

## IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

## KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

## TWÓJ KOSZYK

DODAJ DO KOSZYKA

## CENNIK I INFORMACJE

ZAMÓW INFORMACJE  
O NOWOŚCIACH

ZAMÓW CENNIK

## CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

# 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.



# Spis treści

<b>Wstęp .....</b>	<b>11</b>
<b>Wprowadzenie .....</b>	<b>15</b>
0.1. Podejście pierwsze.....	15
0.2. Jak to zrobić bez klas?.....	18
0.3. Dlaczego w C++ było łatwiej?.....	19
0.4. Bardziej rozbudowany przykład .....	20
0.5. Wnioski .....	20
<b>Część I   Motywacja .....</b>	<b>23</b>
<b>Rozdział 1. Dlaczego używam C++? .....</b>	<b>25</b>
1.1. Problem .....	25
1.2. Historia i kontekst.....	26
1.3. Automatyczna dystrybucja oprogramowania.....	27
1.4. Czas na C++ .....	30
1.5. Oprogramowanie z odzysku .....	34
1.6. Postscriptum .....	35
<b>Rozdział 2. Dlaczego pracuję nad rozwojem C++?.....</b>	<b>37</b>
2.1. Sukces małych projektów.....	37
2.2. Abstrakcja.....	39
2.3. Maszyny powinny pracować dla ludzi.....	42
<b>Rozdział 3. Życie w prawdziwym świecie.....</b>	<b>43</b>
<b>Część II   Klasy i dziedziczenie .....</b>	<b>49</b>
<b>Rozdział 4. Lista kontrolna dla autorów klas .....</b>	<b>51</b>
<b>Rozdział 5. Klasy surogatów .....</b>	<b>61</b>
5.1. Problem .....	61
5.2. Rozwiązanie klasyczne .....	62
5.3. Wirtualne funkcje kopiujące.....	63
5.4. Definiowanie klasy surogatu .....	64
5.5. Podsumowanie.....	67
<b>Rozdział 6. Uchwyty — część 1. ....</b>	<b>69</b>
6.1. Problem .....	69
6.2. Prosta klasa.....	70
6.3. Przyłączanie uchwytu .....	72

6.4. Dostęp do obiektu.....	72
6.5. Prosta implementacja.....	73
6.6. Uchwyty z licznikami użycia.....	74
6.7. Kopiowanie przy zapisie.....	76
6.8. Omówienie .....	77
<b>Rozdział 7. Uchwyty — część 2. ....</b>	<b>79</b>
7.1. Przypomnienie.....	80
7.2. Separowanie licznika użycia.....	81
7.3. Abstrahowanie liczników użycia .....	82
7.4. Funkcje dostępowe i kopiowanie przy zapisie.....	84
7.5. Omówienie .....	85
<b>Rozdział 8. Program obiektowy .....</b>	<b>87</b>
8.1. Problem .....	87
8.2. Rozwiązanie obiektowe.....	88
8.3. Klasy uchwytów .....	91
8.4. Rozwinięcie 1. — nowe operacje .....	94
8.5. Rozwinięcie 2. — nowe typy węzłów .....	96
8.6. Refleksje.....	98
<b>Rozdział 9. Analiza ćwiczenia praktycznego — część 1. ....</b>	<b>99</b>
9.1. Problem .....	99
9.2. Projektowanie interfejsu .....	101
9.3. Kilka brakujących elementów .....	103
9.4. Testowanie interfejsu.....	104
9.5. Strategia.....	104
9.6. Taktyka.....	105
9.7. Łączenie obrazów .....	108
9.8. Wnioski .....	111
<b>Rozdział 10. Analiza ćwiczenia praktycznego — część 2. ....</b>	<b>113</b>
10.1. Strategia.....	113
10.2. Korzystanie z możliwości struktury .....	125
10.3. Wnioski .....	128
<b>Rozdział 11. Kiedy nie używać funkcji wirtualnych? .....</b>	<b>131</b>
11.1. Argumenty „za”.....	131
11.2. Argumenty „przeciw”.....	132
11.3. Szczególna rola destruktorów.....	137
11.4. Podsumowanie.....	139
<b>Część III Szablony .....</b>	<b>141</b>
<b>Rozdział 12. Tworzenie klasy zasobnika .....</b>	<b>143</b>
12.1. Co jest w środku? .....	143
12.2. Co oznacza kopiowanie zasobnika? .....	144
12.3. Jak dostać się do elementów w zasobniku? .....	147
12.4. Jak odróżnić odczyt od zapisu? .....	148
12.5. Jak poradzić sobie z rozrostem zasobnika?.....	150
12.6. Jakie operacje udostępnia zasobnik? .....	151
12.7. Jakie są założenia związane z typem elementu zasobnika? .....	152
12.8. Zasobniki i dziedziczenie .....	153
12.9. Projektowanie klasy „tablicopodobnej”.....	154

---

<b>Rozdział 13. Dostęp do elementów zasobnika .....</b>	<b>161</b>
13.1. Imitowanie wskaźnika .....	161
13.2. Dostęp do danych .....	163
13.3. Pozostałe problemy .....	165
13.4. Pointer wskazujący const Array .....	169
13.5. Użyteczne dodatki .....	170
<b>Rozdział 14. Iteratory .....</b>	<b>175</b>
14.1. Uzupełnianie klasy Pointer .....	175
14.2. Co to jest iterator? .....	178
14.3. Usuwanie elementu .....	179
14.4. Usuwanie zasobnika .....	180
14.5. Inne względy projektowe .....	181
14.6. Podsumowanie .....	182
<b>Rozdział 15. Sekwencje .....</b>	<b>183</b>
15.1. Dzieło sztuki .....	183
15.2. Stara, radykalna idea .....	185
15.3. Może jeszcze kilka dodatków .....	189
15.4. Przykład zastosowania .....	192
15.5. Może jeszcze coś .....	196
15.6. Do przemyślenia .....	198
<b>Rozdział 16. Szablony jako interfejsy .....</b>	<b>199</b>
16.1. Problem .....	199
16.2. Pierwszy przykład .....	200
16.3. Separowanie iteracji .....	200
16.4. Iterowanie poprzez dowolne typy .....	203
16.5. Dodawanie innych typów .....	204
16.6. Uogólnianie sposobu przechowywania .....	204
16.7. Dowód przydatności .....	207
16.8. Podsumowanie .....	208
<b>Rozdział 17. Szablony a algorytmy ogólne .....</b>	<b>211</b>
17.1. Konkretny przykład .....	212
17.2. Uogólnianie typu elementu .....	213
17.3. Przełożenie zliczania elementów na później .....	214
17.4. Niezależność od adresu .....	215
17.5. Wyszukiwanie w strukturach niebędących tablicami .....	217
17.6. Podsumowanie .....	218
<b>Rozdział 18. Iteratory ogólne .....</b>	<b>221</b>
18.1. Inny algorytm .....	221
18.2. Kategorie i wymogi .....	223
18.3. Iteratory wejściowe .....	224
18.4. Iteratory wyjściowe .....	224
18.5. Iteratory postępowe .....	225
18.6. Iteratory dwukierunkowe .....	226
18.7. Iteratory o dostępie swobodnym .....	226
18.8. Dziedziczenie? .....	227
18.9. Wydajność .....	228
18.10. Podsumowanie .....	228
<b>Rozdział 19. Korzystanie z iteratorów ogólnych .....</b>	<b>231</b>
19.1. Typy iteratorów .....	232
19.2. Wirtualne sekwencje .....	232

19.3. Iterator strumienia wyjściowego.....	234
19.4. Iterator strumienia wejściowego.....	236
19.5. Omówienie .....	239
<b>Rozdział 20. Adaptery dla iteratorów.....</b>	<b>241</b>
20.1. Przykład.....	241
20.2. Asymetria kierunkowa.....	243
20.3. Konsekwencja a asymetria .....	244
20.4. Automatyczne odwracanie .....	245
20.5. Omówienie .....	247
<b>Rozdział 21. Obiekty funkcji .....</b>	<b>249</b>
21.1. Przykład.....	249
21.2. Wskaźniki funkcji.....	252
21.3. Obiekty funkcji.....	254
21.4. Szablony obiektów funkcji .....	255
21.5. Ukrywanie typów pośrednich.....	256
21.6. Jeden typ zawiera kilka.....	257
21.7. Implementacja .....	258
21.8. Omówienie .....	260
<b>Rozdział 22. Adaptery funkcji .....</b>	<b>261</b>
22.1. Dlaczego obiekty funkcji?.....	261
22.2. Obiekty funkcji dla operatorów wbudowanych .....	262
22.3. Funkcje wiążące .....	263
22.4. Spojrzenie z bliska.....	264
22.5. Dziedziczenie interfejsu .....	265
22.6. Używanie klas .....	266
22.7. Omówienie .....	267
<b>Część IV Biblioteki .....</b>	<b>269</b>
<b>Rozdział 23. Biblioteki w bieżącym zastosowaniu .....</b>	<b>271</b>
23.1. Problem .....	271
23.2. Istota problemu — część 1. ....	273
23.3. Implementacja — część 1. ....	273
23.4. Istota problemu — część 2. ....	276
23.5. Implementacja — część 2. ....	276
23.6. Omówienie .....	278
<b>Rozdział 24. Obiektowa lekcja projektowania interfejsu biblioteki.....</b>	<b>281</b>
24.1. Komplikacje .....	282
24.2. Poprawianie interfejsu .....	283
24.3. Szczegółowe rozważania.....	285
24.4. Pisanie kodu .....	286
24.5. Wnioski .....	288
<b>Rozdział 25. Projektowanie bibliotek jako budowanie języka.....</b>	<b>289</b>
25.1. Ciągi znakowe .....	289
25.2. Wyczerpanie dostępnej pamięci .....	290
25.3. Kopiowanie .....	293
25.4. Ukrywanie implementacji.....	296
25.5. Konstruktor domyślny .....	298
25.6. Inne operacje .....	299
25.7. Podciągi.....	301
25.8. Wnioski .....	302

<b>Rozdział 26. Projektowanie języka jako budowanie bibliotek.....</b>	<b>303</b>
26.1. Abstrakcyjne typy danych .....	303
26.2. Biblioteki i abstrakcyjne typy danych.....	305
26.3. Rezerwacja pamięci.....	308
26.4. Przyporządkowywanie i inicjalizacja składowych klas .....	309
26.5. Obsługa wyjątków.....	311
26.6. Podsumowanie.....	312
<b>Część V Techniki.....</b>	<b>313</b>
<b>Rozdział 27. Klasy, które się śledzą .....</b>	<b>315</b>
27.1. Projektowanie klasy śledzącej .....	315
27.2. Tworzenie martwego kodu .....	318
27.3. Generowanie wyników inspekcji dla obiektów .....	319
27.4. Weryfikowanie zachowania zasobnika.....	321
27.5. Podsumowanie.....	325
<b>Rozdział 28. Grupowy przydział pamięci dla obiektów .....</b>	<b>327</b>
28.1. Problem .....	327
28.2. Projektowanie rozwiązania.....	327
28.3. Implementacja .....	330
28.4. Dziedziczenie .....	332
28.5. Podsumowanie.....	333
<b>Rozdział 29. Aplikatory, manipulatory i obiekty funkcji .....</b>	<b>335</b>
29.1. Problem .....	336
29.2. Rozwiązanie .....	338
29.3. Inne rozwiązanie.....	338
29.4. Dodatkowe argumenty.....	340
29.5. Przykład.....	341
29.6. Formy skrócone.....	343
29.7. Przemyslenia .....	344
29.8. Uwagi historyczne, źródła i podziękowania .....	345
<b>Rozdział 30. Uniezależnianie bibliotek aplikacji od wejść i wyjść.....</b>	<b>347</b>
30.1. Problem .....	347
30.2. Rozwiązanie 1. — spryt i metoda siłowa.....	348
30.3. Rozwiązanie 2. — abstrakcyjne wyjście .....	349
30.4. Rozwiązanie 3. — spryt bez metody siłowej.....	351
30.5. Uwagi .....	354
<b>Część VI Podsumowanie .....</b>	<b>355</b>
<b>Rozdział 31. Przez złożoność do prostoty.....</b>	<b>357</b>
31.1. Świat jest złożony.....	357
31.2. Złożoność staje się ukryta.....	358
31.3. Komputery mają to samo.....	359
31.4. Komputery rozwiązują prawdziwe problemy .....	361
31.5. Biblioteki klas i semantyka języka .....	362
31.6. Ułatwianie jest trudne.....	364
31.7. Abstrakcja a interfejs.....	365
31.8. Konserwacja złożoności .....	366

<b>Rozdział 32. „Witaj świecie” i co dalej?.....</b>	<b>367</b>
32.1. Znajdź w pobliżu eksperta.....	367
32.2. Wybierz narzędzie i opanuj je .....	368
32.3. Niektóre elementy C są ważne.....	368
32.4. ...inne niekoniecznie.....	370
32.5. Wyznacz sobie szereg problemów do rozwiązania.....	371
32.6. Konkluzja .....	374
<b>Dodatki .....</b>	<b>377</b>
<b>Skorowidz.....</b>	<b>379</b>

## 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 służy 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 najprawdopodobniej 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 surogatu spowoduje skopiowanie odpowiadającego mu obiektu, a przypisanie surogatowi nowej wartości będzie polegać na usunięciu starego obiektu, przed skopiowaniem na jego miejsce nowego.<sup>1</sup> Na szczęście wyposażyliśmy już klasę `Vehicle` w wirtualną funkcję `copy`, która pozwala na tworzenie takich kopii. Wobec tego surogat można zdefiniować następująco:

<sup>1</sup> 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 surogatu 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 jakiegokolwiek 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.