Zaawansowane CPP/Wykład 6: Funkcje typów i inne sztuczki

From Studia Informatyczne

< Zaawansowane CPP

Spis treści

Wprowadzenie

Funkcja jest podstawowym pojęciem w większości języków programowania, z którym wszyscy jesteśmy dobrze obeznani. Funkcje przyjmują zestaw argumentów i zwracają jakąś wartość. Szablony dają ciekawą możliwość interpretowania ich jako funkcji typów: funkcje których argumentem są typy, a wartością zwracaną typ lub jakaś wartość. Weźmy dla przykładu szablon `sum_traits` z poprzedniego modułu. Można interpretować go jako dwie funkcje przyjmujące typ poprzez parametr szablonu i zwracające albo wartość

```
sum_traits<int>::zero();
```

albo typ

```
sum_traits<int>::sum_type
```

Takie funkcje możemy definiować poprzez wypisanie wszystkich specjalizacji pełnych jak w przypadku `sum_traits` albo przez wykorzystanie specjalizacji częściowych, jak np. dla `iterator_traits`. Najciekawsza możliwość to jednak pisanie "obliczalnego kodu" takich funkcji, przy czym "obliczenia" dokonują się w czasie kompilacji. Przykłady takich funkcji zostaną podane na tym wykładzie. W realnych zastosowaniach wykorzystuje się kombinację wszystkich powyższych możliwości.

Korzystając z szablonów można również uogólnić pojęcie listy i definiować listy typów. Umożliwiają one między innymi indeksowany dostęp do swoich składowych, przekazywanie zmiennej liczby parametrów typu do szablonów i automatyczną generację hierarchii klas opartych na tych typach. Listy typów omówię w drugiej części tego wykładu.

Funkcje typów

If-Then-Else

Zacznę od przydatnej konstrukcji implementującej możliwość wyboru jednego z dwu typów na podstawie stałej logicznej (typu `bool`). Dokonujemy tego za pomocą szablonu podstawowego

```
template<bool flag,typename T1,typename T2> struct If_then_else {
typedef T1 Result;
};
```

który będzie podstawiany dla wartości `flag=true` i jego specjalizacji dla wartości `flag=false`:

```
template<typename T1,typename T2>
struct If_then_else<false,T1,T2> {
typedef T2 Result;
};
```

Teraz możemy go np. wykorzystać do wybrania większego z dwu typów:

```
template<typename T1,typenameT2> struct Greater_then {
typedef typename If_then_else<sizeof(T1)>sizeof(T2),T1,T2>::result
 result;
};
```

Sprawdzanie czy dany typ jest klasą

Następny przykład to szablon służący do sprawdzania czy dany typ jest klasą. Wykorzystamy w tym celu operator `sizeof()` i przeciążane szablony funkcji razem z zasadą "nieudane podstawienie nie jest błędem" (zob. wykład 3.5 (http://osilek.mimuw.edu.pl/index.php?title=Zaawansowane_CPP/Wyk%C5%82ad_3:_Szablony_II#prz.3.5)). Potrzebujemy więc wyrażenia, które nie ma sensu dla dla typów nie będacych klasami. Takim wyrażeniem będzie `int C::*` oznaczające wskaźnik do pola składowego klasy `C` typu `int`. Cały szablon wygląda następująco.

```
template<typename T> class Is_class {
  //najpierw definiujemy dwa typy różniące sie rozmiarem
  typedef char one;
  typedef struct {char c[2];} two;
  //teraz potrzebne bedą dwa przeładowane szablony
  template<typename U> static one test(int U::*);
  template<typename U> static two test(...);
  //to który szablon został wybrany sprawdzamy poprzez sprawdzenie rozmiaru zwracanego typu
  enum {yes = (sizeof(test<T>(0))==sizeof(one) )};
};
```

(Źródło: is_class.cpp)

Operator `sizeof(test<T>(0))` musi rozpoznać typ zwracany przez funkcję `test<T>(0)`. W tym celu uruchamiany jest mechanizm rozstrzygania przeciążenia. Jeśli typ `T` nie jest klasą to próba podstawienia pierwszej funkcji spowoduje powstanie niepoprawnej konstrukcji `T::*` i podstawienie się nie powiedzie. Druga funkcja ma argumenty pasujące do czegokolwiek więc jej podstawienie zawsze się powiedzie. Druga funkcja zwraca typ o rozmiarze większym od rozmiaru typu `one` więc stała `yes` otrzyma wartość `false`.

Jeśli typ `T` jest klasą to `int T::*` jest poprawne (na tym etapie nie jest istotne czy klasa w ogóle posiada taką składową) i mechanizm rozstrzygania będzie pracował dalej, sprawdzając czy argument wywołania jest zgodny z `int T::*`. W tym przypadku jest to zero, które może być użyte jako wskaźnik, więc podstawienie się powiedzie. Oczywiście drugie podstawienie też by się powiodło ale jest mniej specjalizowane. W wyniku

zmienna `yes` otrzyma wartość `true`. Warto zauważyć, że gdybyśmy zamiast zera podstawili jako argument funkcji `test<T>(int T::*)` jakąś inna liczbę całkowitą to podstawienie się nie powiedzie i zawsze otrzymamy fałsz dla zmiennej `yes`.

Sprawdzanie możliwości rzutowania

Kolejny przykład wykorzystuje tę samą technikę w celu stwierdzenia czy jeden z typów można rzutować na drugi.

```
template<typename T,typename U> class Is_convertible {
typedef char one;
typedef struct {char c[2];} two;
tym razem korzystamy ze zwykłych przeciążonych funkcji
static one test(U) ;
static two test(...);
static T makeT();

public: enum {yes = sizeof(test(makeT()) )==sizeof(one),
  same_type=false }; };
```

(Źródło: convertible.cpp)

Teraz sprawdzane są dwie przeciążone funkcje. `makeT()` zwraca obiekt typu `T` więc jeśli typ `T` może być rzutowany na `U` to wybrane zostanie dopasowanie pierwszej funkcji jako bardziej wyspecjalizowanej. Funkcja `makeT()` została użyta zamiast np. `T()`, bo konstruktor defaultowy może dla tego typu nie istnieć. Dodatkowa specjalizacja

```
template<typename T> class Is_convertible<T,T> {
public:
  enum {yes = true,
      same_type=true };
};
```

(Źródło: convertible.cpp)

pozwala nam użyć tej klasy do identyfikacji identycznych typów.

Zdejmowanie kwalifikatorów

Każdy typ w C++ może być opatrzony dodatkowymi kwalifikatorami, takimi jak `const` czy `&` (referencja). Mając dany typ `T` dodanie do niego kwalifikatora jest proste. Z pozoru jednak może się to wiązać z możliwością wygenerowania niepoprawnych podwójnych kwalifikatorów np. `const const T`. Okazuje się jednak, że o ile

```
const const int i =0;
```

jest konstrukcją nieprawdłową, to w przypadku argumentów szablonu nadmiarowe kwalifikatory zostaną zignorowane. Wyrażenie:

```
template<typename T> struct const_const {
const T t = T();
};
const_const<const int> a;
```

jest poprawne i pole `t` będzie typu `const int`. To samo tyczy się referencji. Pomimo tych udogodnień może być konieczna operacja usunięcia jednego lub obydwu kwalifikatorów i uzyskanie gołego typu podstawowego.

W tym celu możemy zdefiniować szablon (zob. D. Vandervoorde, N. Josuttis *"C++ Szablony, Vademecum profesjonalisty"*, rozdz. 17)

```
template<typename T> struct Strip {
  typedef T arg_t;
  typedef T striped_t;
  typedef T base_t;
  typedef const T   const_type;

  typedef T&        ref_type;
  typedef T&        ref_base_type;
  typedef const T & const_ref_type;
};
```

i jego specjalizację dla typów z kwalifikatorem `const`

```
template<typename T> struct Strip< T const> {
  typedef const T arg_t;
  typedef       T striped_t;
  typedef typename Strip<T>::base_t  base_t;
  typedef T const   const_type;

  typedef T const & ref_type;
  typedef T &       ref_base_type;
  typedef T const & const_ref_type;
};
```

i dla referencji

```
template<typename T> struct Strip<T&> {

  typedef T& arg_t;
  typedef T  striped_t;
  typedef typename Strip<T>::base_t  base_t;
  typedef base_t const    const_type;

  typedef T               ref_type;
  typedef base_t &        ref_base_type;
  typedef base_t const &  const_ref_type;
};
```

(Źródło: strip.cpp)

Proszę zwrócić uwagę na konstrukcję

```
typedef typename Strip<T>::base_t  base_t;
```

Ponieważ typ `T` może oznaczać typ kwalifikowany za pomocą `const`, wykorzystujemy rekurencyjne odwołanie aby uzyskać typ podstawowy. Jest to przykład techniki metaprogramowania, która bardziej szczegółowo będzie omówiona w wykładzie 8 (http://osilek.mimuw.edu.pl/index.php?title=Zaawansowane_CPP/Wyk%C5%82ad_8:_Metaprogramowanie).

Cechy typów

Jednym z rodzajów klas cech (omawianych w poprzednim wykładzie) są cechy typów. Nie są one specjalizowane do jednego konkretnego celu, ale służą do dostarczania ogólnych informacji na temat dowolnych typów. Wymieniamy je w tym wykładzie, ponieważ do ich implementacji często stosowane są różne techniki opisane powyżej. Bardzo dobry szczegółowy opis implementacji cech typów znajduje się w D. Vandervoorde, N. Josuttis *"C++ Szablony, Vademecum profesjonalisty"*. Gotową implementację typu

`boost::type_traits` można znaleźć w repozytorium `boost`. Inna - to biblioteka `Loki` opisana w A. Alexandrescu *"Nowoczesne projektowanie w C++"*.

Cechy promocji

Załóżmy, że postanowiliśmy zaimplementować możliwość dodawania wektorów:

```
template<typename T> std::vector<T>
operator+(const std::vector<T>  &a,
          const std::vector<T>  &b) {

  assert(a.size()==b.size());

  std::vector<T> res(a);
  for(size_t i=0;i<a.size();++i)
    res[i]+=b[i];

  return res;
}
```

(Źródło: promote.cpp)

Teraz polecenia

```
std::vector<double> x(10);
std::vector<double> y(10);

x+y;
```

(Źródło: promote.cpp)

skompilują się, ale

```
std::vector<int>    l(10);

x+l;
```

już nie. Ponieważ dodawanie `double` do `int` jest dozwolone, to rozsądne by było aby nasza implemenatacja też na to zezwalała. Przerabiamy więc nasz `operator+()` (lub dodajemy przeciążenie):

```
template<typename T,typename U> std::vector<T>
operator+(const std::vector<T>  &a,
          const std::vector<U>  &b) {

  assert(a.size()==b.size());

  std::vector<T> res(a);
  for(size_t i=0;i<a.size();++i)
    res[i]+=b[i];

  return res;
}
```

Kłopot z tą definicją to typ zwracany, który zależy teraz od kolejności składników dodawania, a nie od ich typu. Abu rozwiązać ten problem zdefiniujemy sobie klasę cech, która będzie określała typ wyniku na podstawie typu składników. Wybierzemy następującą strategię: jeśli typy mają różny rozmiar to wybieramy typ większy, jeżeli mają ten sam rozmiar to liczymy na specjalizacje:

```
template<typename T1,typename T2> struct Promote {
typedef typename
```

```
        If_then_else<(sizeof(T1) > sizeof(T2)),
                     T1,
                     typename If_then_else< (sizeof(T1)< sizeof(T2)),
                                            T2,
                                            void>::Result >::Result Result;
};
```

(Źródło: promote.cpp)

Dla identycznych typów wynik jest jasny (choć jak przekonuje nas przykład 5.1 (http://osilek.mimuw.edu.pl/index.php?title=Zaawansowane_CPP/Wyk%C5%82ad_5:_Klasy_cech#prz.5.1) typ sumy nie musi być typem składników), ale niemniej musimy go zdefiniować:

```
template<typename T> struct Promote<T,T> {
  typedef T Result;
};
```

(Źródło: promote.cpp)

Resztę musimy definiować ręcznie korzystając ze specjalizacji pełnych. Można to sobie uprościć definiując makro

```
#define MK_PROMOTE(T1,T2,Tr)               \
    template<> class Promotion<T1, T2> {   \
      public:                              \
        typedef Tr Result;                 \
    };                                     \
                                           \
    template<> class Promotion<T2, T1> {   \
      public:                              \
        typedef Tr Result;                 \
    };

MK_PROMOTE(bool, char, int)
MK_PROMOTE(bool, unsigned char, int)
MK_PROMOTE(bool, signed char, int)
```

(Źródło: promote.cpp)

Listy typów

Definicja

Listy typów są ciekawą konstrukcją zaproponowaną przez Alexandrescu. Ich szczegółowy opis i zastosowania można znaleźć w A. Alexandrescu *"Nowoczesne projektowanie w C++"*. Definicja listy typów opiera się na rekurencyjnej implementacji listy znanej miedzy innymi z `Lisp-a`: lista składa się z pierwszego elementu (głowy) i reszty (ogona), co możemy zapisać jako:

```
template<typename H,typename T> struct Type_list {
typedef H head;
typedef T tail;
}
```

(Źródło: typelist.h)

Do tego potrzebna nam jest jeszcze definicja pustego typu:

```
class Null_type {};
```

jako znacznika końca listy. I tak:

```
Type_list<int,Null_type>
```

oznacza jednoelementową listę `(int)`

```
Type_list<double,Type_list<int,Null_type> >
```

listę dwuelemntową `(double,int)` i tak dalej.

Length

Ta prosta struktura daje zadziwiająco wiele możliwości. Na początek napiszemy funkcję zwracającą długość listy. W tym celu skorzystamy z rekurencyjnej definicji: długość listy to jeden plus długość ogona, długość listy pustej wynosi zero. Tę definicję implementujemy następująco:

```
template<typename T> struct Length;
template<> struct Length<Null_type> {
enum {value = 0};
};

template<typename H,typename T> struct Length<Type_list<H,T> > {
enum {value = 1 + Length<T>::value};
};
```

(Źródło: typelist.h)

Indeksowanie

Tą samą techniką możemy zaimplementować indeksowany dostęp do elementów listy. Znów korzystamy z rekurencji: element o indeksie `i` to albo głowa (jeśli `i==0`), albo element o indeksie `i-1` w jej ogonie.

```
template<typename T,size_t I> struct At;

template<typename T,typename H> struct At<T,0> {
typedef T result;
};

template<typename T,typename H> struct At<T,I> {
typedef typename At<H,I-1>::result result;
};
```

(Źródło: typelist.h)

Generowanie switch-a

W bibliotece `boost` jest zaimplementowana klasa `any`, której obiekty mogą reprezentować dowolną wartość (zob. boost (http://www.boost.org/doc/html/any.html)). Żeby taką wartość odczytać musimy użyć odpowiedniej konkretyzacji szablonu funkcji `any_cast`:

```
boost::any val;
cout<<any_cast<int>(val)<<endl;
```

Jeśli w szablonie podstawimy nie ten typ co trzeba, to otrzymamy wyjątek. Chcemy teraz napisać funkcję, która drukuje wartości typu `any`, przy czym wiemy, że przechowywane są w nich wartości tylko kilku wybranych typów (np. `int`, `double`, `string`). Ponieważ `any` udostępnia informację o typie przechowywanej w niej wartości moglibyśmy taką funkcję `print_any()` zaimplementować następująco:

```
print_any(std::ostream &of,boost::any val) {
  if(val.type()==typeid(int) )
       of<<any_cast<int>(val)<<std::endl;
     else if val.type()==typeid(double) )
       of<<any_cast<double>(val)<<std::endl;
     else if val.type()==typeid(string) )
       of<<any_cast<string>(val)<<std::endl;
     else
       of<<"unsuported type"<<std::endl;
}
```

Sprobujemy teraz zaimplementować to samo za pomocą list typów, dodatkowo zażądamy aby móc drukować również wartości typu `vector<T>`, gdzie `T` jest typem z listy.

Jak zwykle musimy sformułować problem rekurencyjnie: sprawdzamy czy typ `val` jest typem głowy listy; jeśli tak to drukujemy, jeśli nie to próbujemy drukować `val` używając ogona listy.

```
template<typename T> void print_val(std:ostream &of,boost::any val) {
typedef typename T::Head Head;
typedef typename T::Tail Tail;

if(val.type()==typeid() ) {
           of<<any_cast<Head>(val)<<std::endl;
         }
   else if (val.type()==typeif(std::vector<Head>))
      {
        of<<any_cast<std::vector<Head> >(val)<<std::endl;
      }
      else
      print_val<Tail>(of,val);
}
```

(Źródło: any_print.h)

Potrzebujemy jeszcze warunku kończącego rekurencję

```
template<> void print_val<Null_type>(std::ostream &of,boost::any) {
of<<"don't know how to print this"<<std::endl;
}
```

i możemy już używać naszego szablonu:

```
typedef
TypeList<double,
        Type_list<int,
                  Type_list<string,
                            Null_type> > > my_list;

print_val<my_list>(val);
```

(Źródło: typelist.cpp)