Zaawansowane CPP/Wykład 7: Klasy wytycznych

From Studia Informatyczne

< Zaawansowane CPP

Spis treści

Wprowadzenie

Klasy wytycznych, nazywane również klasami reguł (policy classes) służą do parametryzowania zachowania innych klas. Rozważmy przykład funkcji `accumulate`. Posiada ona również przeciążoną wersję umożliwiającą postawienie dowolnej operacji zamiast dodawania:

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

Jedyna zmiana w implementacji klasy w stosunku do przykładu 5.2 (http://osilek.mimuw.edu.pl/index.php?title=Zaawansowane_CPP/Wyk%C5%82ad_5:_Klasy_cech#prz.5.2) to zmiania operacji sumowania na:

```
       init = binary_op(init, *first);
```

Pomimo, że pojawił się dodatkowy szablon klasy, to nie jest to typowa klasa wytycznych. Zachowanie jest określone nie tyle przez ten parametr, co przez funktor przekazany jako argument wywołania.

Możemy jednak zmienić trochę implementację:

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

Takiego szablonu możemy używać następująco:

```
template<typename T> Sumation {
static op(const T &a,const T &b) {
         return a+b;
       }
};
```

```
accumulate<Summation<double> >accumulate(first,last,0.0);
```

Klasa (szablon) `Summation` jest właśnie klasą wytycznych. W zasadzie nie ma powodów aby implementować funkcję `accumulate` za pomocą klas wytycznych, poza być może nadzieją na trochę bardziej efektywny kod.

W następnej części przedstawię bardziej realistyczny, ale i bardziej skomplikowany przykład.

Projektowanie za pomocą klas wytycznych

Problemem w uniwersalnych bibliotekach jest duża ilość możliwych implementacji pojedynczego komponentu. Dzieje się tak kiedy implementując komponent możemy podjąć kilka prawie niezależnych od siebie decyzji. Projektując kontener mamy np. do wyboru różne sposoby alokacji pamięci i różne strategie obsługi błędów. Te zagadanienia są w dużej mierze ortogonalne do siebie. Jeśli więc mamy trzy strategie przydziału pamięci i trzy strategie obsługi błędu, to w sumie dostajemy dziewięć możliwych kombinacji. Decyzja o możliwości pracy w środowisku wielowątkowym zwiększą tę liczę dwukrotnie.

Klasy wytycznych mogą pomóc opanować ten kombinatoryczny wzrost ilości możliwości. Idea polega na tym, aby za każdą decyzję odpwiedzialną zrobić jedną klasę wytyczną, przekazywaną jako parametr szablonu. W przytoczonym przykładzie szablon kontenera mógłby posiadać dwa parametry wytycznych

```
template<typename T,
         typename Allocator_policy,
         typename Checking_policy>
 Kontener;
```

i potrzebowalibyśmy 6 (3 `Allocator_policy` i 3 `Checking_policy`) różnych klas wytycznych. Sześć może się wydawać niewiele mniejsze od dziewięciu, ale takie podejście skaluje się liniowo z liczbą wytycznych: dodanie nowej strategii alokacji pamięci wymaga jednej dodatkowej klasy, a liczba kombinacji zwieksza się do 12. W praktyce wszystkie wytyczne miałyby wartości domyślne.

Stack

Pokażę teraz jak to działa w praktyce na przykładzie znanego już nam szablonu klasy `Stack`, w którym na początek dokonam drobnych zmian:

```
template<typename T = int , size_t N = 100> class Stack {
private:
 T rep[N];
 size_t _top;
public:
 Stack():_top(0) {}
 void push(const T &val) {_rep[_top++]=val;}
 void pop()              {--_top;}
 const T& top()  const   {return _rep[top-1];}
 bool is_empty           {return !_top;}
}
```

Zmiany polegają na rozdzieniu operacji odczytywania wierzchołka stosu i zdejmowania elementu ze stosu. Umożliwia to między innymi przekazywanie wartości zwracanej ze stosu przez referencje, poza tym jest to bardziej bezpieczne.

Ten kod jest, delikatnie rzecz ujmując, bardzo prościutki. Możemy rozbudowywać go w co najmniej dwu kierunkach. Po pierwsze można użyć dynamicznej alokacji pamięci, po drugie możemy zaimplementować sprawdzanie zakresu aby wykryć próbę włożenia elementu na pełny stos, lub zdjęcia/odczytania elementu ze stosu pustego. W tym przypadku mamy różne możliwości reakcji na te błędy.

Żeby zaimplementować sprawdzanie zakresu dodajemy nowy parametr do szablonu `Stack`, który będzie określał klasę wytyczną dla tej strategii:

```
template<typename T = int , size_t N = 100,
         typename Checking_policy = No_checking_policy >
class Stack {
private:
  T _rep[N];
  size_t _top;
public:
  Stack():_top(0) {};

  void push(const T &val) {

    Checking_policy::check_push(_top,N);
    _rep[_top++]=val;
  }

  void pop()              {
    Checking_policy::check_pop(_top);
    --_top;
  }

  const T& top()  const   {
    Checking_policy::check_top(_top);
    return _rep[top-1];
  }

  bool is_empty()         {
    return !_top;
  }

};
```

(Źródło: stack1.h)

Klasa

```
struct No_checking_policy {
  static void check_push(size_t,size_t) {};
  static void check_pop(size_t) {};
  static void check_top(size_t) {};
};
```

(Źródło: checking_policy.h)

implementuje najprostszą strategię sprawdzania zakresu: brak sprawdzania. Proszę zauważyć, że w tym wypadku najprawdopodobniej żaden kod nie zostanie dodany: kompilator "wyoptymalizuje" puste funckje.

Inne możliwe strategie to np.

```
class Abort_on_error_policy {
public:
  static void check_push(size_t top,size_t size) {

    if(top >= size) {
      std::cerr<<"trying to push elemnt on full stack: aborting"<<std::endl;
      abort();
    }
  };

  i podobnie dla pozostałych funkcji sprawdzających
};
```

(Źródło: checking_policy.h)

Programując w C++ wstyd by było nie użyć wyjątków:

```
struct Std_exception_on_error_policy {

  static void check_push(size_t top,size_t size) {

    if(top >= size) {
      throw std::range_error("over the top");
    }
  };

  i podobnie dla pozostałych funkcji sprawdzających

};
```

(Źródło: checking_policy.h)

Teraz możemy prosto konfigurować szablon `Stack` podając mu odpowiednie argumenty:

```
Stack<int,10>                                    s_no_check;
Stack<double ,100,Abort_on_error_policy>         s_abort;
Stack<int *,25,Std_exception_on_error_policy>    s_except;
```

(Źródło: policy1.cpp)

W celu zaimplementowania różnych strategii przydziału pamięci dodajemy dodatkowy parametr szablonu, który sam będzie szablonem:

```
template<typename T = int , size_t N = 100,
         typename Checking_policy = No_checking_policy,
         template<typename U,size_t M>  class Allocator_policy
         = Static_table_allocator > class Stack;
```

Szablon typu `Allocator_policy` posiada jeden typ stowarzyszony i szereg funkcji:

```
template<typename T,size_t N = 0> struct Static_table_allocator {
         typedef T rep_type[N];
         void init(rep_type &,size_t) {};
         void expand_if_needed(rep_type &,size_t) {};
         void shrink_if_needed(rep_type &size_t) {};
         void dealocate(rep_type &){};

         static size_t size() {return N;};

};
```

(Źródło: allocator2.h)

Szablon `Stack` implementujemy teraz następująco:

```
template<...> class Stack {

  typedef typename Allocator_policy<T,N>::rep_type rep_type;
  rep_type  _rep;
  size_t _top;
  Allocator_policy<T,N> alloc;
public:
  Stack(size_t n = N):_top(0) {
```

```
    alloc.init(_rep,n);
  };

  void push(const T &val) {
    alloc.expand_if_needed(_rep,_top);
    Checking_policy::check_push(_top,alloc.size());
    _rep[_top++]=val;
  }

  void pop()              {
    Checking_policy::check_pop(_top);
    --_top;
    alloc.shrink_if_needed(_rep,_top);
  }

  const T& top()  const   {
    Checking_policy::check_top(_top);
    return _rep[top-1];
  }

  bool is_empty()         {
    return !_top;
  }

  ~Stack() {alloc.dealocate(_rep);}

};
```

(Źródło: stack2.h)

Szablon

```
template<typename T,size_t N > struct Expandable_new_allocator {
  typedef T * rep_type;
  size_t _size;
  void init(rep_type &rep,size_t n) {_size=n;rep = new T [_size];};
  void expand_if_needed(rep_type & rep,size_t top) {
    if(top == _size) {
      _size=2*_size;
      T *tmp= new T[_size];
      std::copy(rep,&rep[top],tmp);
      delete [] rep;
      rep = tmp;
    }
  };
  void shrink_if_needed(rep_type &size_t) {
  };
  void dealocate(rep_type &rep){delete [] rep;};

  size_t size() const {return _size;};
};
```

(Źródło: allocator2.h)

definiuje strategię dynamicznego przydziału pamięci "na żądanie". Możemy teraz dowolnie składać nasze strategie:

```
  int n=10;
  Stack<int,0,Std_exception_on_error_policy,Expandable_new_allocator > s1(n);
  Stack<int,10,Abort_on_error_policy,Static_table_allocator > s2(n);
```

(Źródło: policy2.cpp)

Widać, że takie podejście jest bardzo elastyczne, użytkownik może praktycznie dowolnie konfigurować sobie zachowanie klasy `Stack`, zwłaszcza, że ma możliwość tworzenia własnych klas wytycznych.

Oczywiście powyższy przykład nie jest do końca dopracowany. Przede wszystkim strategie przydziału pamięci i strategie sprawdzenia zakresu nie są całkowicie niezależne. Np. w funkcji `push` jeśli powiedzie się wywołanie funkcji `expand_if_needed()` to nie ma potrzeby wywoływania funkcji `check_push()`. Po drugie - całkowicie pominęliśmy kwestię diagnostyki funkcji alokujących pamięć. Możliwe rozwiązanie to przekazanie `Checkin_policy` jako argumentu szablonu do `allocator_policy`. Można też rozważyć posiadanie dwu różnych klas wytycznych, jednej dla obsługi błędów przekroczenia zakresu, drugiej do obsługi błędów przydziału pamięci.

Dziedziczenie wytycznych

Stosowanie wytycznej `Checking_policy` sprowadzało się do używania funkcji statycznych. W przypadku wytycznej `Allocator_policy` musieliśmy utworzyć obiekt tej klasy, ponieważ niektóre implementacje tej wytycznej posiadają stan (w tym przypadku jest to zmienna `_size`). Alternatywnym sposobem użycia takiej wytycznej jest wykorzystanie dziedziczenia:

```
template<typename T = int , size_t N = 100,
         typename Checking_policy = No_checking_policy,
         template<typename U,size_t M>  class Allocator_policy
         = Static_table_allocator >
class Stack: private Checking_policy, private Allocator_policy<T,N> {

  typedef typename Allocator_policy<T,N>::rep_type rep_type;
  rep_type  _rep;
  size_t _top;
public:
  Stack(size_t n = N):_top(0) {
    init(_rep,n);
  };
  void push(const T &val) {
    expand_if_needed(_rep,_top);
    Checking_policy::check_push(_top,this->size());
    _rep[_top++]=val;
  }
  void pop()              {
    Checking_policy::check_pop(_top);
    --_top;
    this->shrink_if_needed(_rep,_top);
  }
  const T& top()  const   {
    Checking_policy::check_top(_top);
    return _rep[top-1];
  }
  bool is_empty()         {
    return !_top;
  }
  ~Stack() {this->dealocate(_rep);}
};
```

(Źródło: stack3.h)

Główna zmiana to konieczność kwalifikowania nazw niektórych funkcji przez `this->` tak, aby stały się nazwami zależnymi (zob. wykład 3.7.1 (http://osilek.mimuw.edu.pl/index.php?title=Zaawansowane_CPP/Wyk%C5%82ad_3:_Szablony_II#Zale.C5.BCne_klasy_bazowe)). Skorzystałem z dziedziczenia prywatnego aby zaznaczyć, że dziedziczę implementację a nie interfejs (`Stack` **nie jest** `Allocator_policy`). Jednak odziedziczenie również interfejsu klasy `Allocator_policy` poprzez dziedziczenie publiczne może być użyteczne. Aby się o tym przekonać rozważymy kolejną modyfikację naszego przykładu: przeniesiemy zmienną `_rep` z klasy `Stack` do klasy wytycznej np.

```
template<typename T,size_t N > class Dynamic_table_allocator {
protected:
  typedef T * rep_type;
  rep_type _rep;
  size_t _size;
```

```
  void init(size_t n) {_size=n;_rep = new T[_size];};
  void expand_if_needed(size_t) {};
  void shrink_if_needed(size_t) {};
  void dealocate(){delete [] _rep;};

  size_t size() const {return _size;};
public:
  void resize(size_t n) {
      T *tmp= new T[n];
      std::copy(_rep,&_rep[(_size<n)?_size:n],tmp);
      delete [] _rep;
      _rep = tmp;
      _size=n;
  }
};
```

(Źródło: allocator3_1.h)

Pociąga to za sobą zmiany w klasie `Stack`:

```
template<typename T = int , size_t N = 100,
         typename Checking_policy = No_checking_policy,
         template<typename U,size_t M>  class Allocator_policy
         = Static_table_allocator >
class Stack: private Checking_policy, public Allocator_policy<T,N> {

  size_t _top;

  public: Stack(size_t n = N):_top(0) { Stack::init(n); }; void
  push(const T &val) {
    Stack::expand_if_needed(_top);
    Checking_policy::check_push(_top,this->size());
    Stack::_rep[_top++]=val;
  }
  void pop() {
    Checking_policy::check_pop(_top); --_top;
    Stack::shrink_if_needed(_top);
  }
  const T& top() const {
    Checking_policy::check_top(_top);
    return Stack::_rep[top-1]; }
  bool is_empty() { return !_top; }  Stack() {
    Stack::dealocate();}
};
```

(Źródło: stack3_1.h)

Zmieniłem również sposób uzależniania nazw niezależnych na kwalifikację ich nazwą klasy `Stack`. Teraz możemy korzystać z interfejsu klasu `Dynamic_table_allocator` w klasie `Stack`.

```
  Stack<int,n,Std_exception_on_error_policy,Dynamic_table_allocator > s(n);
  s.resize(20);
```

(Źródło: policy3_1.cpp)