Tworzenie kopii obiektów

Wzorzec prototypu

Kopiowanie obiektów, czyli tworzenie duplikatów, przechowujących te same informacje bez niszczenia oryginału, jest jedną z podstawowych operacji, które wykorzystujemy w programowaniu. Artykuł opisuje tę czynność, analizując techniki wspierające proces tworzenia kopii w języku C++.

Dowiesz się:
- Co to jest leniwe kopiowanie;
- Co to jest wzorzec prototypu;
- Jak stworzyć fabrykę prototypów.

Powinieneś wiedzieć:
- Jak pisać proste programy w C++;
- Co to jest dziedziczenie i funkcje wirtualne;
- Co to są szablony.

Poziom trudności

Kopiowanie obiektów jest operacją wykonywaną bardzo często: przekazując argumenty przez wartość, zwracając wyniki obliczeń, przechowując elementy w kontenerach i w wielu innych sytuacjach są tworzone kopie. Jeżeli obiekt jest dostępny pośrednio, na przykład przez wskaźnik, to można wykonać kopię wskaźnika (lub innego uchwytu) albo całego obiektu. W związku z tym możemy wyróżnić trzy rodzaje kopiowania: kopiowanie płytkie, gdy kopiujemy uchwyty (wskaźniki), kopiowanie głębokie, gdy tworzymy kopię obiektu, oraz kopiowanie leniwe, które łączy kopiowanie płytkie i głębokie. Do demonstracji tych technik będziemy używali klasy `Foo` pokazanej na Listingu 1.

Kopią płytką nazywa się kopiowanie jedynie obiektów pośredniczących, wskaźników, referencji, uchwytów itp. Kopia taka jest tworzona szybko, ponieważ wskaźnik lub inny obiekt pośredniczący jest zazwyczaj małym obiektem. Po wykonaniu płytkiej kopii ten sam obiekt jest dostępny z wielu miejsc, obiekt wskazywany nie jest kopiowany, zmiana jego stanu będzie widoczna we wszystkich kopiach. Głęboka kopia oznacza rzeczywiste kopiowanie obiektów wskazywanych. Tworzenie takiej kopii zajmuje więcej czasu i zasobów, ale obiekt i kopia są od siebie niezależne. Zmiany obiektu nie mają wpływu na kopię. Na Rysunku 1 pokazano zawartość wskaźników po wykonaniu płytkiej i głębokiej kopii, przykład kodu zawiera Listing 1.

Kopiowanie opóźnione

Kopiowanie opóźnione lub leniwe wykorzystuje obie strategie kopiowania opisane powyżej. Na początku wykonujemy kopię płytką, która jest przeprowadzana szybko i umożliwia poprawne odczytywanie informacji przechowywanych w zarządzanym obiekcie. Przy próbie modyfikacji obiektu badamy, czy obiekt jest wskazywany przez jeden, czy przez kilka wskaźników. Jeżeli istnieje tylko jeden wskaźnik, to modyfikacja odbywa się na zarządzanym obiekcie, natomiast jeżeli wskaźników jest więcej, wykonuje się głęboką kopię wskazywanego obiektu i modyfikuje się tę kopię. Leniwe kopiowanie umożliwia więc optymalne połączenie obu strategii, a ceną jest konieczność przechowywania dodatkowej składowej, która pozwala rozstrzygnąć, czy należy robić głęboką kopię. Składową tą jest licznik odniesień lub flaga. Można pozbyć się tej składowej, tworząc głęboką kopię obiektu za każdym razem, gdy wołana jest opera-

Listing 1. *Tworzenie kopii płytkiej i głębokiej*

```
class Foo { //klasa pomocnicza
public:
   Foo() : i_(0) {}
   int get() const { return i_; }
   void set(int i) { i_ = i; }
};
Foo* shellCopy(Foo* f) { //płytka kopia
   return f; //wskaźniki pokazują na ten sam obiekt
}
Foo* deepCopy(Foo* f){ //głęboka kopia
   return new Foo(*f); //wskaźniki pokazują na różne obiekty
}
Foo* p1 = new Foo();
Foo* p2 = shellCopy(p1); //płytka kopia
Foo* p3 = deepCopy(p1); //głęboka kopia
p1->set(2); //zmiana obiektu wskazywanego przez p1
assert( p2->get() == 2 ); //obiekt wskazywany przez p2 został zmieniony
assert( p3->get() == 1 ); //obiekt wskazywany przez p3 nie został zmieniony
```

cja modyfikująca, ale wtedy wiele kopii jest zbędnych.

Przykład leniwego kopiowania został pokazany na Listingu 2. Przedstawiona tam klasa wykorzystuje sprytne wskaźniki `boost::shared_ptr`, które zostały omówione w SDJ 11/2009. Sprytne wskaźniki to szablony, które pozwalają automatycznie usuwać obiekt utworzony na stercie, przechowują one i aktualizują licznik odniesień do wskazywanego obiektu. Szablony te wspierają tworzenie leniwej kopii, dostarczają metodę `unique`, która pokazuje, czy zarządzany obiekt jest wskazywany przez jeden, czy więcej wskaźników. Metoda ta jest wykorzystana w klasie `LazyFoo` do rozstrzygania, czy można modyfikować bieżący obiekt, czy raczej należy zrobić kopię.

Tworząc kopię głęboką obiektu tymczasowego, który będzie usunięty po zakończeniu operacji kopiowania, można wykonać kopię płytką i nie usuwać tego obiektu, co przypomina przeniesienie właściciela obiektu. Taki mechanizm dla wskaźników dostarcza `std::auto_ptr` (SDJ 11/2009), w ogólnym przypadku wymaga on wsparcia w języku. Takie wsparcie będzie dostarczone w nowym standardzie C++200x poprzez referencję do r-wartości, co pozwoli na implementację różnych konstruktorów kopiujących. Używając konstruktora kopiującego do r-wartości, będzie można przenieść zawartość obiektu, unikniemy wtedy zbędnej kopii.

Szybkie kopiowanie głębokie

Dla pewnych typów obiektów kopia głęboka może być wykonana bez użycia konstruktora kopiującego za pomocą operacji kopiujących fragmenty pamięci. Obiekty, które będą w ten sposób kopiowane, nie mogą mieć składowych, które są wskaźnikami, bo wskazywane przez te składowe obiekty także będą musiały być kopiowane przy tworzeniu kopii głębokiej. Informacji o tym, czy obiekt może być kopiowany za pomocą kopiowania bajtów, dostarcza klasa cech (*trejt*) `has_trivial_copy` , który jest dostarczany przez bibliotekę `boost::traits`. Funkcja `fastDeepCopy`, pokazana na Listingu 3, wykorzystuje dodatkowy argument, który jest tworzony w czasie kompilacji na podstawie informacji o typie. Jego wartość nie jest istotna, natomiast typ pozwala wybrać odpowiednią funkcję kopiującą. Jeżeli obiekty mogą być kopiowane za pomocą funkcji kopiującej fragmenty pamięci, to jest ona wołana, w przeciwnym wypadku woła się konstruktor kopiujący. Technika trejtów została opisana w SDJ 11/2009.

Wzorzec prototypu

Jeżeli posługujemy się wskaźnikiem lub referencją do klasy bazowej, to możemy wykonać jedynie płytką kopię. Kopia głęboka jest niedostępna, ponieważ przy tworzeniu obiektu należy podać konkretny typ (patrz SDJ 2/2010), a my dysponujemy tylko typem interfejsu. Rzeczywisty typ obiektu może być inny.

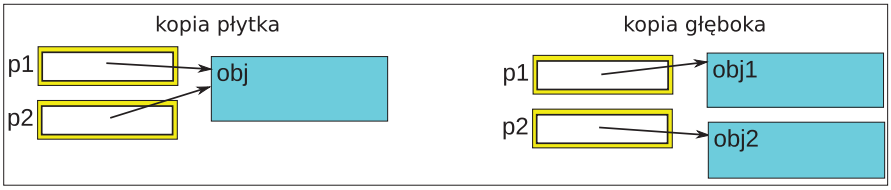
Wzorzec prototypu, nazywany też wirtualnym konstruktorem, opisany w książce „Wzorce projektowe" przez „bandę czworga" (Gamma, Helm, Johnson, Vlissides), pozwala na tworzenie głębokiej kopii w takich przypadkach. Pomysł polega na przeniesieniu odpowiedzialności za tworzenie obiektów do klas konkretnych, wykorzystując mechanizm funkcji wirtualnych. Klasa bazowa dostarcza metody czysto wirtualnej, która jest nadpisywana w klasach konkretnych (gdzie znany jest typ), więc można utworzyć głęboką kopię obiektu. Przykład pokazano na Listingu 4, klasa bazowa `Figure` dostarcza metody czysto wirtualnej `clone`. Metoda ta jest nadpisy-

Listing 2. *Leniwa kopia z użyciem boost::shared_ptr*

```
class LazyFoo { //przechowuje leniwą kopię obiektu typu Foo (Listing 1)
public:
   LazyFoo(int i) : ptr_(new Foo(i) ) {}
   LazyFoo(const LazyFoo& l) : ptr_(l.ptr_) {}
   int get() const { return ptr_->get(); }
   void set(int i) { //metoda zmienia stan obiektu
      if(ptr_.unique() ) { ptr_->set(i); } //bada czy istnieje konieczność
                     tworzenia kopii
      else { ptr_ = PFoo(new Foo(i) ); }
   }
private:
   typedef shared_ptr<Foo> PFoo;
   PFoo ptr_;
};
```

Listing 3. *Wykorzystanie klasy cech do wyboru algorytmu kopiowania*

```
template<typename T> //kopiowanie za pomocą memcpy
T* doFastDeepCopy(const T* element, true_type) {
   char* mem = new char[sizeof(T)]; //przydziela pamięć
   memcpy(mem, element, sizeof(T));
   return reinterpret_cast<T*>(mem);//zwraca obiekt odpowiedniego typu
}
template<typename T> //woła konstruktor kopiujący
T* doFastDeepCopy(const T* element, false_type) {
   return new T(*element);
}
template<class T> //algorytm tworzenia kopii wykorzystuje trejty
T* fastDeepCopy(const T* element) {
   return doFastDeepCopy(element, has_trivial_copy<T>() ); //tworzy dodatkowy
                     argument
}
```



Rysunek 1. *Kopia płytka i głęboka dla obiektów dostępnych pośrednio*

Szybki start

Aby uruchomić przedstawione przykłady, należy mieć dostęp do kompilatora C++ oraz edytora tekstu. Niektóre przykłady korzystają z udogodnień dostarczanych przez biblioteki boost, warunkiem ich uruchomienia jest instalacja bibliotek boost (w wersji 1.36 lub nowszej). Na wydrukach pominięto dołączanie odpowiednich nagłówków oraz udostępnianie przestrzeni nazw, pełne źródła dołączono jako materiały pomocnicze.

Listing 4. *Wzorzec prototypu*

```
class Figure {//klasa bazowa
public:
   virtual Figure* clone() const = 0;//wirtualny konstruktor
   virtual ~Figure(){}
};
class Square : public Figure {//klasa konkretna
public:
   Square(int size) : size_(size) {}
   Square(const Square& sq) : size_(sq.size_) {}
   Figure* clone() const { return new Square(*this); } //tworzy głęboką kopię
private:
   int size_;
};
class Circle : public Figure {//klasa konkretna
public:
   Circle(int r) : r_(r) {}
   Circle(const Circle& c) : r_(c.r_) {}
   Figure* clone() const { return new Circle(*this); }//tworzy głęboką kopię
private:
   int r_;
};

//przykład użycia wirtualnego konstruktora do tworzenia głębokiej kopii obiektów
typedef vector<Figure*> Figures;
Figures figures;//kolekcja figur, która będzie kopiowana
figures.push_back(new Square(2) );
figures.push_back(new Circle(3) );
figures.push_back(new Circle(1) );
Figures copy; //głęboka kopia kolekcji figur
for(Figures::const_iterator ii = figures.begin(); ii != figures.end(); ++ii)
   copy.push_back( (*ii)->clone() );//wykorzystuje wzorzec prototypu
```

Listing 5. *Fabryka prototypów*

```
class FigCloneFactory { //fabryka prototypów dla hierarchii figur
public:
   int registerFig(Figure* prototype, int id) { //rejestruje nowy obiekt oraz jego
                     identyfikator
      prototypes_.insert( make_pair(id, prototype) );
   }
   Figure* create(int id) {//tworzy obiekt danego typu i w danym stanie
      map<int, Figure*>::const_iterator i = prototypes_.find(id);
      if(i ! = prototypes_.end() ) //jeżeli znalazł odpowiedni wpis
         return prototypes_.find(id)->second->clone(); //wzorzec prototypu
      return 0L; //zwraca nullptr jeżeli nie znalazł prototypu
}
private:

   map<int, Figure*> prototypes_;//przechowuje obiekty wzorcowe
};
```

wana w klasach konkretnych, jeżeli ją będziemy wołali, to będzie tworzona głęboka kopia obiektu o odpowiednim typie.

Jeżeli jest dostępny wirtualny konstruktor, możemy tworzyć głęboką kopię, wykorzystując interfejs klasy bazowej, wołając metodę `clone()`. Listing 4 zawiera przykład, który tworzy głęboką kopię kolekcji figur i wykorzystuje przedstawioną technikę.

Fabryka prototypów

Wzorzec prototypu możemy wykorzystać w fabryce, która będzie dostarczała obiektów danego typu, nazywanej fabryką prototypów. Fabryki są to klasy pośredniczące w tworzeniu nowych obiektów, jeden z rodzajów fabryk został omówiony w SDJ 2/2010. Fabryka prototypów przechowuje obiekty wzorcowe, które będą kopiowane, jeżeli użytkownik zleci utworzenie nowego obiektu. Fabryka taka pozwala tworzyć obiektów różnych typów na podstawie identyfikatora, ponadto możemy nadać różne identyfikatory obiektom tego samego typu różniącym się stanem. Fabryki prototypów zazwyczaj zużywają więcej zasobów niż fabryki obiektów, konieczne jest przechowywanie obiektów wzorcowych, na podstawie których będą tworzone kopie. Przykład takiej fabryki pokazano na Listingu 5.

Fabryki prototypów pozwalają wygodnie tworzyć obiekty z danej hierarchii klas, wymagają, aby w tej hierarchii był implementowany wzorzec prototypu. Dodatkowym kosztem tego rodzaju fabryki jest używanie mechanizmu późnego wiązania (funkcje wirtualne), więc obiekty muszą zawierać wskaźnik na tablicę funkcji wirtualnych, wołanie metody `clone()` odbywa się pośrednio.

Podsumowanie

Przedstawione techniki związane z tworzeniem kopii są powszechnie stosowane w różnych językach programowania. Ich znajomość pozwala na tworzenie płytkiej lub głębokiej kopii w zależności od potrzeb.

W Sieci

- *http://www.boost.org* – dokumentacja bibliotek boost;
- *http://www.open-std.org* – dokumenty opisujące nowy standard C++.

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 „Średnio zaawansowane programowanie w C++", która ukaże się niebawem.

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