz wywołania

```
test(f);          F = void ()(double)
test(h);          F = void ()(double)
```

skompilują się, ale nie skompiluje się linijka

```
test(g);          F = void (*)(double)
```

bo pole `_fun` stanie się typu `void (**)(double)`. Kompilator `g+-3.3` nawet tego kodu nie skompiluje, bo nie zezwala na referencje do stałych typów funkcyjnych.

Widać więc, że przy przekazywaniu funkcji najlepiej używać wywołania przez wartość, która i tak jest automatycznie konwertowana na przekazywanie wskaźnika do funkcji.

Obiekty funkcyjne

Definiowanie własnych obiektów funkcyjnych jest możliwe dzięki możliwości przeładowania operatora wywołania (będziemy go też nazywać operatorem nawiasów): `operator()(...)`. Np.

```
struct Sin {
double operator()(double x) {return sin(x);}
}
```

Z obiektów typu `Sin` możemy teraz korzystać jak z funkcji:

```
Sin c_sin;
c_sin(0);
```

Nie jest to być może porywający przykład, ale głównym powodem używania obiektów funkcyjnych jest to, że mogą posiadać stan. Skorzystamy z tego, aby umożliwić wybór typu argumentu funkcji `sin`. Zwyczajowo kalkulatory (ktoś wie co to jest?) zezwalają na podawanie argumentów funkcji trygonometrycznych w radianach, stopniach lub gradusach. Możemy to zaimplemetować następująco:

```
class Sin {
public:
  enum arg {radian,degree,grad};
  Sin(arg s = radian):_state(s) {};

  void set_radians() { _state=radian;}
```

```
  void set_degrees() {_state=degree;}
  void set_grads()   {_state=grad;}

  double operator()(double x) {return sin(conv(x));}
private:
  arg _state;
  double conv(double x) {
    switch (_state) {
    case radian: return x;
    case degree : return x*(M_PI/180.0);
    case grad   : return x*(M_PI/200.0);
    }
  }
};
```

(Źródło: sin.cpp)

Nie jest to najwydajniejsza implementacja, bo za każdym wywołaniem funkcji `sin` wykonywana jest insrukcja `switch`. Przerobimy ją więc na:

```
class BetterSin {
public:
  enum arg {radian,degree,grad};

  void set_radians() { _conv=&BetterSin::to_radians;}
  void set_degrees() { _conv=&BetterSin::to_degrees;}
  void set_grads()   { _conv=&BetterSin::to_grads;}

  double operator()(double x) {return sin( (this->*_conv)(x) );}

  BetterSin(arg s = radian) {
    switch (s) {
    case radian: set_radians();break;
    case degree : set_degrees();break;
    case grad   : set_grads();break;
    }
  }
private:
  double (BetterSin::* _conv)(double);
  double to_radians(double x) {return x;};
  double to_degrees(double x) {return x*(M_PI/180.0);};
  double to_grads(double x) {return x*(M_PI/200.0);};
};
```

(Źródło: sin.cpp)

Wskaźniki do metod (funkcji składowych)

Ten przykład, który powinien działać szybciej, ilustruje ponadto użycie wskaźników do funkcji składowych (metod). Różnice pomiędzy wskaźnikami do funkcji i wskaźnikami do metod są opisane w D. Vandervoorde, N. Josuttis: *"C++ Szablony, Vademecum profesjonalisty"* oraz S. B. Lippman *"Model obiektu w C++"*, tutaj zwrócę tylko uwagę na różnice w ich używaniu. Jak już pisałem wskaźniki do funkcji można używać na skróty, bez jawnego wywoływania operatorów `&` i `*`. W przypadku wskaźników do metod, musimy pobierać adres jawnie i dereferencjonować go przed wywołaniem. Ponadto każda metoda ma dodatkowy niejawny argument, którym jest wskaźnik na obiekt, przez który została ona wywołana, dlatego wywołując metodę poprzez wskaźnik też musimy ten argument podać:

```
struct X {
  void f(){std::cout<<"f1"<<"";};
};
main(){
  typedef void (X::*f_ptr)();
    //f_ptr pf1=X::f; błąd kompilacji
  f_ptr pf1=&X::f;
    X x;
```

```
  X *px = new X;
    (x.*pf1)();
    (px->*pf1)();
}
```

(Źródło: m_ptr.cpp)

Adaptery funktorów

Obiekty mogą jednak posiadać więcej informacji niż tylko swój stan, mogą zawierać również informacje o typie. Przede wszystkim funktory same posiadają typ, konsekwencje tego faktu omówię poźniej, teraz zajmę się typami stowarzyszonymi. Nasuwająca się od razu możliwość, to stowarzyszenie z funktorem typu wartości zwracanej oraz typów jego argumentów. Można stowarzyszyć również informację o liczbie argumentów. W przypadku szablonu `Sin` i `BetterSin` moglibyśmy dodać

```
typedef double result_type;
typedef double argument_type;
enum {n_arguments = 1};
```

Taki schemat nazewnictwa nie najlepiej się rozszerza na większą ilość argumentów, ale właśnie takie definicje typów są wymagane dla jednoargumentowych obiektów funkcyjnych w STL. W celu ułatwienia tworzenia własnych funktorów spełniających te wymagania, dostarczony jest szablon klasy, z której można dziedziczyć:

```
template <class Arg, class Result>
struct unary_function {
typedef Arg argument_type;
typedef Result result_type;
};
```

czyli moglibyśmy również napisać:

```
class BetterSin: public unary_function<double,double> {...};
```

STL nie wymaga definiowania liczby argumentów.

Podobnie zdefiniowany jest szablon dla funkcji dwuargumentowych:

```
template <class Arg1, class Arg2, class Result>
struct binary_function {
typedef Arg1 first_argument_type;
typedef Arg2 second_argument_type;
typedef Result result_type;
};
```

Opis bardziej uniwersalnego schematu obiektów funkcyjnych, uwzględniający funktory z dowolną ilością argumentów znajduje się w D. Vandervoorde, N. Josuttis *"C++ Szablony, Vademecum profesjonalisty"*. Dodajmy, że do określenia typów argumentów można użyć listy typów przedstawione 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), razem z szablonem indeksowania (zob. A. Alexandrescu *"Nowoczesne projektowanie w C++"*).

My jednak pozostaniemy na razie przy schemacie z STL. Trzeba jasno powiedzieć, że obiekty funkcyjne używane przez algorytmy STL nie muszą posiadać wymienionych typów stowarzyszonych, w szczególności mogą to być zwykłe funkcje. Ale jeśli takie typy posiadają, to można używać ich w adapterach funkcji.

Adaptery funkcji to funktory, które w jakiś sposób modyfikują działanie innych funktorów. STL definiuje dość skromny wybór adapterów, ale na szczęście istnieje wiele innych niezależnych źródeł, w szczególności SGI i `boost`.

Jak działają adaptery wyjaśnię na przykładzie adaptera `binder1st` z STL. `binder1st` reprezentuje funktor jednoargumentowy, powstały przez zastąpienie pierwszego argumentu podanego funktora dwuargumentowego podaną wartością. Rozważmy następujący przykład:

```
class Cov:public std::binary_function<double,double,double> {
  const double _ax,_ay,_axy;
public:
  Cov(double ax,double ay,double axy):_ax(ax),_ay(ay),_axy(axy) {};
  double operator()(double x,double y) {return _ax*x*x+_ay*y*y+_axy*x*y;}
};

main() {
  Cov f(1.0,2.0,2.0);

  std::cout<<f(1.0,1.0)<<"";
  std::cout<<f(1.0,2.0)<<"";

  std::cout<<::bind1st(f,1.0)(1.0)<<"";
  std::cout<<::bind1st(f,1.0)(2.0)<<"";
}
```

(Źródło: bind.cpp)

Działanie adaptera nietrudno zrozumieć, jeśli się zrozumiało działanie szablonów wyrażeń (zob. wykład 9 (http://osilek.mimuw.edu.pl/index.php?title=Zaawansowane_CPP/Wyk%C5%82ad_9:_Szablony_wyra%C5%BCe%C5%84)). Szablon `binder1st` można zdefiniować następująco:

```
template<typename F> class binder1st {
  typedef  typename F::first_argument_type  bind_type;
  typedef  typename F::second_argument_type first_argument_type;
  typedef  typename F::result_type result_type;

  const bind_type _val;
  F _op;
  public:
  binder1st(F op,bind_type val):_op(op), _val(val) {};

  result_type operator()(first_argument_type x) {
    return op(_val,x);
  }
};
```

(Źródło: bind.cpp)

Podobnie jak w przypadku szablonów wyrażeń, będziemy jeszcze potrzebowali funkcji, która wytworzy nam taki obiekt. To zadanie spełni nam funkcja `bind1st`:

```
template<typename F>
binder1st<F>
bind1st(F op,typename F::first_argument_type val) {
return binder1st<F>(op,val);
}
```

(Źródło: bind.cpp)

Na podstawie tego przykładu mam nadzieję, że już łatwo Państwu będzie wywnioskować implementację pozostałych adapterów STL: `binder2nd`, `unary_negate` i `binary_negate`.

STL dostarcza ponadto dodatkowe adaptery, które opakowują zwykłe wskaźniki do funkcji i wskaźniki do metod tak, aby można było ich użyć razem z adapterami obiektów.

Składanie funktorów

Żaden z adapterów zaimplementowanych w STL nie umożliwia składania funktorów, czyli tworzenia funktora poprzez połączenie dwu lub wiecej innych funktorów. Oczywiście istnieje wiele możliwych sposobów składania funkcji, w N.M. Josuttis *"C++ Biblioteka standardowa, podręcznik programisty"* autor wyróżnia pięć takich adapterów realizujących następujące złożenia:

$f(g(x))$	(unary_compose)
$f(g(x,y))$	
$f(g(x),h(x))$	(binary_compose)
$f(g(x),h(y))$	

Dwa z nich (`unary_compose` i `binary_compose`) są zdefiniowane w implementacji STL firmy SGI, a więc dostępne razem z kompilatorem `g++`.

Implementacja ich jest analogiczna do implementacji `binder1st`, ale dla wprawy podam tu możliwą implementację złożenia $f(g(x,y))$.

```
template<typename F,typename G> class compose_f_gxy_t {
  typedef typename F::result_type result_type;
  typedef typename G::first_argument_type  first_argument_type;
  typedef typename G::second_argument_type  second_argument_type;

  F _f;
  G _g;

public:
  compose_f_gxy_t(F f,G g):_f(f),_g(g) {};

  result_type operator()(first_argument_type x,
                         second_argument_type y) {
    return _f(_g(x,y));
  }

};

template<typename F,typename G>
compose_f_gxy_t<F,G>
compose_f_gxy(F f,G g) {return  compose_f_gxy_t<F,G>(f,g);};
```

(Źródło: bind.cpp)

Używamy tego funktora następująco:

```
std::cout<<compose_f_gxy(
                         __gnu_cxx::compose1(std::ptr_fun(exp),
                                             std::negate<double>()),
                         f)(1.0,1.0)<<"";
```

(Źródło: bind.cpp)

Powyższe wyrażenie efektywnie produkuje funktor obliczający $\exp(-f(x,y))$ gdzie $f(x,y) = a_x x^2 + a_y y^2 + a_{xy} x * y$. Wykorzystaliśmy w nim szereg konstrukcji

1. Adapter realizujący złożenie dwu funkcji jednoargumentowych `unary_compose` (zwrócony przez funkcję `compose1()`) dostarczony wraz z kompilatorem `g++`.

2. Za pomocą tego adaptera złożyliśmy funkcję $\exp$ ze standardowej biblioteki C z obiektem funkcyjnym `std::negate<double>` predefiniowanym w STL.
3. Aby móc użyć funkcji $\exp$ w adapterze musiałem ją opakować za pomocą adaptera `std::ptr_fun`.
4. Funkcję $\exp(-f)$ złożyłem z $f(x,y)$ za pomocą `compose_f_gxy`.

Na powyższym przykładzie widać siłę, ale i też pewne niedogodności używania funktorów, przynajmniej w takiej postaci, jak zdefiniowanej w STL. Podany kod jest dość rozwlekły i mało czytelny, co gorsza, tak zdefiniowanego funktora nie możemy łatwo przechować w jakiejś zmiennej, bo jego typ też jest bardzo skomplikowany (zobacz zadania).

Klasy cech dla funktorów

Programowanie uogólnione za pomocą funktorów mogłoby być prostsze gdybyśmy posiadali jakiś uniwersalny sposób dostępu do informacji o nich. Taki uniwersalny szkielet funktora z możliwościami introspekcji jest opisany w D. Vandervoorde, N. Josuttis *"C++ Szablony, Vademecum profesjonalisty"*, rozdz.22. Tutaj zaprezentuję tylko możliwą implementację klasy cech dostarczającej informacji o funktorach w stylu STL i wskaźnikach na funkcje.

Załóżmy, że mamy jakiś typ `F` i chcemy się dowiedzieć czy jest on funktorem czy nie. W C++ nie mamy możliwości sprawdzić czy dana klasa posiada operator nawiasów czy nie. Funktory będziemy więc rozpoznawać po posiadanych typach stowarzyszonych. Skorzystamy z zasady rozstrzygania przeciążenia szablonów funkcji: "nieudane podstawienie nie jest błędem". Podobnie jak w przypadku szablonu `Is_class` (zob. wykład 6.2.2 (http://osilek.mimuw.edu.pl/index.php?title=Zaawansowane_CPP/Wyk%C5%82ad_6:_Funkcje_typ%C3%B3w_i_inne_sztuczki#6.2.2_Sprawdzanie_czy_dany_typ_jest_klas.C4.85) wykorzystamy dwa typy:

```
template<typename F> funktor_info {
typedef char one;
typedef struct {char a[2];} two;
```

(Źródło: functor_type.h)

i dwa szablony funkcji:

```
template<typename C> one test_arg(typename C::argument_type *) ;
template<typename C> two test_arg(...) ;
```

(Źródło: functor_type.h)

Za ich pomocą możemy sprawdzić czy dany typ `F` posiada zdefiniowany stowarzyszony typ `argument_type`:

```
enum {has_argument = (sizeof(test_arg<F>(0))==sizeof(one))};
```

(Źródło: functor_type.h)

Podobnie możemy zdefiniować jeszcze trzy stałe logiczne:

```
enum {has_result = (sizeof(test_res<F>(0))==sizeof(one))};
enum {has_first_argument  = (sizeof(test_arg1<F>(0))==sizeof(one))};
enum {has_second_argument = (sizeof(test_arg2<F>(0))==sizeof(one))};
```

(Źródło: functor_type.h)

Pary funkcji `test_...` są zdefiniowane analogicznie do `test_arg`. Za pomocą tych stałych możemy wyrazić inne własności obiektu `F`, np:

```
  enum {has_one_argument  = has_argument && !has_first_argument
       && !has_second_argument};
  enum {has_two_arguments = has_first_argument && has_second_argument
       && !has_argument};
  enum {has_no_arguments = !has_argument && !has_first_argument
       && !has_second_argument};
  enum {is_generator = has_result && has_no_arguments};
  enum {is_unary_function = has_result && has_one_argument};
  enum {is_binary_function = has_result && has_two_arguments};
  enum {is_functor = is_generator || is_unary_function || is_binary_function};
  enum {is_function= false};
```

(Źródło: functor_type.h)

Funkcje możemy rozpoznawać za pomocą specjalizacji częściowych:

```
template<typename A1,typename A2 , typename R >
struct functor_info<R (*)(A1,A2) >{

  enum {has_result  = true};
  enum {has_argument  = false};
  enum {has_first_argument  = true};
  enum {has_second_argument  = true};
... tak samo jak powyżej
};
```

(Źródło: functor_type.h)

Podobnie dla funkcji zero- i jednoargumentowych. Mając już klasę `functor_info` można zdefiniować klasę

```
template<typename F,int n_args = functor_info<F>::n_args>
struct functor_traits ;
```

i jej specjalizacje:

```
template<typename F> struct functor_traits<F,2> {
  typedef typename F::result_type result_type;
  typedef typename F::first_argument_type  arg1_type;
  typedef typename F::second_argument_type arg2_type;
  typedef std::binary_function<arg1_type,arg2_type,result_type> f_type;
  enum {n_args=2};
};
template<typename F> struct functor_traits<F,1> {
  typedef typename F::result_type result_type;
  typedef typename F::argument_type  arg1_type;
  typedef empty_type  arg2_type;
  typedef std::unary_function<arg1_type,result_type> f_type;
  enum {n_args=1};
};
template<typename F> struct functor_traits<F,0> {
  typedef typename F::result_type result_type;
  typedef empty_type   arg1_type;
  typedef empty_type  arg2_type;
  typedef generator<result_type> f_type;
  enum {n_args=0};
};
```

(Źródło: functor_type.h)

Podobnie dla wskaźników do funkcji:

```
template<typename R,typename A1,typename A2>
struct functor_traits<R (*)(A1,A2),2> {
  typedef R   result_type;
  typedef A1  arg1_type;
  typedef A2  arg2_type;
  typedef std::binary_function<arg1_type,arg2_type,result_type> f_type;
  enum {n_args=2};
};
...
```

(Źródło: functor_type.h)

Jeśli teraz użyjemy klasy `functor_traits` w definicji adaptera `binder1st`:

```
template<typename F> class binder1st {
  typedef  typename functor_traits<F>::arg1_type  bind_type;
  typedef  typename functor_traits<F>::arg2_type first_argument_type;
  typedef  typename functor_traits<F>::res_type result_type;

  const bind_type _val;
  F _op;
  public:
  binder1st(F op,bind_type val):_op(op), _val(val) {};

  result_type operator()(first_argument_type x) {
    return _op(_val,x);
  }
};

template<typename F>
binder1st<F>
bind1st(F op,typename functor_traits<F>::arg1_type val) {
return binder1st<F>(op,val);
};
```

to bedziemy mogli używać go z wskaźnikami do funkcji bez konieczności opakowywania ich w adapter `ptr_fun`.

Można stosunkowo łatwo rozszerzyć kod adaptera `binder1st` tak, aby można go było używać zarówno do funktorów dwu- jak i jednoargumentowych.

Biblioteki funktorów

Kod przedstawiony w poprzednim podrozdziale daje, mam nadzieję, pewne wyobrażenie o tym, jak można implemetować bardziej zaawansowane funktory i operacje na nich. Widać jednak, że nie jest to zbyt proste: kod szybko staje się skomplikowany i trudny do debugowania.

Na szczeście istnieją już gotowe implementacje. Jak już wspomiałem propozycje bardziej rozwiniętego szkieletu funktorów znajdą państwo w D. Vandervoorde, N. Josuttis *"C++ Szablony, Vademecum profesjonalisty"*, rozdz. 22. Ponadto biblioteka `boost` oferuje szereg narzędzi, w tym biblioteki `lambda` i `bind`. O tej pierwszej już wspominałem przy omawianiu szablonów wyrażeń. Biblioteka `lambda` dostarcza funkcjonalności adapterów `bind...` i `compose...` za pomocą wyrażenia jednego wyrażenia `bind`. Korzystając z niego, można kod podany w poprzednim podrozdziale zapisać następująco:

```
#include<boost/lambda/lambda.hpp>
#include<boost/lambda/bind.hpp>
using namespace boost::lambda;

  double x=1;
  std::cout<<bind(f,1.0,_1)(x)<<"";
  std::cout<<bind(f,1.0,_1)(make_const(2.0))<<"";
  wyrażenie z biblioteki lambda nie przyjmują stałych stąd
```

```
  konieczność użycia zamienej x, lub wyrażenia makeconstant()
  std::cout<<bind(exp,-bind(f,_1,_2))(x,x)<<"";
```

(Źródło: bind_lambda.cpp)

- Tę stronę ostatnio zmodyfikowano o 13:42, 22 lis 2006;