c++ type_traits

Piotr Padlewski

Warsaw C++ Users Group
13.05.2013

Plan prezentacji

- Iteratory w stl - jak zostały zaprojektowane,
- przeładowanie szablonów funkcji,
- własny type_trait

Cel

Optymalna wersja std::swap(lhs, rhs)

- tam gdzie się da - wywołaj lhs.swap(rhs)
- gdzie się nie da - normalny swap

Słowem wstępu

Biblioteka <type_traits> powstała z myślą rozwiązania problemów generycznego kodu opartego na szablonach.

type_traits pojawił się w standardzie C++11, jednak wcześniej objawił się światu w bibliotece boost.

iteratory

Co łączy te dwie funkcje?

- std::distance(Iter first, Iter last)
- std::advance(Iter it, int distance)

Obie używają +=, -=, -, + dla random access iteratorów, oraz ++, -- dla innych

iterator_category

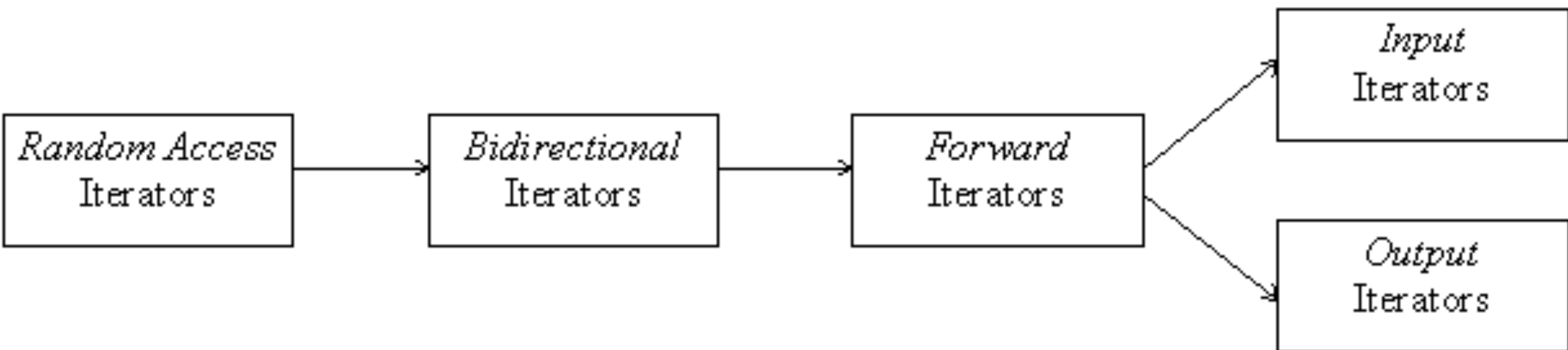
Iteratory muszą mieć

```
typedef xxx_iterator_tag iterator_category;
```

iterator_category

```
struct iterator {
    typedef random_access_iterator_tag iterator_category;
    // ...różne funkcje
};
```

iterator_category



⟶ means, iterator category on the left satisfies the requirements of all iterator categories on the right

iterator_category

```
struct input_iterator_tag { };

struct output_iterator_tag { };

struct forward_iterator_tag : input_iterator_tag { };

struct bidirectional_iterator_tag : forward_iterator_tag { };

struct random_access_iterator_tag : bidirectional_iterator_tag { };
```

std::distance

```
template <typename Iter>
int distance(Iter lhs, Iter rhs) {
	if (typeid(Iter::category) ==
			typeid(random_acces_iterator_tag))
		return rhs - lhs;
	else {
		int result = 0;
		Iter temp = lhs;
		for (; temp != rhs ; ++temp, ++result) {}
		return result;
	}
}
```

Auć

to nie python!

Rozwiązanie te nie skompiluje się dla typu iteratora innego niż swobodny
(brak operatora -)

implementacja std::distance

```
template <typename Iter>
int distance(Iter lhs, Iter rhs)
{
    return doDistance(lhs, rhs,
            typename Iter::iterator_category());
}
```

dla iteratorów swobodnych

```
template <typename Iter>
int doDistance(Iter lhs, Iter rhs,
                         const random_access_iterator_tag&)
{
    return rhs - lhs;
}
```

dla forward iteratorów

```
template <typename Iter>
int doDistance(Iter lhs, Iter rhs, const &forward_iterator_tag)
{
    int result = 0;
    Iter temp = lhs;
    for (; temp != rhs ; ++temp, ++result) {}
    return result;
}
```

A co ze wskaźnikami?

```
int *ptr1, *ptr2;
…
int d = std::distance(ptr1, ptr2); //Błąd kompilacji
```

Rozwiązanie?

- specjalizacja distance dla wskaźników?
  - spowoduje dwuznaczność
  - częściowa specjalizowacja szablonów funkcji jest zabroniona

- przeładowanie funkcji dla wskaźników?
  - przeładowywać każdą funkcje?

iterator_traits

iterator_traits<Iterator> jest “proxy” dla cech iteratorów.

Jest wyspecjalizowana dla wskaźników, aby nadać im porządane cechy.

iterator_traits

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

specjalizacja dla wskaźników

```
template<typename T>
struct iterator_traits<T*>   //to samo dla const T*
{
    typedef random_access_iterator_tag        iterator_category;
    typedef T                                 value_type;
    typedef ptrdiff_t                         difference_type;
    typedef T*                                pointer;
    typedef T&                                reference;
};
```

distance z iterator_traits

```
template <typename Iter>
typename iterator_traits<Iter>::difference_type
distance(Iter lhs, Iter rhs)
{
        return doDistance(lhs, rhs,
                typename iterator_traits<Iter>::iterator_category());
}
```

Jak pisać własne iteratory?

- dziedziczyć po std::iterator który zapewnia typedefy

```
template <class Category,
              class T,
              class Distance = ptrdiff_t,
              class Pointer = T*, class Reference = T&>
        struct iteratotor;
```

- używać boost::iterator

przeładujmy jakąś funkcje!

```
template <typename T>
class vector
{
    size_t size_;
    T *ptr_;
public:
    explicit vector(size_t size = 0, const T& value = T());

    template <typename Iter>
    vector(Iter first, Iter last);
};
```

implementacja konstruktora

```
vector::vector(size_t size = 0, const T& value = T());
        : size_(size),
        ptr_(new T[size])
    {
        std::fill(ptr_, ptr_ + size_, value);
    }
```

implementacja konstruktora

```
template <typename Iter>
  vector::vector(Iter first, Iter last)
      : size_(std::distance(first, last)),
      ptr_(new T[size_])
  {
      std::copy(first, last, ptr_);
  }
```

Co się stanie gdy...

```
vector<int> v(42, 101010);
```

error: no matching function for call to ‘distance(int&, int&)’

```
        : size_(std::distance(first, last))
```

Odpalił się konstruktor dla iteratorów!

```
explicit vector(size_t size = 0, const T& value = T()); //1
template <typename Iter>
vector(Iter first, Iter last); //2

vector<int> v(42, 101010);
```

funkcja 1 potrzebuje konwersji inta na unsigned inta, dlatego kompilator woli wybrać wersje 2

is_arithmetic

Kiedy nie powinniśmy wywołac konstruktora 2?

- Kiedy typ Iter jest typem arytmetycznym takim jak int, uint, long long, …

Z pomocą przychodzą nam type_traits

integral_constant

```
template <class T, T v>
struct integral_constant {
  static constexpr T value = v;
  typedef T value_type;
  typedef integral_constant<T,v> type;
  constexpr operator T() { return v; }
};
```

Służy do tworzenia różnych typów dla różnych wartości typu T

```
typeid(integral_constant<bool, true>()) != typeid(integral_constant<bool, false>())
```

true_type i false_type

Pomocnicze typedefy

```
typedef integral_constant<bool, true> true_type;
typedef integral_constant<bool, false> false_type;
```

Tworząc jakiś type_trait dziedziczymy po integral_constant jak i true_type i false_type aby uzyskać potrzebne typedefy

is_arithmetic

```
template<class T>
struct is_arithmetic : std::integral_constant < bool,
        is_integral<T>::value || is_floating_point<T>::value > {};
```

is_integral<T> sprawdza czy T jest liczbą całkowitą

is_floating_point<T> sprawdza cz T jest liczbą zmiennoprzecinkową

is_integral

```
template <typename T>
struct is_integral : false_type {}

//specjalizacja dla typów całkowitych
template<>
struct is_integral<int> : true_type {}
```

… i tak dla każdego typu całkowitoliczbowego (15 typów)

Odwołania się do innej funkcji

```
void foo(size_t size, int val)
{
    _dispath_foo(size, val, std::true_type());
}

template <typename T>
void foo(T first, T last)
{
    _dispath_foo(first, last, typename is_arithmetic<T>::type());
}
```

Odwołania się do innej funkcji

```
template <typename T>
void _dispath_foo(T, T, const std::false_type&)
{
    cout << "foo(T, T)" << endl;
}

void _dispath_foo(size_t, int, const std::true_type&)
{
    cout << "foo(size_t, int)" << endl;
}
```

Rozwiązanie nr. 2 - SFINAE

SFINAE - Substitution failure is not an error

Jeśli nie uda się podstawić wszystkich parametrów funkcji która pasuje najbardziej, wtedy nie jest to błędem, i wybierana jest następna w kolejności sygnatura.

enable_if

```
template<bool Cond, class T = void> struct enable_if
{
};

template<class T> struct enable_if<true, T>
{
    typedef T type;
};
```

enable_if - sposób działania

enable_if<false>::type nie skompiluje się - enable_if<false> nie posiada zagnieżdżonego typu type

natomiast enable_if<true>::type już się skompiluje

Rozwiązanie 2.

```
void foo(size_t, int)
{
    std::cout << "foo(size_t, int)";
}

template <typename T>
void foo(T, T, typename enable_if<!is_arithmetic<T>::value>::type* = 0)
{
    std::cout << "foo(T, T)" << endl;
}
```

Rozwiązanie 2.

Rozwiązanie to działa, lecz jest

- bardzo brzydkie,
- nieczytelne,
- daje użytkownikowi możliwość dostania się do 3. wartości (void*)

Rozwiązanie 3.

```
void foo(size_t, int)
{
    std::cout << "foo(size_t, int)";
}

template <typename T>
typename enable_if<!is_arithmetic<T>::value, void>::type
foo(T, T)
{
    std::cout << "foo(T, T)" << endl;
}
```

Rozwiązanie 3.

Rozwiązanie wyrzuca 3. argument tak że użytkownik nie ma do niego dostępu, jednak jest

- nadal mało czytelne,
- wymaga aby funkcja coś zwracała (nie można zastosować dla konstruktorów)

Rozwiązanie 4.

```
void foo(size_t, int)
{
    cout << "foo(size_t, int)" << endl;
}

template <typename T,
    typename = typename enable_if<!is_arithmetic<T>::value>::type
    >
void foo(T, T)
{
    std::cout << "foo(T, T)";
}
```

```
template <typename T>
class vector
{
    size_t size_;
    T *ptr_;
public:
    explicit vector(size_t size = 0, const T& value = T())
        : size_(size),
        ptr_(new T[size])
    {
        std::fill(ptr_, ptr_ + size_, value);
    }

    template <typename Iter,
              typename = typename std::enable_if<
                  !std::is_arithmetic<Iter>::value>::type
             >
    vector(Iter first, Iter last)
        : size_(std::distance(first, last)),
        ptr_(new T[std::distance(first, last)])
    {
        std::copy(first, last, ptr_);
    }
};
```

Jak działają proste type_traits?

is_class<T> - czy T jest klasą/strukturą

```
namespace detail {
    template <class T> char test(int T::*);
    struct two { char c[2]; };
    template <class T> two test(...);
}

template <class T>
struct is_class : std::integral_constant<
                                    bool, sizeof(detail::test<T>(0))==1
                                    && !std::is_union<T>::value
                                    > {};
```

std::swap

- Wszystkie kontenery w stl mają w sobie metodę
  void swap(Container& c) // gdzie C to np vector, string
  która jest najczęściej szybsza niż zwykły swap(lhs, rhs)

- Mają one również przeładowaną funkcje
  std::swap(lhs, rhs)
  tak że wywołuje ona lhs.swap(rhs)

std::swap

- Przeładowanie std::swap jest nie tylko lukrem syntaktycznym - std::swap jest również używane przy np sortowaniu.

- Dla własnej klasy posiadającej metodę swap można także przeładować std::swap. Jest jednak kilka przyczyn dla czego jest to złe.

std::swap

- Jedną z przyczyn jest to, że nie powinno się dodawać nic do przestrzeni nazw std.

- Jeśli przeładowuje się swapa to można to zrobić poza przestrzenią std

std::swap

```
swap(MyType lhs, MyType rhs) { …}

template <typaname T>
void luckySort(T& lhs, T& rhs)
{
      using std::swap;
      swap(a, b);
}
```

```
MyType a, b;
int c, d;

// wywoła przeładowany swap
luckySort(a, b)

//wywoła std::swap
luckySort(c, d)
```

std::swap

Takie przeładowanie powiela jednak ciągle ten sam kod.

Czy można było zatem zrobić to lepiej?

Można!

```
template <typename T>
void swap(T& lhs, T& rhs)
{
    typedef typename has_swap<T>::type type;
    swap_aux(lhs, rhs, type());
}
```

Można!

```
template <typename T>
void swap_aux(T& lhs, T& rhs, std::true_type)
{
    lhs.swap(rhs);
}
```

```
template <typename T>
void swap_aux(T& lhs, T& rhs, std::false_type)
{
    std::swap(lhs, rhs);
}
```

- Jedyne co nam zostało do zrobienia to napisać has_swap<T>

has_swap

```
template <typename T>
struct has_swap : std::integral_constant<bool,
                                          has_swap_helper<T>::value>
{
};
```

```
template<typename Type>
class has_swap_helper {
    template <typename F, F x>
    struct sfinae { };

    template <typename U>
    static char deduce(U*, sfinae<void (Type::*)(Type&), &U::swap>* = 0);
    static int deduce(...);
public:
    static const bool value = sizeof(deduce((Type*)(0))) == sizeof(char);
};
```

Dygresja na koniec

Taki kod nam się nie skompiluje

```
std::list<int> l(...);
std::sort(l.begin(), l.end());
//std::list<int>::iterator nie jest iteratorem swobodnym
```

i jest to bardzo dobre, bo ostrzega nas przed tym że chcemy zrobić coś bardzo głupiego czyli

sortować w O(n^2 log n)

Dygresja na koniec

jednak już coś takiego się skompiluje

```
std::set<int> s(...);
int x = 42;
std::set<int>::iterator it = std::lower_bound(s.begin(), s.end(), x);
```

Powyższy kod będzie się wykonywać w czasie O(n), gdzie wywołanie s.lower_bound(x) zadziałałoby w O(log n). Dlaczego zatem stl pozwala na coś tak głupiego, skoro można by użyć std::find(s.begin(), s.end(), x) ?

Dygresja na koniec

Jedynym sensownym wytłumaczeniem jest to że std::
lower_bound wykona minimalną ilość porównań, czyli może się to przydać jeśli operacja porównania jest o wiele bardziej kosztowna od przesunięcia iteratora.

Większość się jednak zgodzi z tym że fajniej by było jakby kompilator się na czymś takim wykrzaczył, albo żeby wywołanie std::set<int>::iterator it = std::lower_bound(s.begin(), s.end(), x);
wywołało s.lower_bound(x)

Dziękuję za uwagę!