Stałość

Stałość logiczna i stałość fizyczna

Oznaczanie obiektów, argumentów, metod i składowych jako stałe zwiększa czytelność kodu oraz dostarcza dodatkowych warunków poprawności.

Dowiesz się:
- Dlaczego oznacza się obiekty jako stałe;
- Co to jest stałość logiczna;
- Co to jest stałość fizyczna.

Powinieneś wiedzieć:
- Jak pisać proste programy w C++.

Wprowadzenie

Obiekty w języku C++ mają przypisany typ, który określa zbiór dopuszczalnych operacji. Pozwala to wykrywać przypadki nieprawidłowego użycia obiektów już na etapie kompilacji. Artykuł omawia kwalifikator `const`, oznaczający zobowiązanie do niemodyfikowania obiektu. Klasyfikator ten modyfikuje typ obiektu, pozwalając kompilatorowi na dodatkową kontrolę. Obiekty, które mają typ poprzedzony kwalifikatorem `const`, nazywa się stałymi. Stosowanie stałych zwiększa czytelność kodu, programista wie, że obiekt nie będzie zmieniany, jeżeli został przekazany jako stały.

Prostym zastosowaniem stałych jest zastępowanie tzw. „magicznych" liczb lub „magicznych" napisów. Czytelność kodu jest zwiększana, gdyż nazwa stałej niesie informacje o jej znaczeniu, podatność kodu na zmiany również rośnie, ponieważ można tę samą stałą stosować wielokrotnie i jeżeli zajdzie potrzeba zmian, to wystarczy zmienić wartość w jednym miejscu, co pokazano na Listingu 1. Kompilator może w miejsce obiektu stałego wstawiać odpowiadającą mu wartość.

Dla wskaźników niezależnie określa się, czy stała jest zawartość wskaźnika (przechowywany adres), czy wartość wskazywana (obiekt umieszczony pod adresem przechowywanym we wskaźniku). Najwygodniej zapamiętać składnię, czytając napis od prawej do lewej, tzn. `int const*` oraz `const int*` oznacza wskaźnik do stałej całkowitej (adres można zmienić, wartość wskazywaną – już nie), zaś `int* const` czytamy jako stały wskaźnik do obiektu przechowującego liczbę całkowitą. Obiekt typu `const int* const` jest wskaźnikiem, zabroniona jest zmiana wskaźnika (adresu) oraz wartości wskazywanej.

W kontenerach biblioteki standardowej rolę wskaźników pełnią iteratory i tutaj także możemy definiować dwa rodzaje stałości: stały iterator oraz iterator do stałej. Z tego względu dostarcza się dwóch rodzajów iteratora: iterator do zmiennej, który pozwala na zmianę wartości wskazywanej (`T::iterator`, gdzie `T` jest typem kontenera), oraz iterator do stałej, nie pozwala on modyfikować wartości wskazywanej (`T::const_iterator`). Używając kwalifikatora `const` można utworzyć stały iterator oraz stały iterator do stałej, patrz Listing 2.

Metody, które nie są statyczne, mają dodatkowy (niejawny) argument będący wskaźnikiem na obiekt, na rzecz którego wołamy metodę (wskaźnik `this`). Jeżeli metoda nie modyfikuje tego obiektu, to można ją oznaczyć jako metodę stałą (literał `const` po deklaracji metody). Wtedy obiekt `this` jest wskaźnikiem na stałą (tzn. `const T*`). Dzięki wyróżnieniu metod nieinwazyjnych, zwiększamy czytelność kodu, programista widzi już w deklaracji metody, że nie zmienia ona stanu obiektu. Kompilator pozwala wołać takie metody na rzecz stałych.

Szybki start

Aby uruchomić przedstawione przykłady, należy mieć dostęp do kompilatora C++ oraz edytora tekstu. Na wydrukach pominięto dołączanie odpowiednich nagłówków oraz udostępnianie przestrzeni nazw, pełne źródła dołączono jako materiały pomocnicze.

Oznaczenie metody jako stałej (nieinwazyjnej) zmienia jej sygnaturę, mogą istnieć dwie różne wersje metody o tej samej nazwie i takiej samej liście argumentów, różniące się stałością. Jedna z nich będzie wołana dla stałych, zaś druga dla zmiennych, patrz Listing 3.

Unikanie powielania kodu

Obiekt zawsze można przekształcić w obiekt stały tego samego typu. Często wykonujmy takie przekształcenie właśnie po to, aby wykorzystać kompilator do wykrywania prób zmiany stanu obiektu we fragmentach kodu, które tego robić nie powinny. Próba przekształcenia obiektu stałego na obiekt zmienny tego samego typu (przekształcenie odwrotne), jeżeli jest dokonywane niejawnie, jest niedozwolone. Konieczność wykonania takiego przekształcenia oznacza zazwyczaj błąd projektowy. Przykładem takiej, niedozwolonej konwersji, jest sytuacja, gdy argumentem formalnym funkcji jest `T&` ,a argumentem dostarczanym jest `const T&`. Wystąpi wtedy błąd kompilacji.

W pewnych sytuacjach takie przekształcenie jest przydatne, dlatego język dysponuje operatorem rzutowania `const_cast`. Operator ten jest pomocny, gdy chcemy dostarczyć dwóch wersji metody (wersja dla stałej i wersja dla zmiennej) i chcemy unikać powielania kodu. Przykład przedstawiony na Listingu 4 wykorzystuje ten sam kod w dwóch wersjach metody `get()`. Jeżeli obiekt jest dostarczony jako stały, to wymuszenie traktowania go jako zmienny jest poprawne tylko wtedy, gdy mamy pewność, że obiekt jest zmienną.

Stałość fizyczna i logiczna

Metody oznaczona jako stałe nie mogą modyfikować stanu obiektu. Jest to sprawdzane przez kompilator i jest nazywane stałością fizyczną. Metody stałe mogą jednak zezwalać na zmianę stanu innym fragmen-

Listing 1. *Stałe zwiększają czytelność kodu oraz ułatwiają jego zmiany*

```
const int MAX_VALUE = 5 ;// stała nazwana zamiast „magicznej” liczby 5
for( int i = 0; i < MAX_VALUE; ++i )
   // w tej pętli wykorzystuje się tą stałą
for( int j = 0; j < MAX_VALUE; ++j )
   // tutaj używa się tej samej stałej, wystarczy jej wartość zmienić w jednym miejscu
```

Listing 2. *Stałość w bibliotece standardowej*

```
const int TAB[] =  2, 3, 8, 5 ;// tablica stałych
const int TAB_SIZE = sizeof(TAB)/sizeof(TAB[0]);
std::vector<int> v(TAB, TAB + TAB_SIZE); // inicjacja wektora
const std::vector<int>::iterator it = v.begin(); // stały iterator
++it; // błąd, stały iterator
*it = 5;//można zmieniać wartość wskazywaną
std::vector<int>::const_iterator it2 = v.begin(); //iterator do stałej
++it2;//można zmieniać iterator
*it2 = 8; // błąd, stała wartość wskazywana
const std::vector<int>::const_iterator it3 = v.begin();//stały iterator do stałej
++it3; //błąd
*it3 = 13; //błąd
```

Listing 3. *Metody stałe*

```
class Foo {
public:
   void f() const; //metoda stała, nie zmienia stanu obiektu
   void f(); //metoda, która  nie jest stała (zmienia stan obiektu) o tej samej nazwie i parametrach
   void g(); // metoda g jest dostarczana tylko w wersji inwazyjnej
};
const Foo f; // obiekt f jest stałą
f.f(); // woła metodę 'void f() const'
f.g(); // błąd, próba wołania metody g dla stałego obiektu
```

tom kodu, na przykład wtedy, gdy zwracają wskaźnik lub referencję do składowej. Jeżeli występuje taka sytuacja, to stan obiektu może być zmieniony, gdy wołane są jedynie metody stałe. Jest to błędem nazywanym brakiem stałości logicznej. Błąd ten nie jest wykrywany przez kompilator. Przykład pokazano na Listingu 5.

Stałość logiczna jest pojęciem bardziej ogólnym, niż stałość fizyczna. Oznacza ona brak możliwości zmiany stanu dla stałych obiektów zarówno przez metody danej klasy, jak też przez inne fragmenty kodu. Osiągnięcie stałości logicznej jest pożądane przy tworzeniu oprogramowania. Aby zapewnić ten rodzaj stałości w metodach stałych nie należy zwracać wskaźników ani referencji do składowych, można zwracać wskaźnik do stałej albo stałą referencję. Poprawiony kod klasy `Foo`, zapewniający stałość logiczną, jest pokazany na Listingu 6.

Zazwyczaj, jeżeli zapewniona jest stałość logiczna, występuje także stałość fizyczna (stałość fizyczna oznacza brak modyfikacji składowych w metodach oznaczonych jako stałe). W pewnych szczególnych przypadkach stałość logiczna występuje w metodach, które modyfikują składowe, czyli gdy brak jest stałości fizycznej. Ilustracją tego przypadku jest klasa `Str` przedstawiona na Listingu 7, która obudowuje wskaźnik do napisu w stylu C i dostarcza metodę zwracającą długość napisu.

Metoda `getLength` oblicza długość napisu tylko raz, przy pierwszym wołaniu tej metody, zaś przy każdym kolejnym zwraca zapamiętaną długość. Ze względu na tę optymalizację, metoda nie może być ozna-

Listing 6. *Klasa zapewnia stałość logiczną*

```
class Foo {
   int* x_;
public:
   const int* get() const { // metoda stała pozwala
                            tylko na odczyt
      return x_;
   }
   void set(int* x) {// metoda zmienia stan, nie jest
                     stała
      x_ = x;
   }
};
```

Listing 4. *Wykorzystanie operatora const_cast do dostarczania dwóch wersji metody*

```
class Foo {
public:
   Foo(int v) : val_(v) {}
    int& get() {//metoda wołana dla zmiennej
      // rzutowanie const_cast bezpieczne, metoda stała zwraca referencję do składowej, która może być zmieniana
      // rzutowanie static_cast niezbędne, aby uniknąć rekurencji
      return const_cast<int&>( static_cast<const Foo*>(this)->get() );
   }
   const int& get() const {//metoda wołana dla stałej
       // dodatkowe czynności, np. rejestracja dostępu
       // return val_;
   }
private:
   int val_;
};
```

Listing 5. *Klasa nie zapewnia stałości logicznej*

```
struct Foo {
   int* x_;// możliwy swobodny dostęp do wskaźnika
};
void fun(const Foo& f) { // obiekt f przekazywany jako stała
   int i = 34;
   f.x_ = &i; // błąd, stan f został zmieniony, kompilator pozwoli na to
   *f.x_ = 4; // błąd, jak wyżej
}
```

Listing 7. *Klasa obudowuje napis w stylu C. Metoda getLength nie może być oznaczona jako stała*

```
class Str {
public:
   const char* str() const { // metoda stała
      return str_;
   }
   int getLength() { // metoda nie może być stała
      if(! validLength_) {
         length_ = strlen(str_); // zmienia składową
         validLength_ = true; // zmienia składową
      }
      return length_;
   }
private:
   char* str_;
   bool validLength_;
   int length_;
};
```

Listing 8. *Klasa obudowuje napis w stylu C. Metoda getLength nie może być oznaczona jako stała*

```
class Str {
public:
   const char* str() const { // metoda stała
      return str_;
   }
   int getLength() const { // metoda stała
      if(! validLength_) {
         length_ = strlen(str_);
         validLength_ = true;
      }
      return length_;
   }
private:
   char* str_;
   mutable bool validLength_; // składowa może być
                              zmieniana w metodach stałych
   mutable int length_; // składowa może być
                        zmieniana w metodach stałych
};
```

czona jako stała, ponieważ zmienia ona składowe `validLength_` oraz `length_`. Pomimo tego, że odczyt długości napisu nie zmienia tego napisu, czyli jest zachowana stałość logiczna, nie można wołać tej metody dla stałego obiektu. Aby umożliwić tworzenie metod stałych w takich przypadkach stosuje się modyfikator `mutable`, którym oznacza się składowe, które mogą być zmieniane w metodach stałych. Poprawiony kod klasy `Str` został pokazany na Listingu 8.

Składowymi, które mogą być zmieniane w metodach stałych, są często obiekty synchronizacyjne (np. muteksy) albo obiekty związane z optymalizacjami (tak jak powyżej). Nierozsądne stosowanie modyfikatora `mutable` może doprowadzić do zupełnego braku kontroli nad stałością, dlatego mechanizmu tego powinniśmy używać tylko w uzasadnionych przypadkach.

Obiekty ulotne

Język C++ dostarcza kwalifikatora `volatile`, który oznacza zależność stanu obiektu od innego wątku lub innego procesu. Obiekty oznaczone jako ulotne nie mogą podlegać optymalizacjom. Kwalifikator `volatile`, podobnie jak kwalifikator `const` modyfikuje typ obiektu. Pomiędzy stałością i ulotnością nie ma sprzeczności, obiekt może być jednocześnie stały i ulotny.

Obiekty ulotne, w odróżnieniu od obiektów stałych, są tworzone sporadycznie, dlatego nie będą opisywane tutaj szerzej.

Podsumowanie

Oznaczanie klas, metod, argumentów jako stałe zwiększa kontrolę kompilatora nad dostarczanym kodem. Warto wykorzystywać takie konstrukcje, gdyż pomimo nieco większego nakładu pracy (musimy oznaczać odpowiednie byty jako stałe), pozwalają one wychwycić szereg błędów, związanych z niewłaściwym użyciem obiektów.

Więcej w książce

Zagadnienia dotyczące współcześnie stosowanych technik w języku C++, wzorce projektowe, programowanie generyczne, prawidłowe zarządzanie zasobami przy stosowaniu wyjątków, programowanie wielowątkowe, ilustrowane przykładami stosowanymi w bibliotece standardowej i bibliotekach boost, zostały opisane w książce Robert Nowak, Andrzej Pająk „Język C++: mechanizmy, wzorce, biblioteki", która ukaże się niebawem.

W Sieci

- *http://www.parashift.com/c++-faq-lite* - opis wykorzystywania stałości w C++
- *http://www.possibility.com/Cpp/const.html* – opis wykorzystywania stałości w C++

ROBERT NOWAK

Adiunkt w Zakładzie Sztucznej Inteligencji Instytutu Systemów Elektronicznych Politechniki Warszawskiej, zainteresowany tworzeniem aplikacji bioinformatycznych oraz zarządzania ryzykiem. Programuje w C++ od ponad 10 lat.

Kontakt z autorem:rno@o2.pl