Klasy cech w programowaniu generycznym

W języku C++ do tworzenia generycznych algorytmów lub struktur danych używamy szablonów. Artykuł zawiera techniki odpowiadające instrukcji warunkowej, która będzie wykonywana w czasie kompilacji.

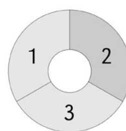
Dowiesz się:

- Jak wybierać algorytm lub wartość w czasie kompilacji;
- Co to są klasy cech (trejty).

Powinieneś wiedzieć:

- Jak pisać proste programy w C++;
- Co to są szablony (templates).

Poziom trudności



Programowanie generyczne w C++(patrz ramka) wykorzystuje podzbiór konstrukcji języka, ponieważ byty, którymi operujemy, muszą mieć ustaloną wartość, znaną podczas kompilacji. Nie można używać zmiennych, nie można wykonywać iteracji (tworzyć pętli) ani używać instrukcji warunkowych. Zamiast tego stosujemy techniki, które dają równoważne efekty. W dalszej części tekstu będzie przedstawione rozwiązanie, pozwalające na wybór algorytmu, typu lub pewnej stałej, w zależności od parametrów szablonu, co w programowaniu generycznym odpowiada instrukcji warunkowej.

Przedstawiona technika do wyboru odpowiedniego algorytmu lub odpowiedniej wartości wykorzystuje dodatkowe klasy nazywane trejtami lub klasami cech. Na Listingu 1 trejtami są klasy `numeric_traits`, dostarczają one stałej `min_value`, o wartości zależnej od parametru szablonu, wykorzystując specjalizację. Funkcja `find_max` znajduje maksymalną wartość w tablicy. Inicjuje ona zmienną `current_max` za pomocą trejtów, a więc różnymi wartościami dla różnych typów.

Trejty albo klasy cech są to typy, których głównym zadaniem jest przechowywanie informacji o innych typach. Mechanizm ten pozwala uporządkować dostęp do stałych, które mają podobne znaczenie. Biblioteka standardowa dostarcza trejtów `std::numeric_limits`, które definiują wartości graniczne dla wbudowanych typów liczbowych. Aby pobrać wartość takiej stałej, piszemy `numeric_limits<double>::min()` zamiast `__DBL_MIN__`, `numeric_limits<int>::min()` zamiast `INT_MIN`, `numeric_limits<int>::max()` zamiast `INT_MAX`, itd. Taki zapis zwalnia programistę z obowiązku wyszukiwania nazwy stałej dla danego typu oraz nagłówka, który ją deklaruje.

Wybór algorytmu w czasie kompilacji

Biblioteka standardowa udostępnia kilka innych trejtów, natomiast nowy standard C++200x będzie zawierał kolejnych kilkadziesiąt, obecnie udostępnianych przez biblioteki boost (`type_traits`, `call_traits`, `function_types`). Biblioteki boost są znanym zbiorem bibliotek eksperymentalnych C++, z których wiele będzie umieszczonych w nowej wersji standardu.

Przykładem wykorzystania trejtu `has_trivial_assign`, dostępnego w omawianym zbiorze, jest funkcja `fastCopy`, pokazana na Listingu 2, która kopiuje tablice, wykorzystując `std::memcpy` (kopiowanie bajtów), jeżeli elementy tablicy są typów, dla których kopiowanie takie jest poprawne, albo algorytm `std::copy`, jeżeli należy wołać operator przypisania dla każdego obiektu. Trejt `has_trivial_assign` bada, czy typ ma trywialny operator przypisania, to znaczy, jeżeli przypisanie dla typu `T` jest równoznaczne z kopiowaniem pamięci zajmowanej przez obiekt, to `has_trivial_assign<T>` jest typu `true_type`, `has_trivial_assign<T>::value` ma wartość `true`, w przeciwnym wypadku trejt dziedziczy po `false_type`, zaś składowa `value` ma wartość `false`.

Funkcja `fastCopy` wykorzystuje dodatkowy, czwarty argument, który jest tworzony w czasie kompilacji na podstawie informa-

Szybki start

Aby uruchomić przedstawione rozwiązania, należy mieć dostęp do dowolnego kompilatora C++ oraz edytora tekstu. Niektóre przykłady zakładają dostęp do bibliotek boost. Warunkiem ich uruchomienia jest instalacja tych bibliotek (w wersji 1.36 lub nowszej) oraz wykorzystywać kompilator oficjalnie przez nie wspierany, to znaczy msvc 7.1 lub nowszy, gcc g++ 3.4 lub nowszy, Intell C++ 8.1 lub nowszy, Sun Studio 12 lub Darvin/GNU C++ 4.x. Na wydrukach pominięto dołączanie odpowiednich nagłówków oraz udostępnianie przestrzeni nazw, pełne źródła dołączono jako materiały pomocnicze.

cji o typie. Jego wartość nie jest istotna, natomiast typ pozwala wybrać odpowiednią funkcję kopiująca. Dodatkowy argument, którego typ jest jedyną istotną informacją jest często stosowaną techniką w programowaniu generycznym.Kompilator wykorzystuje typ argumentu do wyboru odpowiedniej wersji funkcji lub metody, nie zwiększając wielkości kodu wynikowego (optymaliztor będący częścią kompilatora usuwa kod związany z argumentami, które nie są wykorzystywane).

Innym przykładem wykorzystania klas cech jest szablon `getId` dostarczający identyfikatora obiektu. Funkcja ta zwraca identyfikator przechowywany w obiekcie, dla obiektów typu pochodnego po `HasId`, albo adres dla pozostałych obiektów. Aby wybrać odpowiedni sposób ,stosujemy trejt `is_base_of`, patrz Listing 3.

Rozwiązanie wykorzystuje trejt `is_base_of`, zależny od dwu parametrów, dostarczający informacji o tym, czy pierwszy typ jest klasą bazową dla drugiego. Gdy `Base` jest klasą bazową `Derived` ,to `is_base_of<Base, Derived>` jest typu `true_type`, w przeciwnym wypadku `is_base_of<Base, Derived>` jest typu `false_type`. Trejt ten wykorzystujemy do utworzenia pomocniczego obiektu, a następnie przekazujemy go jako dodatkowy parametr, który pozwala wybrać jedną z kilku przeciążonych funkcji w czasie kompilacji. Funkcji `getId` możemy używać dla dowolnych obiektów, uzyskując albo adres, albo wynik wołania metody `getId`.

```
//typ z własnym identyfikatorem
struct ClassWithId : public HasId  {
   ClassWithId(long id) : HasId(id)  { }
};
//typ bez identyfikatora
struct ClassWithoutId { };
ClassWithId c1(1);
   //obiekt z identyfikatorem równym 1
ClassWithoutId c2;
   //obiekt bez identyfikatora
getId(c1); //zwraca wartość 1
getId(c2); //zwraca adres obiektu c2
```

Optymalizacja przy pomocy klas cech

Klasy cech możemy wykorzystywać do optymalizacji przekazywania argumentów. Dla typów użytkownika argumenty powinny być przekazywane przez stałą referencję, ponieważ unika się tworzenia kopii, natomiast dla typów wbudowanych oraz dla wskaźników argumenty przekazujemy przez wartość, ponieważ tworzenie kopii jest mało kosztowne, natomiast referencja wprowadza narzut przy odwoływaniu się do obiektu.

Klasa cech `boost::call_traits`, dostarczana przez biblioteki boost, definiuje między innymi optymalny sposób przekazywania argumentów dla obiektów danego typu. Trejt ten definiuje, oprócz stałych, pewne pomocnicze typy, co pokazano na Listingu 4, pozwalając optymalnie przekazywać parametry. Jeżeli parametrem tego szablonu będzie `int` ,składowa `param_type` będzie definiowała typ `int` (typy wbudowane przekazujemy przez wartość), jeżeli parametrem będzie `Foo` (przykładowy typ użytownika), to `param_type` dostarczy typu `const Foo&`.

Możemy zdefinować nagłówek naszej funkcji tak jak poniżej,

```
template<typename T>
void f(typename call_traits<T>::param_type
                              value) {}
```

wtedy argument będzie przekazywany przez wartość dla typów wbudowanych

Listing 1. *Inicjowanie zmiennej za pomocą trejtów*

```
template <typename T> struct number_traits {
   static const int min_value = 0; //dla dowolnego typu stała ma wartość zero
};
template<> struct number_traits<int> { //specjalizacja dla typu int
   static const int min_value = INT_MIN; //definiuje odpowiednią stałą
};
template<> struct number_traits<long> { //specjalizacja dla typu long
   static const long min_value = LONG_MIN;
};
//znajduje maksymalną wartość w tablicy
template<typename T> find_max(const T* first, const T* last) {
   T current_max = number_traits<T>::min_value; //wykorzystuje trejty
   for( ;first != last; ++first)
      if( current_max < *first )
         current_max = *first;
    return current_max;
 }
```

Szablony– przypomnienie

Szablony (*templates*) dostępne w języku C++ umożliwiają implementację generycznych, to znaczy niezależnych od typów, algorytmów oraz struktur danych. Przykładowy szablon *swap*, pokazany poniżej, zamienia zawartość dwu obiektów, możemy go wołać dla dowolnych obiektów tego samego typu, jeżeli dostarczają one konstruktora kopiującego i operatora przypisania.

```
template<typename T> void swap(T& a, T& b) {
   T tmp = a;
   a = b;
   b = tmp;
}
```

Podczas kompilacji następuje konkretyzacja szablonu, co oznacza generowanie kodu dla właściwych typów. Kod generowany na podstawie szablonów nie różni się od kodu tworzonego *ręcznie*, nie ma żadnych narzutów pamięciowych i czasowych, jedyną niedogodnością jest dłuższy czas kompilacji, ale to zazwyczaj nie jest problemem.

Specjalizacja to wersja szablonu, która będzie użyta do generacji kodu zamiast wersji ogólnej, gdy parametrami będą odpowiednie typy. Przykładem specjalizacji jest szablon `swap<Foo>` pokazany poniżej. Ponieważ typ `Foo` zawiera jedynie wskaźnik na obiekt zawierający składowe (klasa `Foo` ukrywa implementację), wystarczy zamienić te wskaźniki, jeżeli chcemy zamienić zawartość obiektów. Jest to bardziej wydajne niż zamiana przy pomocy obiektu tymczasowego.

```
struct Foo { //przykładowa klasa, która ukrywa implementację
   struct Impl; //klasa wewnętrzna, przechowuje składowe
   Impl* pImpl_; //wskaźnik jest składową publiczną, aby uprościć szablon
};
template<> void swap<Foo>(Foo& a, Foo& b) { //specjalizacja szablonu swap
   Foo::Impl* tmp = a.pImpl_; //zamienia wskaźniki, a nie całe obiekty
   a.pImpl_ = b.pImpl_;
   b.pImpl_ = tmp;
}
```

(oraz wskaźników i referencji) lub przez stałą referencję dla typów użytkownika.

Przy pomocy tego samego trejtu rozwiązuje się problem podwójnej referencji, który wynika z tego, że nie można tworzyć referencji do referencji. Jeżeli szablon używa referencji do typu T, który jest parametrem, to gdy przekażemy typ referencyjny jako parametr następuje błąd kompilacji.Rozwiązanie to wykorzystywanie w szablonach typu `call_traits<T>::reference` zamiast `T&`. Odpowiednia specjalizacja klasy cech `call_traits` zapewni, że jeżeli parametrem będzie typ referencyjny, to referencją będzie ten sam typ.

Trejty możemy stosować, aby zmniejszyć wielkość kodu wynikowego oraz aby optymalizować jego czas wykonania. Dla każdego typu, dla którego szablon został użyty, jest generowany kod, który jest kompilowany i dołączany do wersji binarnej tworzonej aplikacji czy biblioteki. Aby zmniejszyć wielkość kodu wynikowego, stosuje się te same rozwinięcia szablonów dla różnych typów, jeżeli są dozwolone konwersje pomiędzy tymi typami. Dodatkową zaletą tego rozwiązania jest możliwość wyboru typu, dla którego operacje na danej platformie wykonywane są najszybciej. Przykład pokazany na Listingu 5 wykorzystuje trejty do promocji dla liczb rzeczywistych udostępniane przez biblioteki `boost`.

Dla typów reprezentujących liczby rzeczywiste, które można konwertować do `double`, będzie użyty ten sam kod funkcji `complicateCalculationImpl`, ponieważ typ, który jest parametrem tego szablonu, uzyskujemy za pomocą trejtu `promote`. W przedstawionym rozwiązaniu, jeżeli szablonu używamy dla różnych typów, będzie wykorzystywany ten sam kod binarny, który będzie używał obiektów typu najlepiej wspieranego przez daną platformę. Podobną technikę możemy stosować wykorzystując promocję dla typów całkowitych.

Listing 2. *Wykorzystanie klas cech do wyboru algorytmu kopiowania*

```
template<typename T> //kopiowanie za pomocą memcpy
void doFastCopy(const T* first, const T* last, T* result, true_type) {
   memcpy(result, first, (last - first)*sizeof(T) );
}
template<typename T> //kopiowanie za pomocą std::copy
void doFastCopy(const T* first, const T* last, T* result, false_type) {
  std::copy( first, last, result );
}
template<class T> //algorytm wykorzystuje trejty
void fastCopy(const T* first, const T* last, T* result) {
   doFastCopy(first, last, result, has_trivial_assign<T>() ); //tworzy dodatkowy
                     argument
}
```

Listing 3. *Szablon dostarczający identyfikator dla obiektów klasy*

```
class HasId { //klasa dostarczająca identyfikator
public:
   HasId(long id) : id_(id)  { }
   virtual ~HasId() { }
   long getId() const { return id_; }
private:
   long id_;
};
template<typename T> long doGetId(const T& t, true_type)  {
   return t.getId(); //zwraca wewnętrzny identyfikator
}
template<typename T> long doGetId(const T& t, false_type) {
   return reinterpret_cast<long>(&t); //zwraca adres jako wartość long
}
template<typename T> long getId(const T& t) { //wykorzystuje trejty
   return getIdInternal(t, is_base_of<HasId,T>() );
}
```

Listing 4. *Fragment trejtów boost::call_traits*

```
template <typename T> call_traits { //szablon dla typów użytkownika
   typedef const T& param_type; //sposób przekazywania parametrów danego typu
};
template <typename T> call_traits<T*> { //specjalizacja dla wskaźników
   typedef T param_type; //wskaźniki lepiej przekazywać przez wartość
};
```

Listing 5. *Wykorzystanie trejtów promote udostępnianych przez boost::type_traits*

```
template<typename T> T complicateCalculation( T input ) { //tylko woła inną funkcję
   return complicateCalculationImpl(typename promote<T>::type(input) );
}
template<typename T> T complicateCalculationImpl( T input ) {
   //tutaj złożony kod, który oblicza wartość
   //dla typów float i double będzie wykorzystywany ten sam kod binarny
}
```

Podsumowanie

Techniki stosowane w programowaniu generycznym (inna nazwa to programowanie uogólnione) różnią się od tych stosowanych w programowaniu obiektowym i strukturalnym, ich znajomość pozwala zmniejszać rozmiar kodu źródłowego, zwiększając jego czytelność bez wpływu na wydajność. Szablony dają możliwość tworzenia ogólnych rozwiązań, z tego względu technika ta dominuje wśród bibliotek.

W Sieci

- http://www.boost.org;
- http://www.open-std.org;
- http://www.ddj.com/cpp/184404270.

ROBERT NOWAK

Adiunkt w Zakładzie Sztucznej Inteligencji Instytutu Systemów Elektronicznych Politechniki Warszawskiej, zainteresowany tworzeniem aplikacji dla biologii i medycyny, programuje w C++ od ponad 10 lat.

Kontakt z autorem:rno@o2.pl