Język C++. Koncepcje i techniki programowania

Autor: Andrew Koenig, Barbara Moo
Tłumaczenie: Jarosław Dobrzański
ISBN: 83-7361-702-7
Tytuł oryginału: Ruminations on C++
Format: B5, stron: 362

Język C++ to najpopularniejszy obecnie język programowania. Jego podstawowe zalety – przejrzysta składnia, niewielka ilość słów kluczowych i szeroki wachlarz możliwości – przysporzyły mu wielu zwolenników. Na rynku dostępnych jest wiele książek o programowaniu w C++, jednak większość z nich zawiera sposoby rozwiązywania konkretnych problemów i zadań programistycznych. Niewiele książek koncentruje się na założeniach, na których opiera się programowanie w języku C++.

W książce „Język C++. Koncepcje i techniki programowania" autorzy skoncentrowali się na kluczowych technikach programowania w C++. Jednak nie przedstawiają ich w formie odpowiedzi na pytania „jak to zrobić", ale „dlaczego robimy to tak, a nie inaczej". Opisują szeroki wachlarz idei i technik programowania w C++ począwszy od szczegółowych przykładów kodu, a skończywszy na zasadach i filozofii projektowania.

- Tworzenie klas
- Uchwyty klas
- Zasady projektowania obiektowego
- Szablony i iteratory
- Stosowanie bibliotek
- Projektowanie bibliotek
- Techniki programowania

Dzięki tej książce nauczysz się nie tylko przestrzegać reguł języka C++, ale także myśleć w tym języku podczas pracy nad programem.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 5.

Klasy surogatów

Jak zaprojektować zasobnik C++, który potencjalnie może zawierać obiekty różnych, ale powiązanych ze sobą typów? Już samo przechowywanie obiektów w zasobniku jest trudne, ponieważ zasobniki generalnie przechowują obiekty jednego typu. Przechowywanie wskaźników obiektów, mimo że pozwala na obsługę różnych typów z pomocą dziedziczenia, wiąże się z koniecznością przydziału dodatkowej pamięci.

Przyjrzymy się tutaj jednemu ze sposobów rozwiązania tego problemu, polegającemu na zdefiniowaniu obiektów zwanych *surogatami*, które zachowują się niemal tak, jak obiekty, które reprezentują, ale pozwalają na sprowadzenie całej hierarchii typów do jednego tylko typu. Surogaty to najprostsza postać *klas uchwytów*, opisanych we wstępie do tej części książki. W kolejnych rozdziałach rozwinięte zostaną przedstawione tu ogólnie zagadnienia.

5.1. Problem

Przypuśćmy, że mamy hierarchię klas, która reprezentuje różne rodzaje środków transportu:

```
class Vehicle {
public:
   virtual double weight() const = 0;
   virtual void start() = 0;
   // ...
};
class RoadVehicle: public Vehicle { /* ... */ };
class Automobile: public RoadVehicle { /* ... */ };
class Aircraft: public Vehicle { /* ... */ };
class Helicopter: public Aircraft { /* ... */ };
```

i tak dalej. Wszystkie środki transportu (`Vehicle`) mają pewne wspólne właściwości, które opisują składowe klasy `Vehicle`. Jednak niektóre z nich mają właściwości, których nie posiadają inne. Na przykład, tylko samolot (`Aircraft`) może latać, a tylko helikopter (`Helicopter`) może wykonywać lot wiszący.

Załóżmy teraz, że chcemy śledzić zbiór różnego rodzaju środków transportu (`Vehicle`). W praktyce zastosowalibyśmy zapewne pewien rodzaj klasy zasobnika. Jednak, dla uproszczenia przykładu, zadowolimy się tablicą. Spróbujmy więc:

```
Vehicle parking_lot[1000];
```

Nie udało się. Dlaczego?

Na pozór powodem jest to, że `Vehicle` jest abstrakcyjną klasą bazową — w końcu jej funkcje składowe `weight` i `start` to *funkcje czysto abstrakcyjne*. Poprzez dopisanie `=0` w deklaracjach, funkcje te pozostały jawnie niezdefiniowane. Wynika z tego, że nie mogą istnieć obiekty klasy `Vehicle`, a jedynie obiekty klas potomnych klasie `Vehicle`. Jeżeli nie mogą istnieć obiekty `Vehicle`, niemożliwe jest również stworzenie tablicy takich obiektów.

Istnieje też głębsza przyczyna naszego niepowodzenia, którą można odkryć zadając sobie pytanie: co stałoby się, gdyby istnienie obiektów klasy `Vehicle` było mimo wszystko *możliwe*. Powiedzmy, że pozbyliśmy się wszystkich czysto wirtualnych funkcji z klasy `Vehicle` i napisaliśmy:

```
Automobile x = /* ... */;
parking_lot[num_vehicles++] = x;
```

Przyporządkowanie `x` elementowi tablicy `parking_lot` spowoduje konwersję `x` na typ `Vehicle` drogą odrzucenia wszystkich składowych, które nie stanowią części samej klasy `Vehicle`. Dopiero potem operacja przyporządkowania skopiuje ten (okrojony) obiekt do tablicy `parking_lot`.

W efekcie zdefiniowaliśmy `parking_lot` jako zbiór obiektów `Vehicle`, a nie zbiór obiektów klas potomnych klasie `Vehicle`.

5.2. Rozwiązanie klasyczne

Normalnym sposobem na osiągnięcie elastyczności w tego typu sytuacjach jest wprowadzenie pośredniości. Najprostsza, możliwa tu do zastosowania forma pośredniości sprowadza się do przechowywania wskaźników, zamiast samych obiektów:

```
Vehicle* parking_lot[1000];    // tablica wskaźników
```

Potem możemy napisać:

```
Automobile x = /* ... */;
parking_lot[num_vehicles++] = &x;
```

Tym sposobem pozbywamy się bezpośredniego problemu, ale pojawiają się dwa inne.

Po pierwsze, to co zachowaliśmy w `parking_lot`, to wskaźnik do `x`, który w tym przykładzie jest zmienną lokalną. Wobec tego, gdy tylko zmienna `x` przestanie być widoczna (dostępna), `parking_lot` przestanie cokolwiek wskazywać.

Nowo powstały problem można by rozwiązać, przechowując w parking_lot wskaźniki nie do oryginalnych obiektów, ale do ich kopii. Wówczas wypadałoby również przyjąć konwencję, że uwalniając parking_lot uwalniamy również obiekty, na które tablica ta wskazuje. Poprzedni przykład przybiera więc następującą postać:

```
Automobile x = /* ... */;
parking_lot[num_vehicles++] = new Automobile(x);
```

Co prawda w efekcie takiej zmiany tablica nie przechowuje już wskaźników do lokalnych obiektów, ale w zamian spływa na nas ciężar dynamicznego zarządzania pamięcią. Co więcej, technika ta działa tylko wtedy, gdy znany jest statyczny typ obiektów, jakie mają być umieszczone w parking_lot. A co, jeżeli nie znamy tych typów? Powiedzmy na przykład, że chcemy sprawić, by element parking_lot[p] wskazywał na nowo utworzony parking_lot[q]? Nie możemy napisać:

```
if (p != q) {
   delete parking_lot[p];
   parking_lot[p] = parking_lot[q];
}
```

ponieważ elementy parking_lot[p]i parking_lot[q] wskazywały by wówczas ten sam obiekt. Nie możemy też napisać:

```
if (p != q) {
   delete parking_lot[p];
   parking_lot[p] = new Vehicle(parking_lot[q]);
}
```

ponieważ ponownie stanęlibyśmy przed obliczem wcześniejszego problemu — obiekty typu Vehicle przecież nie mogą istnieć, a nawet gdyby mogły, to są niepożądane!

5.3. Wirtualne funkcje kopiujące

Spróbujmy znaleźć sposób na tworzenie kopii obiektów, których typ nie jest znany w chwili kompilacji. Wiadomo, że aby zrobić cokolwiek z obiektami nieokreślonego typu w C++ należy posłużyć się funkcją wirtualną. Narzucająca się nazwa dla takiej funkcji to copy lub clone.

Jako że najprowdopodobniej zależy nam na możliwości kopiowania dowolnego typu obiektów Vehicle, powinniśmy rozbudować klasę Vehicle o stosowną, czysto wirtualną funkcję:

```
class Vehicle {
public:
   virtual double weight() const = 0;
   virtual void start() = 0;
   virtual Vehicle* copy() const = 0;
   // ...
};
```

Następnie zdefiniujemy kopię funkcji składowej w każdej klasie potomnej Vehicle. Idea polega na tym, że jeżeli vp wskazuje na obiekt pewnej nieustalonej klasy, potomnej

wobec klasy `Vehicle`, to `vp->copy()` da wskaźnik do nowo utworzonej kopii obiektu. Jeżeli, na przykład, klasa `Truck` jest potomkiem (bezpośrednim lub pośrednim) klasy `Vehicle`, to jej funkcja `copy` wygląda tak:

```
Vehicle* Truck::copy() const
{
   return new Truck(*this);
}
```

Oczywiście kiedy obiekt przestaje być potrzebny, będziemy chcieli się go pozbyć. Trzeba więc dopisać do klasy `Vehicle` wirtualny destruktor:

```
class Vehicle {
public:
   virtual double weight() const = 0;
   virtual void start() = 0;
   virtual Vehicle* copy() const = 0;
   virtual ~Vehicle() { }
   // ...
};
```

5.4. Definiowanie klasy surogatu

Wiemy już, w jaki sposób można dowolnie kopiować obiekty. Zajmijmy się teraz przydziałem pamięci. Czy można w jakiś sposób uniknąć konieczności jawnego zajmowania się przydziałem pamięci, zachowując jednocześnie możliwości klasy `Vehicle` w zakresie tworzenia dynamicznych powiązań?

W C++ często okazuje się, że kluczem do rozwiązania jest *zastosowanie klas do reprezentowania pojęć*. Często korci mnie, by właśnie to spostrzeżenie uznać za fundamentalną zasadę projektowania programów w C++.

W kontekście kopiowania obiektów, zastosowanie tej reguły oznacza zdefiniowanie czegoś, co zachowuje się jak obiekt `Vehicle`, ale potencjalnie reprezentuje obiekt każdej klasy potomnej wobec klasy `Vehicle`. Obiekt takiej klasy można właśnie nazwać *surogatem*.

Każdy surogat klasy `Vehicle` będzie reprezentował obiekt jakiejś klasy, potomnej wobec klasy `Vehicle`. Obiekt ten będzie istniał tak długo, jak długo surogat pozostanie z nim skojarzony. Dlatego skopiowanie surogratu spowoduje skopiowanie odpowiadającego mu obiektu, a przypisanie surogatowi nowej wartości będzie polegać na usunięciu starego obiektu, przed skopiowaniem na jego miejsce nowego.[1] Na szczęście wyposażyliśmy już klasę `Vehicle` w wirtualną funkcję `copy`, która pozwala na tworzenie takich kopii. Wobec tego surogat można zdefiniować następująco:

[1] Jak zauważył Dag Bruck, zachowanie to subtelnie różni się od zwykle spodziewanego zachowania operatora przyporządkowania, ponieważ zmienia typ obiektu, z którym skojarzony jest surogat występujący po lewej stronie operatora.

```
class VehicleSurrogate {
public:
    VehicleSurrogate();
    VehicleSurrogate(const Vehicle&);
    ~VehicleSurrogate();
    VehicleSurrogate(const VehicleSurrogate&);
    VehicleSurrogate& operator=(const VehicleSurrogate&);
private:
    Vehicle* vp;
};
```

Surogat ma konstruktor, który pobiera argument `const Vehicle&`, co umożliwia tworzenie surogratu każdego obiektu klasy potomnej wobec klasy `Vehicle`. Klasa posiada również konstruktor domyślny, który umożliwia tworzenie tablic obiektów `VehicleSurrogate`.

Konstruktor domyślny stwarza jednak pewien problem. Skoro `Vehicle` to abstrakcyjna klasa bazowa, to jakie domyślne zachowanie powinniśmy zdefiniować dla `VehicleSurrogate`? Jaki jest typ obiektu, na który wskazuje? Nie może to być typ `Vehicle`, bo nie mogą istnieć obiekty `Vehicle`.

Dla lepszego zrozumienia, wprowadzimy pojęcie *surogatu pustego*, który zachowuje się podobnie jak wskaźnik zerowy. Możliwe będzie tworzenie, likwidacja i kopiowanie takich surogatów, ale każda inna operacja będzie traktowana jako błąd.

Jak dotąd, nie mamy jeszcze innych operacji — brak ten ułatwia uzupełnienie definicji funkcji składowych:

```
VehicleSurrogate::VehicleSurrogate(): vp(0) { }
VehicleSurrogate::VehicleSurrogate(const Vehicle& v): vp(v.copy()) { }
VehicleSurrogate::~VehicleSurrogate()
{
   delete vp;
}
VehicleSurrogate::VehicleSurrogate(const VehicleSurrogate& v): vp(v.vp? v.vp-
>copy(): 0) { }
VehicleSurrogate&
VehicleSurrogate::operator=(const VehicleSurrogate& v)
{
   if (this != &v) {
      delete vp;
      vp = (v.vp ? v.vp->copy() : 0);
   }
   return *this;
}
```

Zastosowano tu trzy triki, które warto zapamiętać.

Po pierwsze, jak widać, każde wywołanie `copy` jest wywołaniem wirtualnym. Wywołania te nie mogą być inne, ponieważ obiekty klasy `Vehicle` nie istnieją. Nawet wywołanie `v.copy` w konstruktorze, które pobiera `const Vehicle&` jest wirtualne, ponieważ `V` to wskaźnik, a nie sam obiekt.

Po drugie, w konstruktorze kopii i operatorze przyporządkowania występuje test, sprawdzający, czy `v.vp` jest niezerowe. Jest to konieczne, ponieważ w innym wypadku wywołanie `v.vp->copy` zakończyłoby się błędem.

Po trzecie, w operatorze przyporządkowania następuje sprawdzenie, czy surogat nie jest przyporządkowywany sobie samemu.

Aby zakończyć tworzenie klasy surogatu, wystarczy jeszcze uzupełnić ją o obsługę innych operacji obsługiwanych przez klasę `Vehicle`. We wcześniejszym przykładzie były to funkcje `weight` i `start`, dodajmy je więc do klasy `VehicleSurrogate`:

```
class VehicleSurrogate {
public:
   VehicleSurrogate();
   VehicleSurrogate(const Vehicle&);
   ~VehicleSurrogate();
   VehicleSurrogate(const VehicleSurrogate&);
   VehicleSurrogate& operator=(const VehicleSurrogate&);
   // operacje z klasy Vehicle
   double weight() const;
   void start();
   // ...
private:
   Vehicle* vp;
};
```

Jak widać, dodane funkcje nie są wirtualne — jedyne obiekty, jakich tutaj używamy, są klasy `VehicleSurrogate`. Nie ma tu obiektów jakiejkolwiek klasy potomnej `VehicleSurrogate`. Oczywiście same funkcje wywołują wirtualne funkcje w odpowiadającym im obiekcie `Vehicle`. Funkcje te również powinny sprawdzać, czy `vp` jest niezerowe:

```
double VehicleSurrogate::weight() const
{
   if (vp == 0)
      throw "empty VehicleSurrogate.weight()";
   return vp->weight();
}
void VehicleSurrogate::start()
{
   if (vp == 0)
      throw "empty VehicleSurrogate.start()";
   vp->start();
}
```

Po wykonaniu całej tej pracy, zdefiniowanie parkingu (`parking_lot`) jest już proste:

```
VehicleSurrogate parking_lot[1000];
Automobile x;
parking_lot[num_vehicles++] = x;
```

Ostatnia instrukcja jest tożsama:

```
parking_lot[num_vehicles++] = VehicleSurrogate(x);
```

co powoduje stworzenie kopii obiektu `X`, wiąże obiekt `VehicleSurrogate` z tą kopią, a następnie przyporządkowuje ten obiekt do elementu tablicy `parking_lot`. Z chwilą usunięcia tablicy `parking_lot` wszystkie te kopie również zostaną usunięte.

5.5. Podsumowanie

Połączenie dziedziczenia z zasobnikami zmusza do zajęcia się dwoma kwestiami: kontrolowaniem przydziału pamięci oraz umieszczaniem obiektów różnych typów w jednym zasobniku. Obydwie te kwestie można rozwiązać posługując się fundamentalną techniką w C++ polegającą na reprezentowaniu pojęć za pomocą klas. W ten sposób otrzymamy pojedynczą klasę zwaną surogatem, której każdy obiekt reprezentuje jeden inny obiekt o klasie dowolnej spośród całej hierarchii. Problem rozwiązujemy umieszczając w miejsce obiektów w naszym zasobniku reprezentujące je surogaty.