Zaawansowane CPP/Wykład 5: Klasy cech

From Studia Informatyczne

< Zaawansowane CPP

Spis treści

Wprowadzenie

Rozważmy próbę implementacji ogólnej funkcji sumowania elementów tablicy (zob. D. Vandervoorde, N. Josuttis *"C++ Szablony, Vademecum profesjonalisty"*, rozdz. 15). Korzystając z wiadomości o szablonach i konwencjach używanych w STL możemy napisać:

```
template<typename T> T sum(T *beg,T *end) {
  T total = T();
  for(;beg!=end;++beg)
    total += *beg;
  return total;
}
```

(Źródło: sum1.cpp)

PRZYKŁAD 5.1

Ten prosty kod ma jednak co najmniej dwa problemy. Pierwszy związany jest z linijką

```
T total = T();
```

i wiąże się z ustaleniem zerowej wartości dla danego typu. Powyższa linijka oznacza, że zmienna `total` jest inicjalizowana konstruktorem domyślnym klasy `T`. W przypadku typów wbudowanych będzie to inicjalizacja wartością zerową, czyli tak jak tego oczekujemy. W przypadku innych typów możemy mieć tylko nadzieję, że konstruktor defaultowy istnieje i robi to co trzeba :). Popatrzmy na możliwe alternatywy:

```
T total;
```

W przypadku typów zdefiniowanych przez użytkownika wywoływany jest defaultowy konstruktor (domyślny jeśli żaden inny nie jest zdefinowany). W przypadku typów wbudowanych wartość jest niekreślona!

```
T total = 0;
```

jest z kolei niepoprawne dla typów, na które nie ma rzutowania z liczb całkowitych.

Problem można ominąć jeżeli się zauważy, że dla niepustych zakresów tzn. `beg!=end` nie potrzebujemy wcale wartości zerowej:

```
template<typename T> T sum(T *beg,T *end) {
  T total= *beg;
  ++beg;
  while(beg != end ) {
     total += *beg; beg++;
  }
  return total;
}
```

Jeśli jednak dopuszczamy podanie zakresu pustego, to funkcja powinna zwrócić zero i problem powraca.

Drugi problem z przykładem 5.1 to typ zmiennej `total`. Popatrzmy na zastosowanie funkcji `sum`.

```
char name[]="@ @ @";
int length=strlen(name);
cout<<sum(name,&name[length]]);
```

Programik powinien wypisać sumę wartości znaków w napisie `"name"`. Łatwo sprawdzić że wypisuje znak o kodzie zero. Problem polega na tym, że typ `T` niekoniecznie musi pomieścić wynik dodawania elementów typu `T`. W tym przykładzie dodawanie znaków dało wynik 256 (co za niezwykły zbieg okoliczności), który już nie mieści się w zakresie tego typu.

Prostym rozwiązaniem jest dodanie dodatkowego parametru szablonu:

```
template<typename R,typename T> R sum(T *beg,T *end) {
  R total = R();
  while(beg != end ) {
    total += *beg; beg++;
  }
  return total;
}
```

(Źródło: sum2.cpp)

i wtedy zastosowanie

```
cout<<sum<int>(name,&name[length]])<<endl;
```

da już oczekiwany wynik. Zaletą tego rozwiązania jest jego prostota i duża elastyczność. Wadą - zwiększenie liczby parametrów szablonu, co zawsze zwiększa złożoność kodu i możliwości popełnienia błędu, zwłaszcza, że typ `R` jest w większości przypadków określony przez typ `T` i nie wnosi niezależnej informacji.

Klasy cech

Pomocą mogą służyć klasy cech: klasy, których funkcją jest dostarczanie dodatkowych informacji o danym typie. W naszym przypadku możemy zadeklarować szablon:

```
template<typename T>  struct sum_traits;
```

i jego specjalizacje:

```
template<>  struct sum_traits<char> {
typedef int sum_type;
```

```
};
template<>  struct sum_traits<int> {
typedef long int sum_type;
};
template<>  struct sum_traits<float> {
typedef double sum_type;
};
template<>  struct sum_traits<double> {
typedef double sum_type;
};
```

Szablon `sum` przerabiamy teraz na

```
template<typename T>
typename sum_traits<T>::sum_type sum(T *beg,T *end) {
  typedef typename sum_traits<T>::sum_type sum_type;
  sum_type total = sum_type();
  while(beg != end ) {
    total += *beg; beg++;
  }
  return total;
}
```

(Źródło: sum3.cpp)

Wadą tego podejścia jest konieczność definiowania specjalizacji szablonu `sum_traits` dla każdego typu, którego sumę będziemy chcieli obliczyć. Można tego uniknąć definiując szablon ogólny

```
template<typename T>  struct sum_traits {
typedef T sum_type;
};
```

i możemy już wtedy użyć

```
complex<double> *c1,*c2;
sum(c1,c2);
```

bez dodatkowych definicji. To czy należy implementować uniwersalną definicję klasy cech zależy od tego czy istnieje sensowna wartość domyślna dla danej cechy. W naszym przypadku, definiując powyższy szablon, zyskujemy na wygodzie ale tracimy na bezpieczeństwie, bo łatwiej jest teraz wywołać funkcję `sum` z nieodpowiednim typem zmiennej `total`.

Cechy wartości

Możemy spróbować rozwiązać za pomocą klas cech również problem inicjalizacji zmiennej `total`, definiując w każdej klasie odpwiednią wartość zerową dla danego typu. Pytanie jak to zrobić? Nasuwa się użycie stałych składowych statycznych:

```
template<>  struct sum_traits<char> {
typedef int  sum_type;
static const sum_type zero = 0;
};
```

Sęk w tym, że standard zezwala na incijalizowanie w klasie statycznych stałych jedynie dla typów całkowitoliczbowych. Taka sama konstrucja dla `double` już nie jest możliwa.

```
template<>  struct sum_traits<float> {
typedef double  sum_type;
```

```
static  const sum_type zero = 0.0; niedozwolone
};
```

Inicjalizator

```
const typename sum_traits<float>::sum_type
sum_traits<float>::zero = 0.0;
```

musi być umieszczony w kodzie źródłowym. Po pierwsze nie bardzo wiadomo gdzie go umieścić (nie może być w pliku nagłówkowym, bo łamało by to zasadę jednokrotnej definicji). Po drugie kompilator najprawdopodobniej nie umiałby powiązać nazwy stałej i jej wartości w czasie kompilacji.

Inną możliwością jest użycie funkcji statycznych rozwijanych w miejscu wywołania:

```
template<>  struct sum_traits<char> {
typedef int  sum_type;
static  sum_type zero() {return 0;}
};
template<>  struct sum_traits<float> {
typedef double  sum_type;
static  sum_type zero() {return 0.0;}
};
```

Odpowiadający temu podejściu kod funkcji `sum` bedzie wyglądał następująco:

```
template<typename T>
typename sum_traits<T>::sum_type sum(T *beg,T *end) {
  typedef typename sum_traits<T>::sum_type sum_type;
  sum_type total = sum_traits<T>::zero();
  while(beg != end ) {
    total += *beg; beg++;
  }
  return total;
}
```

Dobry kompilator powinien bez trudu rozwinąć definicję funkcji i podstawić odpowiednią wartość bezpośrednio w kodzie.

Parametryzacja klasami cech

Opisana powyżej implementacja funkcji `sum` i związanej z nią klasy `sum_traits` jest mało elastyczna. Wybierając typ przekazanej tablicy wybieramy typ zmiennej `total`. Może się jednak zdażyć, że chcemy sumować `int` we `float`, a `float` we `float`.

Możemy dodać dodatkowy parametr do szablonu, który będzie definiował wybraną klasę cech. Ale to jest powrót do rozwiązania odrzuconego na początku. Rozwiązaniem może być uczynienie tego parametru parametrem domyślnym, tak, aby nie trzeba było podawać go jawnie w typowych przypadkach. Jest to bardzo dobre rozwiązanie w przypadku użycia klas cech w szablonach klas. Problem w tym, że szablony funkcji nie dopuszczają stosowania parametrów domyślnych. Możemy to obejść za pomocą przeciążenia definiując:

```
template<typename Traits,typename T >
typename Traits::sum_type sum(T *beg,T *end) {
  typedef typename Traits::sum_type sum_type;
  sum_type total = sum_type();
  while(beg != end ) {
    total += *beg; beg++;
  }
  return total;
};
```

```
template<typename T >
typename sum_traits<T>::sum_type sum(T *beg,T *end) {
  return sum<sum_traits<T>, T>(beg,end);
}

struct char_sum {
  typedef char sum_type;
};
```

(Źródło: sum4.cpp)

```
main() {
char name[]="@ @ @";
int length=strlen(name);

 cout<<(int)sum(name,&name[length])<<endl;
 cout<<(int)sum<char_sum>(name,&name[length])<<endl;
 cout<<(int)sum<char>(name,&name[length])<<endl;
}
```

(Źródło: sum4.cpp)

iterator_traits

Na koniec spróbujmy uogólnić funkcję `sum`, aby działała nie tylko ze wskaźnikami, ale i iteratorami.

```
template<typename IT> sum(IT *beg,IT *end);
```

Widać, że tu użycie klas cech jest już niezbędne, musimy bowiem dowiedzieć się na obiekty jakiego typu wskazuje iterator. Nie można do tego celu użyć typów stowarzyszonych `IT::value_type`, bo jako iterator może zostać podstawiony zwykły wskaźnik. Dlatego w STL istnieje klasa `iterator_traits`, definiująca podstawowe typy dla każdego rodzaju iteratora. Korzystając z tej klasy można zdefiniować ogólny szalon funkcji `sum`

```
template<typename II>
typename
sum_traits<typename iterator_traits<II>::value_type>::sum_type
sum(II beg,II *end) {
  typedef typename iterator_traits<IT>::value_type value_type;
  typedef typename sum_traits<value_type>::sum_type sum_type;
  sum_type total = sum_traits<value_type>::zero();
  while(beg != end ) {
    total += *beg; beg++;
  }
  return total;
}
```

Zanim omówię klasę `iterator_trais` podam rozwiązanie zastosowane w STL. Tam funkcja nazywa się `accumulate` i jest zaimplementowana następująco:

```
template <class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init) {
     for (; first != last; ++first)
       init = init + *first;
     return init;
   }
```

PRZYKŁAD 5.2

Dodanie dodatkowego parametru wywołania funkcji rozwiązuje za jednym zamachem oba nasze problemy: parametr ten dostarcza zarówno typu, jak i wartości początkowej dla zmiennej sumującej. Są jednak inne algorytmy w STL, które wymagają więcej informacji o iteratorze i muszą je pobrać za pomocą `iterator_traits`.

Dla iteratorów nie będących wskaźnikami `iterator_traits` po prostu przepisują ich typy stowarzyszone:

```
template <class Iterator>
  struct iterator_traits {
    typedef typename Iterator::iterator_category iterator_category;
    typedef typename Iterator::value_type        value_type;
    typedef typename Iterator::difference_type   difference_type;
    typedef typename Iterator::pointer           pointer;
    typedef typename Iterator::reference         reference;
  };
```

Dla typów wskaźnikowych jest podana odpowiednia specjalizacja.

```
template <class T>
  struct iterator_traits<T*> {
    typedef random_access_iterator_tag iterator_category;
    typedef T                          value_type;
    typedef ptrdiff_t                  difference_type;
    typedef T*                         pointer;
    typedef T&                         reference;
  };
```

Widać więc, że każdy iterator nie będący wskaźnikiem musi mieć zdefiniowane odpowiednie typy stowarzyszone. Ułatwia to szablon klasy `iterator`, z którego można dziedziczyć:

```
namespace std {
template<class Category, class T, class Distance = ptrdiff_t,
class Pointer = T*, class Reference = T&>
struct iterator {
typedef T value_type;
typedef Distance difference_type;
typedef Pointer pointer;
typedef Reference reference;
typedef Category iterator_category;
};
```

Na uwagę zasługuje typ `iterator_category`. Ten typ służy do automatycznego wyboru odpowiednich funkcji w oparciu o kategorię iteratora. Kategorie odpowiadają konceptom iteratorów i są reprezentowane przez puste klasy. W STL zdefiniowano pięć kategorii iteratorów:

```
namespace std {
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag: public input_iterator_tag {};
struct bidirectional_iterator_tag: public forward_iterator_tag {};
struct random_access_iterator_tag: public bidirectional_iterator_tag {};
}
```

Aby zilustrować zastosowanie typu `iterator_category` przedstawię implementację funkcji `distance()`, która oblicza odległość pomiędzy dwoma iteratorami. Potrzeba użycia `iterator_category` bierze się stąd, że dla iteratorów o dostępie swobodnym możemy policzyć ją bezpośrednio przez odejmowanie:

```
template <class _RandomAccessIterator>
inline
typename iterator_traits<_RandomAccessIterator>::difference_type
__distance(_RandomAccessIterator __first,
```

```
            _RandomAccessIterator __last,
              random_access_iterator_tag) {

  return __last - __first;
}
```

Dla reszty musimy po kolei zwiekszać jeden iterator aż osiągniemy drugi:

```
template <class _InputIterator>
inline
typename iterator_traits<_InputIterator>::difference_type
__distance(_InputIterator __first,
            _InputIterator __last,
              input_iterator_tag)
{
  typename iterator_traits<_InputIterator>::difference_type __n = 0;
  while (__first != __last) {
    ++__first; ++__n;
  }
  return __n;
}
```

Do wyboru pomiędzy tymi dwoma implementacjami służy właśnie `iterator_category`:

```
template <class _InputIterator>
inline
typename iterator_traits<_InputIterator>::difference_type
distance(_InputIterator __first, _InputIterator __last) {
typedef
     typename
     iterator_traits<_InputIterator>::iterator_category
    _Category;

  return __distance(__first, __last, _Category());
}
```

numeric_limits

Przykładem klasy cech zawartej w standardzie C++ jest szablon `numeric_limits`, który zastępuje używane w C makra, zdefiniowane w pliku `limits.h`. Szablon `numeric_limits` posiada specjalizację dla każdego typu podstawowego wbudowanego (zob. tab. 5.1]) i zawiera informację na temat różnych cech ich implementacji (zob. tab. 5.2]). Warto zwrócić uwagę na następującą konstrukcję: szablon `numeric_limits` definiuje stałą logiczną `is_specialised`. Domyślnie jest ona równa `false`. Każda specjalizacja szablonu ustawia ją na `true`. W ten sposób stała `std::numeric_limits<T>::is_specialised` mówi nam czy dany typ jest opisany przez `numeric_limits` czy nie.

```
namespace std {
template<class T> class numeric_limits;
enum float_round_style;
enum float_denorm_style;
template<> class numeric_limits<bool>;
template<> class numeric_limits<char>;
template<> class numeric_limits<signed char>;
template<> class numeric_limits<unsigned char>;
template<> class numeric_limits<wchar_t>;
template<> class numeric_limits<short>;
template<> class numeric_limits<int>;
template<> class numeric_limits<long>;
template<> class numeric_limits<unsigned short>;
template<> class numeric_limits<unsigned int>;
template<> class numeric_limits<unsigned long>;
template<> class numeric_limits<float>;
template<> class numeric_limits<double>;
```

```
template<> class numeric_limits<long double>;
}
```

Tablica 5.1. Specjalizacje szablonu `numeric_limits`

```
namespace std {
template<class T> class numeric_limits {
public:
static const bool is_specialized = false;
static T min() throw();
static T max() throw();
static const int digits = 0;
static const int digits10 = 0;
static const bool is_signed = false;
static const bool is_integer = false;
static const bool is_exact = false;
static const int radix = 0;
static T epsilon() throw();
static T round_error() throw();
static const int min_exponent = 0;
static const int min_exponent10 = 0;
static const int max_exponent = 0;
static const int max_exponent10 = 0;
static const bool has_infinity = false;
static const bool has_quiet_NaN = false;
static const bool has_signaling_NaN = false;
static const float_denorm_style has_denorm = denorm_absent;
static const bool has_denorm_loss = false;
static T infinity() throw();
static T quiet_NaN() throw();
static T signaling_NaN() throw();
static T denorm_min() throw();
static const bool is_iec559 = false;
static const bool is_bounded = false;
static const bool is_modulo = false;
static const bool traps = false;
static const bool tinyness_before = false;
static const float_round_style round_style = round_toward_zero;
};
}
```

Tablica 5.2. Szablon `numeric_limits`