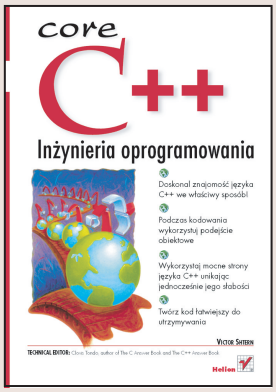
C++. Inżynieria programowania



Autor: Victor Shtern
Tłumaczenie: Daniel Kaczmarek (rozdz. 1 – 6), Adam Majczak (rozdz. 7 – 11), Rafał Szpoton (rozdz. 12 – 19)
ISBN: 83-7361-171-1
Tytuł oryginału: Core C++: A Software Engineering Approach
Format: B5, stron: 1084

Naucz się języka C++ w najlepszy sposób: poznając go z punktu widzenia inżynierii programowania

- Demonstruje praktyczne techniki stosowane przez zawodowych programistów
- Zawiera poprawny, gruntownie przetestowany przykładowy kod źródłowy programów oraz przykłady zaczerpnięte z praktyki
- Skoncentrowana na nowoczesnych technologiach, które muszą poznać programiści
- Zawiera rady profesjonalistów, które pozwolą czytelnikowi tworzyć najlepsze programy

Książka *Wiktora Shterna* zatytułowana „C++. Inżynieria programowania" stosuje wyjątkowy sposób nauki języka C++ przeznaczony dla programistów mających doświadczenie w dowolnym języku programowania: prezentuje możliwość zastosowania w C++ najlepszych technik programistycznych oraz metodologii inżynierii programowania. Nawet jeżeli już wcześniej wykorzystywałeś język C++, ta wyczerpująca książka przedstawi sposób tworzenia poprawniejszego kodu, łatwiejszego do utrzymania i modyfikacji. Książka niniejsza uczy zasad programowania obiektowego przed samą nauką języka, co pozwala wykorzystać wszystkie zalety OOP do tworzenia poprawnych aplikacji. Udoskonalisz znajomość kluczowych składników standardu ANSI/ISO C++ rozpatrywanych z punktu widzenia inżyniera: klas, metod, modyfikatorów const, dynamicznego zarządzania pamięcią, złożeń klas, dziedziczenia, polimorfizmu, operacji wejścia-wyjścia i wielu innych. Jeżeli pragniesz tworzyć w języku C++ najlepsze programy, musisz projektować, myśleć i programować stosując najlepsze obecnie praktyki inżynierii programowania. Lektura książki „C++. Inżynieria programowania" pomoże Ci w tym.

Książka „C++. Inżynieria programowania" kładzie nacisk na:

- Prezentację zastosowań zasad inżynierii programowania w programach pisanych w C++
- Tworzenie kodu łatwego do późniejszych modyfikacji
- Praktyczne zrozumienie zasad programowania obiektowego przed nauką samego języka
- Przedstawienie najnowszych cech standardu ANSI/ISO C++
- Zaprezentowanie setek realistycznych przykładów kodu programów

Wiktor Shtern jest profesorem wykładającym w college'u przy uniwersytecie w Bostonie, uznawanym za jedną z najlepszych w Stanach Zjednoczonych szkół dla osób pracujących zawodowo. Oprócz wykładów z języka C++ na poziomie uniwersyteckim, Shtern prowadzi również zajęcia praktyczne dla doświadczonych programistów.

Dan Costello jest inżynierem oprogramowania w firmie GE Marquette Medical Systems

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

Spis treści

11

Konstruktory i destruktory — potencjalne problemy

Zagadnienia omówione w tym rozdziale:

- Więcej o przekazywaniu obiektów poprzez wartość.
- Przeciążanie operatorów w klasach nie będących klasami numerycznymi.
- Więcej o konstruowaniu kopii obiektów.
- Przeciążenie operatora przypisania.
- Rozważania praktyczne — jak chcielibyśmy to zaimplementować?
- Podsumowanie.

Funkcje operatorowe dokonujące przeciążenia operatorów przydają nowego impulsu programowaniu obiektowemu. Okazało się, że oto zamiast koncentrować się na łączeniu w jedną, logicznie powiązaną całość danych i operacji w oparciu o pewne wspólne koncepcje, zajmujemy się rozważaniami w kategoriach estetycznych i zagadnieniami równego traktowania wbudowanych, elementarnych typów danych oraz typów definiowanych przez programistę w programach pisanych w C++.

Ten rozdział stanowi bezpośrednią kontynuację poprzedniego. W rozdziale 10. „Funkcje operatorowe — jeszcze jeden dobry pomysł" omawiałem zagadnienia odnoszące się do projektowania klas typu numerycznego na przykładzie klas `Complex` (liczba zespolona) oraz `Rational` (ułamek zwykły). Obiekty stanowiące zmienne (instancje) tych klas są pełnoprawnymi obiektami, rzec można z prawdziwego zdarzenia. Stosują się do nich wszystkie zagadnienia odnoszące się do klas i obiektów — deklaracja klasy, kontrola dostępu do składników klasy, projektowanie metod, definiowanie obiektów i przesyłanie komunikatów do obiektów.

Typy danych definiowanych przez programistę (klasy) omawiane poprzednio są danymi numerycznymi. Nawet jeśli mają wewnętrzną strukturę bardziej skomplikowaną niż elementarne typy liczb całkowitych czy liczb zmiennoprzecinkowych, obiekty takich klas mogą być w kodzie

klienta obsługiwane w sposób podobny do liczb całkowitych i liczb zmiennoprzecinkowych. W kodzie klienta obiekty takich typów mogą być dodawane, mnożone, porównywane itp.

Popatrzmy uważnie na sposób wyartykułowania ostatniej myśli (w ostatnich dwóch zdaniach). Określenie „obiekty takich typów mogą…" odnosi się oczywiście do typów zdefiniowanych przez programistę. Co natomiast oznacza „podobny do liczb całkowitych czy liczb zmiennoprzecinkowych…"? Skoro te dane porównujemy z typami zdefiniowanymi przez programistę, mamy tu na myśli „typy wartości całkowitych i zmiennoprzecinkowych", a nie same zmienne tychże typów. Lepiej zapewne byłoby tu użyć sformułowania „wbudowane typy danych". Co oznacza „obiekty takich klas mogą być w kodzie klienta obsługiwane…"? Tu chyba także chodzi o typy zdefiniowane przez programistę? Niezupełnie, ponieważ w tym zdaniu mówi się o obsłudze po stronie kodu klienta. Kod klienta nie obsługuje typów zdefiniowanych przez programistę, lecz obiekty typu zdefiniowanego przez programistę. To zmienne określonego typu (instancje, obiekty) są dodawane, mnożone, porównywane, itp. Autor podkreśla tu wyraźnie ten szczegół, ponieważ wiesz już o klasach i obiektach wystarczająco dużo, by wyczuwać brak precyzji w niektórych sformułowaniach w dyskusji o programowaniu obiektowym i by samemu unikać, w miarę możliwości, takich nieprecyzyjnych sformułowań.

Innymi słowy, obiekty klas (typów) numerycznych, zdefiniowanych przez programistę, mogą po stronie kodu klienta być obsługiwane podobnie do zmiennych elementarnych typów wbudowanych. Z tego powodu obsługa przeciążania operatorów wobec takich klas zdecydowanie ma sens. Zasada C++, by umożliwiać traktowanie zmiennych typów elementarnych i obiektów typów zdefiniowanych przez programistę w jednakowy sposób — działa prawidłowo wobec tych klas. W tym rozdziale autor zamierza omówić przeciążanie operatorów wobec takich klas, których obiekty nie mogą być dodawane, mnożone, odejmowane ani dzielone. Na przykład do obsługi łańcuchów znaków w pamięci można utworzyć klasę `String`. Z racji nienumerycznego charakteru takich klas (z samej ich natury) funkcje operatorowe dokonujące przeciążenia operatorów wobec tych klas wyglądają sztucznie i nienaturalnie. Na przykład możemy dokonać przeciążenia operatora dodawania (`+`) w taki sposób, by umożliwiał konkatenację dwóch obiektów klasy `String` (dodanie drugiego łańcucha znaków do końca pierwszego) lub np. przeciążenia operatora numerycznej równości (`==`) w taki sposób, by umożliwiał porównanie dwóch łańcuchów znaków reprezentowanych przez obiekty klasy `String`. I to wygląda dość rozsądnie. Z drugiej jednak strony, trudno wymyślić jakąkolwiek rozsądną interpretację dla operatorów mnożenia czy dzielenia wobec obiektów klasy `String`. Niemniej jednak funkcje operatorowe dokonujące przeciążania operatorów C++[1] dla klas nienumerycznych są popularne i powinno się wiedzieć, jak z nimi postępować.

Ważną różnicą pomiędzy klasami numerycznymi a nienumerycznymi jest to, że w klasach nienumerycznych mogą występować znaczące różnice w objętości danych, którymi mogą posługiwać się obiekty tej samej klasy. Obiekty klas numerycznych zawsze zajmują taką samą ilość miejsca w pamięci. Na przykład obiekty klasy `Rational` zawierają zawsze dwa pola danych — licznik i mianownik.

Natomiast w przypadku klasy `String` objętość tekstu, który może być przechowywany w pojedynczym obiekcie tej klasy, może być inna niż objętość tekstu przechowywana w innym obiekcie tej samej klasy. Jeśli klasa rezerwuje dla każdego obiektu taką samą (zbyt dużą)

[1] Przypomnijmy: pierwotnie, z natury arytmetycznych — *przyp. tłum.*

ilość pamięci, mamy do czynienia w programie z dwoma niekorzystnymi, a skrajnymi zjawiskami — marnowaniem pamięci (gdy rzeczywista wielkość tekstu jest mniejsza niż ilość zarezerwowanej pamięci) albo z przepełnieniem pamięci (gdy rzeczywista wielkość tekstu okaże się większa niż ilość zarezerwowanej pamięci). Te dwa niebezpieczeństwa czyhają stale i w którąś z tych dwu pułapek zawsze, prędzej czy później, wpadną ci projektanci klas, którzy rezerwują tę samą ilość pamięci dla każdego takiego obiektu.

C++ rozwiązuje ten problem poprzez przyporządkowanie dla każdego obiektu tej samej ilości pamięci (na stercie albo na stosie) zgodnie ze specyfikacją klasy, a następnie ewentualne dodanie, w miarę potrzeby, dodatkowej pamięci na stercie[2]. Taka dodatkowa ilość pamięci na stercie zmienia się i może być zupełnie inna dla poszczególnych obiektów. Nawet dla jednego, konkretnego obiektu podczas jego życia ta ilość dodatkowej pamięci może się zmieniać. Na przykład jeśli do obiektu klasy `String` zawierającego jakiś tekst dodawany jest następny łańcuch znaków (konkatenacja), taki obiekt może powiększyć swoją pamięć, by pomieścić nowy, dłuższy tekst.

Dynamiczne zarządzanie pamięcią na stercie obejmuje zastosowanie konstruktorów i destruktorów. Ich nieumiejętne, niezręczne stosowanie może negatywnie wpłynąć na efektywność działania programu. Co gorsza, może to spowodować utratę danych przechowywanych w pamięci i utratę integralności programu, co jest zjawiskiem charakterystycznym dla C++, nieznanym w innych językach programowania. Każdy programista pracujący w C++ powinien być świadom tych zagrożeń. To z tego powodu zagadnienia te zostały włączone do tytułu niniejszego rozdziału, mimo że ten rozdział stanowi w istocie kontynuację rozważań o funkcjach operatorowych dokonujących przeciążenia operatorów.

Dla uproszczenia dyskusji niezbędne podstawowe koncepcje zostaną tu wprowadzone w oparciu o klasę `Rational` (o stałej wielkości obiektów), znaną z rozdziału 10. W tym rozdziale te koncepcje zostaną zastosowane w odniesieniu do klasy `String` z dynamicznym zarządzaniem pamięci na stercie. W rezultacie twoja intuicja programistyczna wzbogaci się o wyczucie relacji pomiędzy instancjami obiektów po stronie kodu klienta. Jak się przekonasz, relacje pomiędzy obiektami okażą się inne niż relacje pomiędzy zmiennymi typów elementarnych, pomimo wysiłków, by takie zmienne i obiekty były traktowane tak samo. Innymi słowy, powinieneś przygotować się na duże niespodzianki.

Lepiej nie pomijać materiału zawartego w niniejszym rozdziale. Zagrożenia związane z zastosowaniem i rolą konstruktorów i destruktorów w C++ są to niebezpieczeństwa zdecydowanie realne i należy wiedzieć, jak się przed nimi bronić (broniąc przy okazji swojego szefa i swoich użytkowników).

Więcej o przekazywaniu obiektów poprzez wartość

Wcześniej, w rozdziale 7. („Programowanie w C++ z zastosowaniem funkcji”) przytaczano argumenty przeciwko przekazywaniu obiektów do funkcji jako parametrów poprzez wartość albo poprzez wskaźnik do obiektu, natomiast zachęcano, by zamiast tego przekazywać obiekty jako parametry — poprzez referencje.

[2] W sposób dynamiczny — *przyp. tłum.*

Autor wyjaśniał tam, że przekazanie parametru poprzez referencję jest prawie równie proste, jak przekazanie go poprzez wartość, jest jednakże szybsze — w przypadku tych parametrów wejściowych, które nie zostaną zmodyfikowane przez daną funkcję. Przekazywanie poprzez referencję jest równie szybkie, jak przekazywanie poprzez wskaźnik, ale składnia odnosząca się do parametrów przekazywanych poprzez referencję jest znacznie prostsza — dla parametrów wyjściowych, które zostają zmodyfikowane przez daną funkcję w trakcie jej wykonania.

Zaznaczono tam również, że składnia przy przekazywaniu parametrów poprzez referencje jest dokładnie taka sama — i dla parametrów wejściowych, i dla parametrów wyjściowych funkcji. Z tego też powodu autor sugerował stosowanie słowa kluczowego (modyfikatora) `const` dla parametrów wejściowych w celu wskazania, że te parametry nie zostaną zmodyfikowane w wyniku wykonania danej funkcji. Jeśli nie stosujemy żadnych takich sugestii, powinno to w sposób nie wzbudzający wątpliwości wskazywać, że te parametry będą zmienione w trakcie wykonania danej funkcji.

Oprócz tego autor przytaczał argumenty przeciwko zwracaniu obiektów z funkcji poprzez wartość, jeśli tylko nie jest to niezbędne w celu przesłania innych komunikatów do takiego zwróconego obiektu (składnia łańcuchowa w wyrażeniach).

Przy takim podejściu przekazywanie parametrów poprzez wartość powinno być ograniczone do przekazywania parametrów wejściowych funkcji, jeśli są one typów elementarnych, i zwrotu z funkcji wartości tychże typów elementarnych. Dlaczego parametry wejściowe typów wbudowanych są akceptowalne? Przekazywanie ich poprzez wskaźnik zwiększy stopień złożoności i może wprowadzić w błąd osobę czytającą kod, sugerując, że taki parametr może zostać zmodyfikowany podczas wykonania funkcji. Przekazanie ich poprzez referencje (z modyfikatorem `const`) nie jest trudne, ale zwiększa nieco stopień złożoności kodu. Skoro takie parametry są niewielkie, przekazywanie ich poprzez referencje nie zwiększy w najmniejszym stopniu szybkości działania programu. Z wymienionych powodów najprostszy sposób przekazywania parametrów (poprzez wartość) jest stosowny dla typów wbudowanych.

W trakcie ostatniego rozdziału poznaliśmy wystarczająco techniki programowania, umożliwiające nam nie tylko omówienie wad i zalet różnych trybów przekazywania parametrów do funkcji i z funkcji, lecz także umożliwiających nam prześledzenie kolejności wywołań funkcji.

Oprócz tego przy różnych okazjach autor podkreślał, że inicjowanie i przypisanie, nawet jeśli w obu przypadkach następuje użycie tego samego symbolu = , są różnie traktowane . W tym podrozdziale przy użyciu wydruków komunikatów diagnostycznych zostaną zademonstrowane te różnice.

Oba zagadnienia zostaną zademonstrowane za pomocą przykładowego kodu z listingu 11.1, zawierającego uproszczoną (i zmodyfikowaną) wersję klasy `Rational` z poprzedniego rozdziału oraz funkcję testującą — `test driver`.

Listing 11.1. *Przykład przekazania obiektów jako parametrów funkcji poprzez wartość*

```
#include <iostream.h>
class Rational
{
long nmr, dnm;                          // prywatne dane
void normalize();                       // prywatne metody
```

```
public:
Rational(long n=0, long d=1)               // konstruktor ogólny, konwersji i domyślny
{ nmr = n; dnm = d;
this->normalize();
cout << " obiekt utworzony (konstruktor): " << nmr << " " << dnm << endl; }
Rational(const Rational &r)                // konstruktor kopiujący
{ nmr = r.nmr; dnm = r.dnm;
cout << " obiekt skopiowany (konstruktor kopiujący): " << nmr << " " << dnm << endl;
}
void operator = (const Rational &r)        // operator przypisania
{ nmr = r.nmr; dnm = r.dnm;
cout << " obiekt przypisany (operator=()): " << nmr << " " << dnm << endl; }
~Rational()                                // destruktor
{ cout << " obiekt usunięty (destruktor): " << nmr << " " << dnm << endl; }
friend Rational operator + (const Rational x, const Rational y);
void show() const;
} ;                                        // koniec specyfikacji klasy
void Rational::show() const
{ cout << " " << nmr << "/" << dnm; }
void Rational::normalize()                 // prywatna metoda
{ if (nmr == 0) { dnm = 1; return; }
int sign = 1;
if (nmr < 0) { sign = -1; nmr = -nmr; }    // zmień znak
if (dnm < 0) { sign = -sign; dnm = -dnm; } // by obydwa były dodatnie
long gcd = nmr, value = dnm;               // największy wspólny podzielnik
while (value != gcd) {                     // zatrzymaj po znalezieniu n.w.p.
if (gcd > value)
gcd = gcd - value;                         // odejmuj liczbę mniejszą od większej
else value = value - gcd; }
nmr = sign * (nmr/gcd); dnm = dnm/gcd; }   // mianownik dodatni
Rational operator + (const Rational x, const Rational y)
{ return Rational(y.nmr*x.dnm + x.nmr*y.dnm, y.dnm*x.dnm); }

int main()
{ Rational a(1,4), b(3,2), c;
cout << endl;
c = a + b;
a.show(); cout << " +"; b.show(); cout << " ="; c.show();
cout << endl << endl;
return 0;
}
```

Spośród wszystkich funkcji odnoszących się uprzednio do klasy `Rational` pozostawiono tu jedynie funkcje `normalize()`, `show()` oraz `operator+()`. Poza tym zwróć uwagę, że funkcja dokonująca przeciążenia operatora dodawania, `operator+()`, nie jest w tym przypadku metodą, lecz funkcją zaprzyjaźnioną kategorii `friend`. Z tego powodu na początku tego podrozdziału (i dalej) w odniesieniu do klasy `Rational` użyto stwierdzenia: „wszystkich funkcji odnoszących się uprzednio do klasy", a nie „wszystkich metod należących do klasy". Autor postąpił tak, choć chciałby jeszcze raz zaakcentować, że zaprzyjaźnione funkcje kategorii `friend` z punktu widzenia wszelkich zamiarów i zastosowań są jak metody należące do klasy. Funkcja zaprzyjaźniona jest implementowana w tym samym pliku, co wszystkie metody, ma takie same prawa dostępu do prywatnych składników klasy, jak wszystkie metody, jest całkowicie bezużyteczna wobec obiektów klasy innej niż jej klasa „zaprzyjaźniona", w tym przypadku klasa `Rational` — jak wszystkie inne metody. Różnica polega jedynie na składni stosowanej przy wywołaniu tej funkcji. Tylko tu funkcja zaprzyjaźniona różni się od metod,

ale w przypadku funkcji operatorowych, dokonujących przeciążania operatorów, składnia operatorowa jest dokładnie taka sama dla funkcji zaprzyjaźnionych i dla metod należących do klasy.

W obrębie ogólnego (dwuargumentowego) konstruktora klasy `Rational` dodano wydruk komunikatu o charakterze diagnostycznym. Ten wydruk komunikatu powinien następować za każdym razem, gdy obiekt klasy `Rational` jest tworzony i inicjowany przez ten konstruktor — na początku kodu funkcji `main()` i we wnętrzu kodu funkcji operatorowej `operator+()`.

```
Rational::Rational(long n=0, long d=1)      // wartości domyślne
{
nmr = n; dnm = d;                           // inicjowanie danych
this->normalize();
cout << " obiekt utworzony (konstruktor): " << nmr << " " << dnm << endl;
}
```

Dodano tu także konstruktor kopiujący z wydrukiem komunikatu diagnostycznego. Ten komunikat diagnostyczny zostanie wyprowadzony na ekran zawsze, gdy obiekt klasy `Rational` będzie inicjowany za pomocą kopiowania pól danych innego obiektu klasy `Rational`. Takie kopiowanie obiektu na przykład nastąpi wtedy, gdy parametry będą przekazywane poprzez wartość do funkcji `operator+()` lub gdy będzie następował zwrot wartości z funkcji (zwrot obiektu poprzez wartość).

```
Rational::Rational(const Rational &r)       // konstruktor kopiujący
{
nmr = r.nmr; dnm = r.dnm;                   // kopiuj pola danych
cout << " obiekt skopiowany (konstruktor kopiujący): " << nmr << " " << dnm << endl;
}
```

Ten konstruktor kopiujący jest wywoływany, gdy argumenty klasy (typu) `Rational` są poprzez wartość przekazywane do zaprzyjaźnionej funkcji operatorowej `friend operator+()`. Wbrew mylnemu pierwszemu wrażeniu, ten konstruktor kopiujący nie zostanie wywołany, gdy funkcja `operator+()` będzie zwracać obiekt poprzez wartość, ponieważ przed powrotem z tej funkcji operatorowej nastąpi wywołanie dwuargumentowego konstruktora ogólnego.

W klasie `Rational` destruktor nie ma do wykonania odpowiedzialnego zadania i został dodany do specyfikacji tej klasy tylko w celu ułatwienia wyprowadzania komunikatów diagnostycznych; destruktor zgłosi się poprzez wydruk komunikatu diagnostycznego zawsze, gdy obiekt klasy `Rational` będzie usuwany.

Najciekawszą funkcją jest tu funkcja dokonująca przeciążenia operatora przypisania (=). Jej praca polega na kopiowaniu pól danych jednego obiektu klasy `Rational` na pola danych innego obiektu klasy `Rational`. Czym jej działanie różni się od działania konstruktora kopiującego? W tym stadium możemy stwierdzić, że niczym, choć inny jest typ wartości zwracanej. Konstruktor kopiujący, jak to konstruktor, nie może mieć żadnego typu wartości zwracanej, natomiast metoda operatorowa, jak większość metod, musi mieć jakiś typ wartości zwracanej. Dla uproszczenia przyjęto tu ten typ wartości zwracanej jako `void`.

```
void Rational::operator = (const Rational &r)     // operacja przypisania
{
nmr = r.nmr; dnm = r.dnm;                         // kopiowanie danych
cout << " obiekt przypisany (operator=()): " << nmr << " " << dnm << endl;
}
```

Poddany tu przeciążeniu operator przypisania (`=`) jest operatorem dwuargumentowym (binarnym). Skąd to wiadomo? Po pierwsze, funkcja operatorowa ma jako jeden parametr obiekt klasy `Rational`, a nie jest to funkcja zaprzyjaźniona, lecz metoda. Skoro tak, jak każda metoda z jednym parametrem obiektowym, działa na dwóch argumentach. Jednym, niejawnym argumentem jest obiekt docelowy komunikatu (widziany przez metodę jako własny obiekt), drugim, jawnym argumentem jest obiekt — parametr. Drugim wyjaśnieniem jest składnia stosowania operatora przypisania (po stronie kodu klienta). Operator dwuargumentowy jest zawsze wstawiany pomiędzy pierwszy a drugi operand. Gdy dodaje się dwa operandy, kolejność jest następująca: pierwszy operand, operator, drugi operand (np. `a + b`). Podobnie jest, gdy dokonuje się przypisania. I tu kolejność jest następująca: pierwszy operand, operator, drugi operand (np. `a = b`). Odnosząc to teraz do składni wywołania funkcji, zauważamy, że obiekt `a` jest docelowym obiektem komunikatu. W podanej funkcji operatorowej, dokonującej przeciążenia operatora przypisania, pola `nmr` (licznik ułamka) oraz `dnm` (mianownik ułamka) należą do docelowego obiektu komunikatu, `a`. Obiekt `b` jest bieżącym argumentem danego wywołania funkcji operatorowej. W podanej funkcji operatorowej, dokonującej przeciążenia operatora przypisania, pola `r.nmr` oraz `r.dnm` należą do bieżącego argumentu metody, `b`. Skoro tak, to operator (`a = b`) przypisania odpowiada wywołaniu funkcji operatorowej o następującej składni: `a.operator=(b);`.

Ponieważ ta metoda ma typ wartości zwracanej `void` (nie zwraca żadnej wartości), taka funkcja operatorowa nie umożliwia obsługi wyrażeń łańcuchowych po stronie kodu klienta (np. `a = b = c`). Takie wyrażenie jest przez kompilator interpretowane jako `a = (b = c)`. Oznacza to, że wartość zwracana przez operator przypisania (`b = c`), to znaczy poprzez wywołanie funkcji operatorowej `b.operator=(c);`, musiałaby zostać wykorzystana jako argument kolejnego wywołania tej samej funkcji operatorowej: `a.operator=(b.operator=(c));`. Aby takie wyrażenie było poprawne i zostało poprawnie zinterpretowane, operator przypisania powinien zwracać jako wartość obiekt (tu klasy `Rational`). Skoro funkcja została zaprojektowana tak, że jej typ wartości zwracanej jest `void`, wyrażenie łańcuchowe po stronie kodu klienta zostanie przez kompilator zasygnalizowane jako błąd składniowy. Przy pierwszym spojrzeniu na przeciążanie operatora przypisania to nie jest istotne. Wyrażenia łańcuchowe będą stosowane później, w dalszej części niniejszego rozdziału.

Wydruk wyjściowy programu z listingu 11.1 został pokazany na rysunku 11.1. Pierwsze trzy wydrukowane komunikaty „utworzony” pochodzą od konstruktorów w rezultacie utworzenia i zainicjowania trzech obiektów klasy `Rational` w obrębie funkcji `main()`. Dwa komunikaty o kopiowaniu, „skopiowany”, pochodzą z wnętrza funkcji przeciążającej operator dodawania `operator+()`. Kolejny komunikat o utworzeniu obiektu, `utworzony`, wynika z wywołania konstruktora klasy `Rational` wewnątrz ciała funkcji `operator+()`.

Wszystkie te wywołania konstruktorów mają miejsce na początku wykonania funkcji. Później następuje seria zdarzeń, gdy wykonanie kodu funkcji dochodzi do końca (tj. do nawiasu klamrowego zamykającego kod ciała funkcji), a lokalne i tymczasowe obiekty są usuwane. Pierwsze dwa komunikaty destruktora „usunięty”) następują wtedy, gdy dwie lokalne kopie bieżących argumentów (obiekty — ułamki $^3/_2$ oraz $^1/_4$) są usuwane i dla tych dwóch obiektów następuje wywołanie destruktorów. Ten obiekt, który zawiera sumę dwóch argumentów, nie może zostać usunięty, zanim nie zostanie użyty w operacji (tzn. przez operator) przypisania. Kolejny komunikat, „przypisany”, pochodzi z wywołania funkcji dokonującej przeciążenia operatora przypisania — `operator=()`. Wreszcie komunikat `usunięty` pochodzi od destruktora tego obiektu, który został utworzony w obrębie ciała funkcji `operator+()`. Ostatnie trzy komunikaty `usunięty` pochodzą od tych destruktorów, które są wywoływane, gdy wykonanie

Rysunek 11.1.
Wydruk wyjściowy programu z listingu 11.1

```
obiekt utworzony (konstruktor): 1 4
obiekt utworzony (konstruktor): 3 2
obiekt utworzony (konstruktor): 0 1

obiekt skopiowany (konstruktor kopiujący): 3 2
obiekt skopiowany (konstruktor kopiujący): 1 4
obiekt utworzony (konstruktor):  7 4
obiekt usunięty (destruktor):    1 4
obiekt usunięty (destruktor):    3 2
obiekt przypisany (operator=()): 7 4
obiekt usunięty (destruktor):    7 4
1/4 + 3/2 = 7/4

obiekt usunięty (destruktor):    7 4
obiekt usunięty (destruktor):    3 2
obiekt usunięty (destruktor):    1 4
```

programu dochodzi do nawiasu klamrowego kończącego kod funkcji `main()` i usuwane są obiekty `a`, `b`, `c`. Skoro konstruktor kopiujący nie jest wywoływany, komunikat `skopiowany` nie pojawia się na wydruku wyjściowym.

Ta sekwencja zdarzeń rozegra się zupełnie inaczej, jeśli do interfejsu funkcji przeciążającej operator, — `operator+()`, dodane zostaną dwa znaki `&` (ang. *ampersand*) i parametry będą przekazywane poprzez referencje.

```
Rational operator + (const Rational &x, const Rational &y)        // referencje
{ return Rational(y.nmr*x.dnm + x.nmr*y.dnm, y.dnm*x.dnm); }
```

Wymaganie zachowania spójności pomiędzy różnymi częściami kodu w C++ jest postulatem rygorystycznym. W tym przypadku zmieniono interfejs (prototyp funkcji) i analogicznie zmodyfikowano deklarację tejże funkcji w obrębie specyfikacji klasy. (Także tym razem nie ma tu znaczenia, czy jest to metoda, czy też funkcja zaprzyjaźniona kategorii `friend`). W tym przypadku niezapewnienie spójności różnych części kodu programu nie jest śmiertelnym zagrożeniem. W razie czego kompilator powinien nas ostrzec, że kod taki zawiera błędy składniowe.

Wydruk wyjściowy, powstały w wyniku wykonania programu z listingu 11.1 po zmodyfikowaniu funkcji `operator+()` tak jak poprzednio, pokazano na rysunku 11.2. Jak widać, cztery wywołania funkcji przepadły. Dwa obiekty-parametry nie są tworzone (brak komunikatu konstruktora) i odpowiednio dwa obiekty-parametry nie są usuwane (brak komunikatu destruktora).

Rysunek 11.2.
Wydruk wyjściowy programu z listingu 11.2 przy przekazaniu parametrów poprzez referencje

```
obiekt utworzony (konstruktor): 1 4
obiekt utworzony (konstruktor): 3 2
obiekt utworzony (konstruktor): 0 1

obiekt utworzony (konstruktor):  7 4
obiekt przypisany (operator=()): 7 4
obiekt usunięty (destruktor):    7 4
1/4 + 3/2 = 7/4

obiekt usunięty (destruktor):    7 4
obiekt usunięty (destruktor):    3 2
obiekt usunięty (destruktor):    1 4
```

> Unikaj przekazywania obiektów jako parametrów do funkcji poprzez wartość. Powoduje to nadmiarowe (niekonieczne) wywołania funkcji. Przekazuj obiekty do funkcji poprzez referencje, poprzedzając je w prototypie funkcji, na liście argumentów modyfikatorami `const` (jeśli tylko ma to zastosowanie).

Teraz zademonstrujmy różnicę pomiędzy inicjowaniem a przypisaniem. Na listingu 11.1 zmienna `c` podlega działaniu operatora przypisania w wyrażeniu `c = a + b;`. Skąd wiadomo, że jest to przypisanie, a nie zainicjowanie? Ponieważ po lewej stronie nazwy zmiennej `c` nie ma nazwy typu. Typ zmiennej `c` został określony wcześniej, na początku kodu funkcji `main()`. Dla kontrastu, ta wersja funkcji `main()` deklaruje i natychmiast inicjuje obiekt `c` jako sumę obiektów `a` oraz `b` zamiast utworzyć obiekt, a później przypisać mu wartość w dwóch różnych instrukcjach.

```
int main()
{
Rational a(1,4), b(3,2), c = a + b;
a.show(); cout << " +"; b.show(); cout << " ="; c.show();
cout << endl << endl;
return 0;
}
```

Na rysunku 11.3 pokazano wydruk wyjściowy powstały w wyniku wykonania programu z listingu 11.1 z zastosowaniem przekazania parametrów poprzez referencje i z użyciem podanego kodu funkcji `main()`. Jak widać, operator przypisania nie jest tu wywoływany. Nie jest tu wywoływany także konstruktor kopiujący, co stanowi naturalny skutek przejścia od przekazywania parametrów poprzez wartość do przekazania ich poprzez referencje.

Rysunek 11.3.
Wydruk wyjściowy programu z listingu 11.2 przy przekazaniu parametrów poprzez referencje i zastosowaniu inicjowania obiektów zamiast przypisania

```
obiekt utworzony (konstruktor): 1 4
obiekt utworzony (konstruktor): 3 2
obiekt utworzony (konstruktor): 7 4
1/4 + 3/2 = 7/4

obiekt usunięty (destruktor):    7 4
obiekt usunięty (destruktor):    3 2
obiekt usunięty (destruktor):    1 4
```

W dalszej części zastosowana zostanie podobna technika, by zademonstrować różnicę pomiędzy zainicjowaniem a przypisaniem na przykładzie obiektów klasy `String`.

> Rozróżniaj inicjowanie obiektów od przypisywania wartości obiektom. Przy inicjowaniu obiektów wywoływany jest konstruktor, a zastosowanie operatora przypisania zostaje pominięte. Podczas operacji przypisania stosuje się operator przypisania, natomiast wywołanie konstruktora zostaje pominięte.

Niniejsze rozważania i koncepcje odnoszące się do zasadności unikania przekazywania obiektów jako parametrów poprzez wartość oraz rozróżniania inicjowania obiektów od przypisywania im wartości są bardzo ważne. Przekonaj się, że jesteś w stanie przeczytać kod klienta i powiedzieć: „Tu wywoływany jest konstruktor, a tu — operator przypisania”. Doskonal swoją intuicję, by umożliwiała ci dokonywanie tego typu analiz kodów.

Przeciążanie operatorów w klasach nie będących klasami numerycznymi

Jak zaznaczono we wprowadzeniu do tego rozdziału, rozszerzenie działania wbudowanych operatorów tak, by obejmowały klasy numeryczne — jest naturalne. Funkcje operatorowe dokonujące przeciążenia operatorów wobec takich klas są bardzo podobne do działania wbudowanych operatorów. Nieprawidłowa interpretacja symboli takich przeciążonych operatorów i ich znaczenia przez programistę piszącego kod klienta bądź przez serwisanta kodów jest mało prawdopodobna. Taka koncepcja, by wartości elementarnych typów wbudowanych i obiekty typów zdefiniowanych przez programistę były traktowane jednakowo — wydaje się naturalnym sposobem wprowadzenia prostej i zrozumiałej implementacji.

Operatory mogą zostać zastosowane równie dobrze wobec obiektów klas niematematycznych, ale rzeczywiste znaczenie operatora dodawania, odejmowania i innych operatorów może być mało intuicyjne. To nieco przypomina historię ikonek i wprowadzania poleceń w graficznym interfejsie użytkownika.

Na samym początku był interfejs składający się z tekstowego wiersza poleceń (ang. *command line interface*) i użytkownik musiał wpisywać długie polecenia z parametrami, przełącznikami, opcjami, kluczami itp. Następnie wprowadzono menu z listą dostępnych opcji. Poprzez wybór pozycji z menu użytkownik mógł wydać polecenie do wykonania bez potrzeby samodzielnego jego wpisywania. Następnie do takich menu dodano skróty klawiaturowe (ang. *hot keys*). Poprzez naciśnięcie takiej specjalnej kombinacji klawiszy na klawiaturze użytkownik mógł wydać określone polecenie wprost, bez potrzeby odrywania rąk od klawiatury i poszukiwania odpowiedniej pozycji w menu, a czasem w całej strukturze podmenu. W kolejnym kroku wprowadzono *listwy narzędziowe* (ang. *toolbars*) z graficznymi przyciskami odpowiadającymi poszczególnym poleceniom. Wskazawszy taki przycisk polecenia kursorem myszy i kliknąwszy przyciskiem myszy, użytkownik mógł wydać odpowiednie polecenie bez potrzeby zapamiętywania odpowiadającej mu kombinacji klawiszy. Ikonki umieszczane na takich przyciskach poleceń w sposób jednoznaczny i intuitywny (zrozumiały intuicyjnie) były jasne: *Open*, *Close*, *Cut*, *Print* (*Otwórz*, *Zamknij*, *Wytnij*, *Drukuj*). Gdy stopniowo do takich pasków narzędziowych dodawano coraz więcej i więcej ikonek, stopniowo stawały się one coraz mniej intuitywne: *New*, *Paste*, *Output*, *Execute*, *Go* (co odpowiada poleceniom *Nowy*, *Wklej*, *Wyjście*, *Wykonaj*, *Uruchom* itp.).

Aby pomóc użytkownikowi w opanowaniu tych ikonek, dodano wskazówki — podpowiedzi (ang. *tooltips*) wyświetlane automatycznie po wskazaniu danej ikonki kursorem myszy. Interfejs użytkownika stopniowo stawał się coraz bardziej skomplikowany, aplikacje wymagały coraz więcej miejsca na dysku, coraz więcej pamięci operacyjnej i coraz więcej pracy ze strony programistów. Z drugiej strony, mimo tych wszelkich ułatwień, użytkownicy nie mają teraz zapewne łatwiejszego życia niż wtedy, gdy wydawali polecenia, posługując się kombinacjami klawiszy na klawiaturze (ang. *hotkeys*). Na podobnej zasadzie rozpoczęliśmy od przeciążania operatorów wobec klas numerycznych, a teraz zamierzamy zastosować przeciążanie operatorów i funkcji operatorowych w odniesieniu do klas nienumerycznych. Będzie to wymagać od nas opanowania większej liczby reguł, napisania większej ilości kodu

i zetknięcia się z wyższym stopniem złożoności. Z drugiej strony, kod klienta będzie stawał się lepszy dzięki zastosowaniu zamiast starego stylu wywołań funkcji — nowoczesnego przeciążania operatorów.

Klasa String

Omówię tu popularne zastosowanie funkcji przeciążającej operator w odniesieniu do klas nienumerycznych — zastosowanie operatora dodawania + do konkatenacji łańcuchów znaków (dołączenia jednego łańcucha znaków do końca drugiego łańcucha).

Rozważmy klasę `String` zawierającą dwa pola danych — wskaźnik do dynamicznie rozmieszczanej w pamięci tablicy znakowej oraz liczbę całkowitą wskazującą maksymalną liczbę znaków, traktowanych jako ważne dane, które mogą zostać umieszczone w dynamicznie przyporządkowanej pamięci na stercie. W istocie standardowa biblioteka *C++ Standard Library* zawiera klasę `String` (której nazwa rozpoczyna się do małej litery, tj. `class string`), która to klasa jest przeznaczona do spełniania większości wymagań związanych z operowaniem łańcuchami tekstowymi. To wspaniała i bardzo przydatna klasa. Ta firmowa klasa jest znacznie bardziej rozbudowana niż klasa `String`, która będzie tu omawiana. Nie można jednak użyć firmowej klasy `string` do naszych celów, ponieważ jest zbyt skomplikowana i takie techniczne szczegóły odwracałyby naszą uwagę od właściwej dyskusji o dynamicznym zarządzaniu pamięcią i jego konsekwencjach.

Po stronie kodu klienta obiekty naszej klasy `String` mogą być tworzone na dwa sposoby — poprzez określenie maksymalnej liczby obsługiwanych znaków łańcucha tekstowego albo poprzez określenie bezpośrednio zawartości łańcucha tekstowego obsługiwanego przez dany obiekt. Określenie liczby znaków w łańcuchu wymaga podania jednego parametru — liczby całkowitej. Określenie zawartości łańcucha tekstowego także wymaga podania jednego parametru — tablicy znakowej. Typy tych parametrów są różne, dlatego parametry te powinny zostać zastosowane w różnych konstruktorach. Ponieważ każdy spośród tych konstruktorów ma dokładnie jeden parametr typu innego niż typ własnej klasy, który to parametr zostaje poddany konwersji na wartość obiektu danej klasy, obydwa te konstruktory są konstruktorami konwersji.

Pierwszy z tych konstruktorów konwersji, którego parametrem jest liczba całkowita określająca ilość potrzebnej pamięci, którą należy obiektowi przydzielić na stercie, ma domyślną wartość argumentu równą zeru. Jeśli obiekt klasy `String` zostanie utworzony z wykorzystaniem tej domyślnej wartości (tj. jeśli nie wyspecyfikowano żadnej wartości argumentu), domyślna wielkość pamięci przyporządkowana danemu obiektowi w celu przechowywania łańcucha tekstowego zostanie przyjęta jako zerowa. W takim przypadku ten pierwszy konstruktor konwersji zostanie użyty w roli konstruktora domyślnego (tzn. przy składni deklaracji `String s;`).

Drugi spośród tych konstruktorów konwersji, z tablicą znakową (łańcuchem znaków) jako parametrem, nie ma domyślnej wartości argumentu. Przyporządkowanie mu jakiejś wartości domyślnej byłoby dość trudne, chyba żeby zdecydować się tu na pusty łańcuch znaków o zerowej długości: `""`. Wtedy jednak kompilator mógłby mieć wątpliwości z racji niejednoznaczności przy wywołaniu konstruktora domyślnego, tj. w razie deklaracji obiektu w postaci

`String s;`. Pytanie sprowadzałoby się do tego, czy chcemy tu wywołać pierwszy z konstruktorów z domyślną wartością zerowej długości łańcucha znaków, czy też drugi spośród konstruktorów z domyślną wartością — pustym łańcuchem znaków (ang. *empty string*).

Bieżąca zawartość łańcucha tekstowego może być modyfikowana ze strony kodu klienta poprzez wywołanie metody `modify()` (zmodyfikuj), pozwalającej na wyspecyfikowanie nowego łańcucha tekstowego, który po wywołaniu tej metody ma zawierać docelowy obiekt komunikatu. Aby uzyskać dostęp do zawartości obiektu klasy `String`, po stronie kodu klienta można posłużyć się metodą `show()` (pokaż). Ta metoda zwraca wskaźnik do pamięci przydzielonej na stercie dla danego obiektu. Ten wskaźnik może zostać użyty w kodzie klienta do wydrukowania zawartości tegoż łańcucha tekstowego, porównania jej z innym łańcuchem znaków (tekstem) itp. Na listingu 11.2 pokazano przykładowy program zawierający implementację klasy `String`.

Listing 11.2. *Klasa String z dynamicznym przydziałem pamięci na stercie*

```
// klasa String dynamicznie zapamiętuje macierz znakową
#include <iostream>
using namespace std;
class String
{
char *str;                    // dynamicznie zapamiętuje tablicę znakową
int len;
public:
String (int length=0);        // konstruktor domyślny / konwersji
String(const char*);          // konstruktor konwersji
~String ();                   // zwolnienie dynamicznej pamięci
void modify(const char*);     // zmiana zawartości tablicy znakowej
char* show() const;           // zwrot wskaźnika do tej tablicy
} ;
String::String(int length)
{ len = length;
str = new char[len+1];        // domyślny rozmiar to 1
if (str==NULL) exit(1);       // sprawdź, czy się powiodło
str[0] = 0; }                 // pusty łańcuch o długości 0 – to jest OK
String::String(const char* s)
{ len = strlen(s);            // zmierz długość wczytanego tekstu
str = new char[len+1];        // przydziel wystarczającą ilość miejsca na stercie
if (str==NULL) exit(1);       // sprawdź, czy się powiodło
strcpy(str,s); }              // kopiuj tekst do nowego obszaru na stercie
String::~String()
{ delete str; }                         // zwrot pamięci na stercie (nie wskaźnika!)
void String::modify(const char a[])     // żadnego zarządzania pamięcią
{ strncpy(str,a,len-1);                 // ochrona przed przepełnieniem (obcinanie)
str[len-1] = 0; }                       // poprawne zakończenie łańcucha znaków
char* String::show() const              // to nie jest zalecana praktyka, ale OK
{ return str; }

int main()
{
String u("This is a test.");
// po spolszczeniu byłoby: String u("To jest test.");
String v("Nothing can go wrong.");
// po spolszczeniu byłoby: String u("Nic nie może pójść źle.");
cout << " u = " << u.show() << endl;    // rezultat jest OK
```

```
cout << " v = " << v.show() << endl;    // rezultat jest OK
// tu nastąpi obcięcie nadmiarowych znaków
v.modify("Let us hope for the best.");
// po spolszczeniu: v.modify("Nasza nadzieja w b (po 'b' - obcięcie).");
cout << " v = " << v.show() << endl;
strcpy(v.show(),"Hi there");            // niezalecana praktyka
// po spolszczeniu byłoby: strcpy(v.show(),"Hej tam");
cout << " v = " << v.show() << endl;
return 0;
}
```

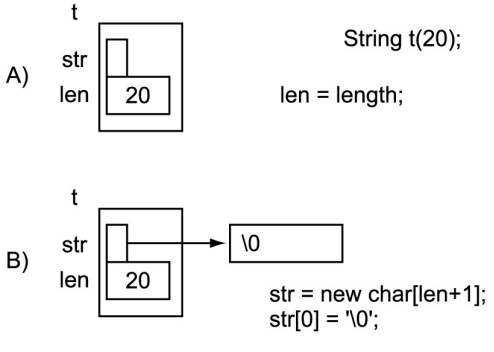
Dynamiczne zarządzanie pamięcią na stercie

Pierwszy wiersz w kodzie pierwszego spośród dwu konstruktorów konwersji ustawia wartość pola danych `len` (długość łańcucha znaków). Drugi wiersz kodu ustawia wartość pola danych `str` (wskaźnik do łańcucha znaków) poprzez przydział wymaganej ilości pamięci na stercie. Następnie w kodzie sprawdza się, czy operacja przyporządkowania pamięci na stercie zakończyła się powodzeniem i do pierwszego bajta przydzielonej na stercie pamięci wpisywany jest znak zerowy[3]. Dla wszystkich funkcji bibliotecznych C++ taki łańcuch znaków jest łańcuchem pustym, choć zajmuje w pamięci miejsce przeznaczone na ilość znaków określoną po stronie kodu klienta.

Jeśli kod klienta definiuje obiekt klasy `String`, a nie dostarcza żadnych argumentów, ten konstruktor zostaje użyty jako konstruktor domyślny, powodując przydział na stercie pamięci przeznaczonej na pojedynczy znak. Ten jedyny znak zostaje zainicjowany jako „terminator" (`\0`), sygnalizując pusty łańcuch znaków.

Na rysunku 11.4 pokazano schemat stanu pamięci na stercie, śledząc krok po kroku wykonanie kolejnych instrukcji wchodzących w skład tego konstruktora po jego wywołaniu w wyniku następującej instrukcji:

```
String t(20);              // 21 znaków na stercie
```



Rysunek 11.4. *Schemat obsługi pamięci dla pierwszego konstruktora konwersji z listingu 11.2*

Na rysunku 11.4a pokazano pierwszą fazę konstruowania obiektu, a na rysunku 11.4b odpowiednio drugą fazę. Prostokąt przedstawia obiekt `t` klasy `String` zawierający dwa pola danych — wskaźnik `str` i liczbę całkowitą `len`. Te pola danych w rzeczywistości mogą

[3] `\0` to kod terminatora łańcucha tekstowego — *przyp. tłum.*

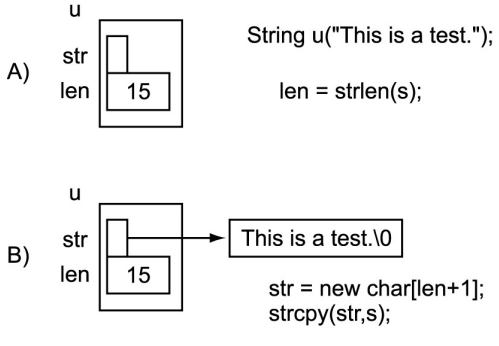
zajmować taką samą ilość pamięci, ale na schematycznym rysunku wskaźnik `str` został oznaczony jako mniejszy prostokąt, by optycznie zasygnalizować, że to pole nie zawiera przetwarzanych danych. Nazwa obiektu `t` i nazwy jego pól danych `str` oraz `len` zostały umieszczone poza prostokątem symbolizującym sam obiekt.

Część schematu (A) pokazuje, że po wykonaniu instrukcji `len = length` pole danych `len` zostaje zainicjowane i przyjmuje wartość 20 (innymi słowy, zawiera wartość 20), natomiast wskaźnik `str` pozostaje niezainicjowany (wskazuje przypadkowy adres w pamięci, czyli co mu się podoba). Część schematu (B) pokazuje, że po wykonaniu pozostałych instrukcji, wchodzących w skład kodu ciała tego konstruktora, przydzielone na stercie miejsce (dla 21 znaków) jest wskazywane poprzez wskaźnik `str` i pierwszy znak jest ustawiony (ma wartość) jako znak zerowy. Rysowanie takiego schematu dla tak prostego obiektu może się wydawać przesadą, ale autor radzi, by rysować takie diagramy dla wszystkich kodów manipulujących wskaźnikami i dynamiczną pamięcią na stercie. To najlepszy sposób do rozwoju naszej intuicji programistycznej odnośnie dynamicznego zarządzania pamięcią.

W pierwszym wierszu kodu ciała drugiego konstruktora konwersji następuje pomiar długości łańcucha znaków wyspecyfikowanego po stronie kodu klienta i zainicjowanie wartości pola danych `len`. Drugi wiersz kodu inicjuje pole danych `str` poprzez przydział wymaganej ilości pamięci na stercie, której początek ma wskazywać wskaźnik `str`, a następnie kopiuje łańcuch znaków dostarczony po stronie kodu klienta do przydzielonej na stercie pamięci. Funkcja biblioteczna `strcpy()` (kopiuj łańcuch znaków) kopiuje łańcuch znaków z tablicy znakowej będącej argumentem konstruktora i po zakończeniu kopiowania na końcu łańcucha dodaje znak terminatora o kodzie zerowym.

Na rysunku 11.5 pokazano schematycznie kolejne stadia inicjowania obiektu w rezultacie wykonania następującej instrukcji.

```
String u("This is a test.");              // 15 symboli, 16 znaków na stercie
```



Rysunek 11.5. *Schemat obsługi pamięci dla drugiego konstruktora konwersji z listingu 11.2*

Są trzy metody postępowania z polem danych zawierającym wielkość obszaru pamięci przyporządkowanego na stercie. Pierwszy sposób polega na tym, że takie pole zawiera całkowitą wielkość pamięci przydzieloną dla określonego pola danych (liczba znaków do przechowywania plus jeden). Drugi sposób polega na tym, że takie pole zawiera liczbę rzeczywiście użytecznych znaków po dodaniu do tej liczby jedynki. Takie dodanie jedynki następuje wtedy, gdy dane są umieszczane w pamięci przyporządkowanej na stercie. My użyjemy tu tej drugiej metody, choć zdecydowanie trudno byłoby wyjaśnić, dlaczego ten drugi sposób miałby być lepszy od pierwszego. Autor nie zmieni jednak zdania, ponieważ wtedy musiałby wyjaśnić, dlaczego ten pierwszy sposób jest lepszy od tego drugiego.

Trzeci sposób polega na tym, by nie przechowywać długości łańcucha znaków jako zawartości numerycznego pola danych ani nie przechowywać jej wcale, lecz za każdym razem obliczać tę długość w czasie wykonania programu poprzez wywołanie funkcji bibliotecznej `strlen()`. To przykład względności, a właściwie współzależności czasu i przestrzeni. Ten trzeci sposób jest lepszy, gdy długość łańcucha znaków nie jest nam przydatna często, a z drugiej strony odczuwamy niechęć do zajmowania pamięci przez dodatkową liczbę całkowitą w każdym obiekcie obsługującym łańcuchy znaków.

Skoro dynamiczny przydział pamięci na stercie następuje dla każdego obiektu w sposób indywidualny, wielu programistów może mieć odczucie, że taka dynamicznie przyporządkowana pamięć powinna być rozpatrywana jako część obiektu. Przy takim podejściu do problemu, obiekty klasy `String` mogą być postrzegane od strony kodu klienta jako obiekty o zmiennej długości, zależnej od przyporządkowanej dla danego obiektu wielkości pamięci na stercie. Taki punkt widzenia jest uprawniony, ale tego typu spojrzenie prowadzi do bardziej skomplikowanych i zaskakujących rozważań o działaniu konstruktorów i destruktorów, a i zakłóca nieco koncepcję samej klasy.

Autor preferuje tu podejście zaprezentowane na diagramach pokazanych na rysunkach 11.4 oraz 11.5. Odzwierciedla to zasadę C++, że klasa jest planem-szablonem dla poszczególnych obiektów danej klasy. Taki plan-szablon jest tu taki sam dla wszystkich obiektów klasy `String`. Zgodnie z tym planem, każdy obiekt klasy `String` zawiera dwa pola danych, a wielkość każdego obiektu klasy `String` jest taka sama. Gdy po stronie kodu klienta następuje wykonanie następującej instrukcji:

```
String t(20);                // dwa pola danych, przydział pamięci na stosie
```

Dla obiektu `t` zostaje na stosie przydzielona pamięć przeznaczona na jego dwa pola danych. Przydział pamięci na stercie następuje za pośrednictwem należących do tego obiektu klasy `String` metod, które zostają wywołane i wykonują się w odniesieniu do danego, konkretnego obiektu. Różne obiekty klasy `String` mogą mieć przyporządkowaną różną wielkość dynamicznej pamięci na stercie. Mogą (niezależnie) zwalniać tę pamięć albo powiększać jej objętość, nie zmieniając swojej tożsamości.

Takie podejście w niczym się nie zmienia, jeśli same obiekty, zamiast na stosie, lokalizowane będą dynamicznie na stercie. Rozważmy następujący przykład kodu klienta.

```
String *p;
// żadnego obiektu klasy String, na stosie tworzony jest tylko wskaźnik
p = new String ( "Hi!" );          // 2 pola danych plus 4 znaki na stercie
```

W tym przypadku obiekt klasy `String` bez nazwy (wskazywany poprzez wskaźnik `p`) jest tworzony na stercie, zajmując tam miejsce przeznaczone na jedno własne pole danych — liczbę całkowitą i na drugie własne pole danych — wskaźnik do łańcucha znaków. Po utworzeniu tego obiektu następuje wywołanie jego konstruktora i tenże konstruktor dokonuje przyporządkowania pamięci na stercie dla wskazywanego łańcucha znaków `Hi!`, czyli dodatkowo dla czterech znaków — oczywiście także na stercie. Prócz tego konstruktor inicjuje wskaźnik `str` tak, by wskazywał na stercie początek tego czteroznakowego łańcucha znaków.

Takie podejście pozwala na komfort myślenia o tych wszystkich obiektach jako o obiektach tej samej klasy i tej samej wielkości. Gdy tworzony jest nowy taki obiekt, przebiegają tu dwa odrębne procesy — proces tworzenia obiektu (zawsze tej samej wielkości) i wywołanie

konstruktora dla tego obiektu (w celu jego zainicjowania). Konstruktor inicjuje pola danych obiektu, włącznie ze wskaźnikiem, który zaczyna wskazywać obszar pamięci na stercie.

Destruktor zwalnia dynamicznie przyporządkowaną na stercie pamięć. Destruktor jest wywoływany bezpośrednio przed usunięciem obiektu. Gdy obiekt jest usuwany, pamięć przyporządkowana dla jego pól `len` oraz `str` zostaje zwolniona i udostępniona do dalszego swobodnego wykorzystania. Jeśli pamięć zostaje obiektowi przydzielona na stosie (tak jest w przypadku obiektów `u`, `v` w funkcji `main()` na listingu 11.2), po usunięciu obiektu zostaje zwolniona i zwrócona na stos. Jeśli sam obiekt został umieszczony na stercie (tak, jak obiekt bez nazwy wskazywany przez wskaźnik `p`), pamięć przydzielona dla jego pól `str` oraz `len` zostaje zwrócona na stertę. Jednakże we wszystkich przypadkach, w których destruktor zwalnia pamięć (destruktor zwalnia pamięć wskazywaną na stercie przez wskaźnik `str`), ta pamięć zostaje zwrócona na stertę, zanim jeszcze znikną pola danych `len` oraz `str`. W przeciwnym razie instrukcja `delete str;` w obrębie destruktora stałaby się nielegalna.

Funkcja `modify()` powoduje zmianę zawartości dynamicznie przyporządkowanej pamięci na stercie. Posługując się funkcją biblioteczną `strncpy()`, by upewnić się, że zawartość pamięci nie zostanie uszkodzona, funkcja ta dokonuje poprawnie kopiowania określonej ilości znaków, nawet wtedy, gdy kod klienta omyłkowo dostarczy łańcuch znaków, który jest większy niż ilość pamięci przydzielona na stercie dla obiektu klasy `String`. W razie przepełnienia funkcja `strncpy()` nie dodaje na końcu łańcucha znaków znaku terminatora. Z tego powodu autor dodaje taki znak pod koniec kodu funkcji. Wydaje się to nadmiarowe w sytuacji, gdy nowy łańcuch znaków jest krótszy niż ilość miejsca dostępnego w pamięci. Pamiętaj, że w takim przypadku funkcja `strncpy()` zawsze wypełni zerami pozostałe miejsce w pamięci, a dodanie jednego zera więcej nie spowoduje spowolnienia działania programu.

Funkcja `modify()` nie może rozszerzyć łańcucha znaków ponad jego pierwotną długość. Większość projektów klas w rodzaju `String` nie pozwala programiście na zmodyfikowanie zawartości obiektu klasy `String`. Skoro potrzebna jest inna zawartość obiektu, nie pozostaje nic innego, jak tylko utworzyć i użyć innego obiektu z potrzebną nam zawartością. Autor wybrał tu implementację kompromisową. Pełne oprzyrządowanie pozwalające na modyfikacje wymagałoby znacznie dłuższego kodu i zarazem przedyskutowania wielu dodatkowych zagadnień. Taka skromna funkcja `modify()` jest zupełnie wystarczająca do niniejszej dyskusji.

Funkcja `show()` zwraca wskaźnik do dynamicznie przydzielonej pamięci na stercie. Listing 11.2 demonstruje dwa sposoby zastosowania tej funkcji po stronie kodu klienta w obrębie funkcji `main()`. Pierwsze zastosowanie tej funkcji następuje w celu wydrukowania zawartości obiektu klasy `String`, który jest obiektem docelowym komunikatu (tego wywołania metody `show()`). Drugie wywołanie następuje w celu zmodyfikowania zawartości obiektu poprzez wykorzystanie wartości zwracanej przez tę metodę jako parametru wyjściowego w wywołaniu funkcji `strcpy()` w kodzie klienta. Pierwsze zastosowanie jest uprawnione, drugie jest aroganckie i napisane raczej w celu wprawienia serwisanta kodu w zakłopotanie niż udzielenia mu pomocy w zrozumieniu intencji projektanta kodu.

Jeden z pierwszych komputerowych języków programowania wysokiego poziomu, APL (skrót od *A Programming Language*), był bardzo skomplikowany. Jest nadal stosowany, głównie w aplikacjach finansowych. Zestaw znaków tego języka jest tak ogromny, że wymaga on stosowania specjalnych klawiatur. Język ten zawiera między innymi potężne operacje do *przetwarzania macierzy i tablic* (ang. *arrays and matrix processing*). Programiści lubią język APL. Za wyraz dobrego smaku uważa się dowcip polegający na napisaniu kilku wierszy kodu w języku APL i pokazaniu go przyjaciołom z pytaniem: „Zgadnij, co to znaczy?”.

Autor jest daleki od sugerowania, jakoby programistów o takim sposobie myślenia należało zwalniać z pracy. Nie mogą oni jednak uczestniczyć w grupowych projektach, w których inne osoby muszą prowadzić obsługę techniczną ich kodów. W dzisiejszych czasach zupełnie nie ma się czym chwalić, jeśli programista pisze takie kody, których zrozumienie wymaga dodatkowego wysiłku.

```
// to nie jest zalecany styl programowania
strcpy(v.show(), "Hej tam!");
```

Zauważ, że oburzenie autora skierowane jest przede wszystkim przeciw takim faktom, gdy serwisant kodu musi włożyć dodatkowy wysiłek w zrozumienie kodu. To, że podany kod nie dokonuje uprzedniej oceny rozmiaru pamięci dostępnej na stercie w obrębie danego obiektu i z tego powodu może doprowadzić do uszkodzenia danych w pamięci jest oczywiście ważne. Ale to tylko drobiazg, który jednak powoduje wyczerpanie się naszej cierpliwości. Można to skorygować poprzez inny podział odpowiedzialności pomiędzy kodem klienta a obiektem — serwerem klasy `String`.

```
int length = strlen(v.show());            // podaj rozmiar
strncpy(v.show(),"Hi there", length);     // przekazać odpowiedzialność w górę
// po spolszczeniu: strncpy(v.show(),"Hej tam", dlug_tekstu);
```

W przypadku obiektów klasy `String` utworzonych za pomocą drugiego konstruktora konwersji wartość parametru `length` to całkowita dostępna przestrzeń w pamięci. Dla obiektów utworzonych za pomocą pierwszego konstruktora konwersji wartość parametru `length` określa ostatni łańcuch znaków, zapamiętany przy zastosowaniu danego obiektu. Ta ostatnia długość może być mniejsza niż cała dostępna przestrzeń w pamięci. Co najważniejsze, ta metoda narusza zasadę przenoszenia odpowiedzialności w dół z kodu klienta na kody serwerów i zasadę ukrywania szczegółów technicznych dotyczących manipulowania danymi przed kodem klienta.

W tym przypadku to kod klienta wykonuje operacje manipulowania danymi na niskim poziomie, nawet jeśli nazwy pól obiektów klasy `String` nie są stosowane w kodzie klienta. Jeśli chcemy ochronić dane znajdujące się na stercie przed ryzykiem ich uszkodzenia, to kod serwera powinien zawierać odpowiednie instrukcje, które sprawdzają rozmiar dostępnej dynamicznej pamięci. Dobrym rozwiązaniem powinno tu być użycie po stronie klienta nazwy funkcji usługowej (metody serwera) zamiast manipulowania danymi serwera bezpośrednio i przerzucenie odpowiedzialności za ochronę spójności pamięci na stercie na serwer. Czy to jest oczywiste? Oto rozwiązanie, które wykonuje tę pracę dobrze, jest bezpieczne i nie wymaga żadnych dodatkowych wyjaśnień. To rozwiązanie już widziałeś.

```
// w oryginale – łańcuch znaków "Hi there":
v.modify("Hej tam!");      // tu testowany jest rozmiar dostępnej pamięci
```

Na rysunku 11.6 pokazano wydruk wyjściowy programu z listingu 11.2. Ten wydruk demonstruje, że wywołanie funkcji `modify()` chroni dynamiczną pamięć przed przepełnieniem (ang. *overflow*) poprzez obcięcie danych z kodu klienta (ang. *truncate*).

Rysunek 11.6.
Wydruk wyjściowy programu z listingu 11.2

```
u = To jest tekst
v = Nic nie może pójść źle
v = Nasza nadzieja w 'b'
v = Hej tam
```

Zastosowanie wskaźnika zwróconego przez funkcję `show()` nie zawiera takiej ochrony pamięci. Oto przykład uszkodzenia danych w pamięci, do którego może doprowadzić metoda `String::show()`.

```
char *ptr = v.show();       // lekkomyślność i brak rozwagi
ptr[200] = 'A';             // uszkodzenie zawartości pamięci
```

Albo jeśli wolimy notację łańcuchową przy stosowaniu obiektów, możemy zrobić to samo, używając tylko jednej instrukcji.

```
v.show()[200] = 'A';        // lekkomyślność, uszkodzenie pamięci
```

To nie jest dobry ani zalecany sposób programowania.

Ochrona danych na stercie należących do obiektu od strony kodu klienta

C++ zapewnia nam sposób ochrony wnętrza obiektu od strony kodu klienta, który posługuje się wskaźnikiem zwróconym przez metodę. Zadeklarowanie takiego wskaźnika jako wskaźnika do stałej zapobiega takim kłopotom. Na przykład zdefiniujmy wartość zwracaną przez metodę `show()` jako wskaźnik do stałej znakowej, a nie jako wskaźnik do zmiennej znakowej, jak to było na listingu 11.2.

```
// zwrot wskaźnika do stałej z metody show():
const char* String::show() const            // zalecana praktyka
```

Teraz, gdyby kod klienta podjął próbę zmodyfikowania zawartości dynamicznie przyporządkowanej pamięci za pośrednictwem wskaźnika będącego wartością zwracaną poprzez metodę `show()`, kompilator oznaczy taką próbę jako błąd składniowy.

```
strcpy(v.show(), "Hi there");               // "Hi there", czyli "Hej tam"
// to nie złe maniery, lecz błąd składni
```

Przy takim projekcie klasy usługowej (serwera) `String` kod klienta jest zmuszony do posługiwania się funkcją `modify()` w celu zmiany stanu obiektu. W efekcie kod klienta zostaje wyrażony w formie wywołań funkcji serwera, odpowiedzialność za bezpieczeństwo operacji zostaje przekazana w dół na klasę-serwer, kod klienta nie jest zmuszany, by przejmować się technicznymi szczegółami wewnętrznej konstrukcji projektu klasy (tj. nie musi wiedzieć o ograniczeniach miejsca na stercie).

Przeciążony operator konkatenacji łańcuchów znakowych

Naszym kolejnym krokiem będzie zaprojektowanie funkcji operatorowej dokonującej przeciążenia operatora dodawania wobec klasy `String` w taki sposób, by operator ten pozwalał na konkatenację dwóch obiektów klasy `String`. Konkatenacja obiektów będzie polegać na dołączeniu zawartości drugiego obiektu (tekstu wskazywanego przez wskaźnik) do końca zawartości pierwszego obiektu (tekstu wskazywanego przez wskaźnik). Oznacza to, że po stronie kodu klienta taki przeciążony operator będzie mógł zostać użyty w następujący sposób.

```
String u("This is a test. ");               // lewy operand
String v("Nothing can go wrong.");          // prawy operand
u += v; // wyrażenie: operand, operator, operand
```

Po wykonaniu tego fragmentu kodu klienta zawartość reprezentowana przez obiekt `v` powinna pozostać bez zmian, natomiast zawartość reprezentowana przez obiekt `u` powinna stanowić połączenie dwóch łańcuchów znaków: `This is a text. Nothing can go wrong.`

Jeśli zaimplementujemy ten operator w formie metody operatorowej, to obiekt `u` powinien być docelowym obiektem komunikatu, natomiast obiekt `v` powinien być bieżącym argumentem w danym wywołaniu takiej metody operatorowej. Rzeczywiste znaczenie ostatniego wiersza w podanym fragmencie kodu jest następujące:

```
// znaczenie u += v; -> u to obiekt docelowy, v to parametr
u.operator+=(v);
```

Skoro tak, interfejs tej funkcji powinien zawierać modyfikator `const` w odniesieniu do parametru funkcji, natomiast nie powinien zawierać modyfikatora `const` w odniesieniu do samej metody (tj. do modyfikowanego docelowego obiektu komunikatu). Typem wartości zwracanej przez tę metodę może być `void`. Ograniczy to stosowanie operatora, nie zezwalając na składnię wyrażeń łańcuchowych, ale z punktu widzenia autora kodu klienta — nie jest to poważne ograniczenie.

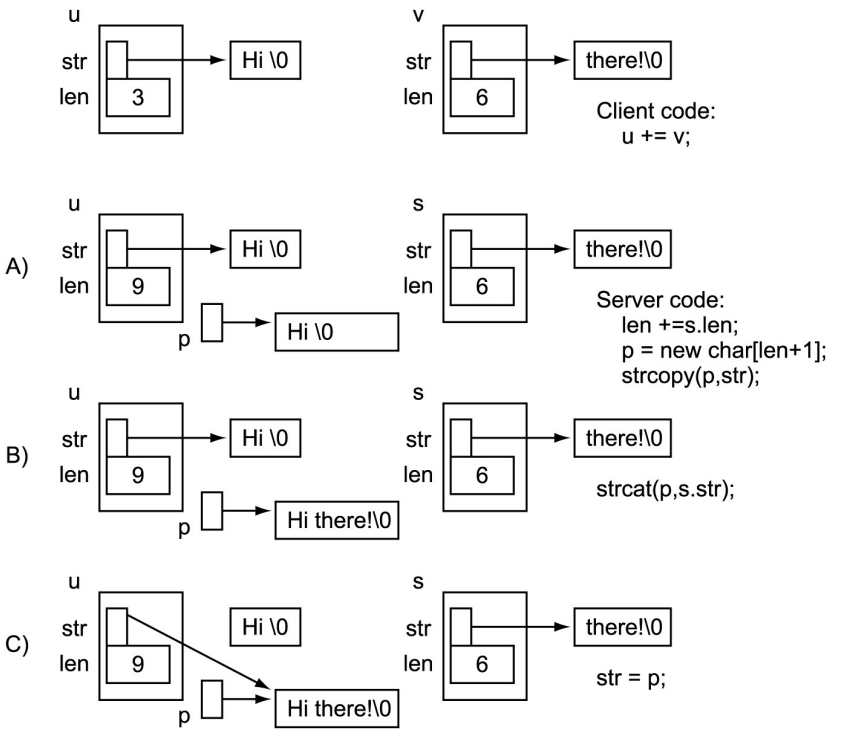
```
// konkatenacja – dołączamy parametr do obiektu docelowego
void operator += (const String s);
```

Pamiętamy, że nie zaleca się przekazywania obiektów poprzez wartość, ale przyjmujemy założenie, że w tym przypadku nie mamy problemów z efektywnością działania programu. W końcu obiekt klasy `String` zawiera tylko dwa niewielkie pola danych — wskaźnik do łańcucha znaków i liczbę całkowitą. Kopiowanie takich pól danych nie powinno potrwać zbyt długo.

Algorytm konkatenacji łańcuchów znakowych powinien zawierać następujące czynności:

1. Dodanie długości dwóch tablic znakowych, by określić sumaryczną, wynikową liczbę znaków.
2. Przydzielić dynamicznie na stercie ilość pamięci wystarczającą do przechowywania tej wynikowej liczby znaków plus znak końca — zero.
3. Sprawdzenie powodzenia operacji przydziału pamięci i przerwanie operacji, jeśli w systemie zabrakło pamięci.
4. Kopiowanie łańcucha znaków z docelowego obiektu komunikatu do nowej, przydzielonej na stercie pamięci.
5. Kopiowanie tablicy znakowej z obiektu przekazanego jako argument do nowej, przydzielonej pamięci (z dołączeniem do końca poprzedniego łańcucha).
6. Ustawienie wskaźnika `str` w obiekcie docelowym komunikatu tak, by wskazywał nową przydzieloną na stercie pamięć.

Na rysunku 11.7 pokazano schematycznie te czynności (poza przerwaniem, gdyby w systemie zabrakło pamięci na stercie) i instrukcje C++ implementujące te działania. Po stronie kodu klienta autor użył nieco krótszych łańcuchów znaków, by ułatwić śledzenie zdarzeń.



Rysunek 11.7. *Schemat pamięci dla funkcji operatorowej konkatenacji łańcuchów znakowych*

W najwyższej części rysunku pokazano dwa obiekty klasy `String`, obiekt `u` (reprezentujący tekst `Hi`) oraz obiekt `v` (reprezentujący tekst `there`). Stadium (A) pokazuje obydwa te obiekty po tym, jak pole `len` pierwszego obiektu zostało zmodyfikowane, przydzielona została pamięć na stercie i istniejąca (początkowa) zawartość obiektu `u` została skopiowana na stertę do tej nowej, przydzielonej pamięci (po wykonaniu kroków 1 – 4 algorytmu). Stadium (B) pokazuje stan pamięci na stercie po wykonaniu kroku 5. Stadium (C) pokazuje stan obu tych obiektów po tym, jak wskaźnik `str` zawarty w docelowym obiekcie `u` zaczął wskazywać nową przyporządkowaną pamięć (krok 6).

Po zebraniu tego wszystkiego razem otrzymujemy następujący kod po stronie serwera:

```
void String::operator += (const String s)    // obiekt jako parametr
{ char* p; // lokalny wskaźnik
len = strlen(str) + strlen(s.str);      // całkowita długość
p = new char[len + 1];                  // przydział pamięci na stercie
if (p==NULL) exit(1);                   // sprawdzenie, czy się powiodło
strcpy(p,str);                          // kopiowanie pierwszej części tekstu wynikowego
strcat(p,s.str);                        // konkatenacja drugiej części tekstu
str = p; }                              // ustaw str, by wskazywał tę nową pamięć
```

Może wydać się nieco przesadne, by w celu wyjaśnienia kolejnych kroków tak prostego algorytmu zagłębiać się w tak drobne szczegóły i wykreślać odrębny rysunek dla każdego maleńkiego kroku w zarządzaniu pamięcią. Jeśli tak to odczuwasz — to bardzo dobrze. Ale należysz do szczęśliwej mniejszości. Dla większości ludzi operacje przy użyciu wskaźników są tajemnicze i sprzeczne z intuicyjnym wyczuciem.

Tylko doświadczeni programiści są tu w stanie zauważyć, że przestrzeń na stercie posiadana przez docelowy obiekt komunikatu nie zostaje poprawnie zwrócona do swobodnego wykorzystania. Ten rysunek pokazuje to w sposób klarowny.

Autor uważa, że rysowanie takich schematów to jedyny sposób na wyrobienie sobie intuicji w odniesieniu do zarządzania pamięcią i wychwytywania błędów. Lepiej spędzić kilka dodatkowych minut na rysowaniu i planowaniu, niż stracić później godziny z debugerem i z innymi skomplikowanymi narzędziami, poszukując drogi w gąszczu, bagnach i zaroślach, pośród instrukcji, których znaczenie i działanie nie jest dla nas do końca zrozumiałe.

Takie rysunki pozostają, oczywiście, tylko narzędziami. To my musimy tak używać tych narzędzi, by upewnić się, że dokładnie rozumiemy każdą instrukcję.

Zapobieganie wyciekom pamięci

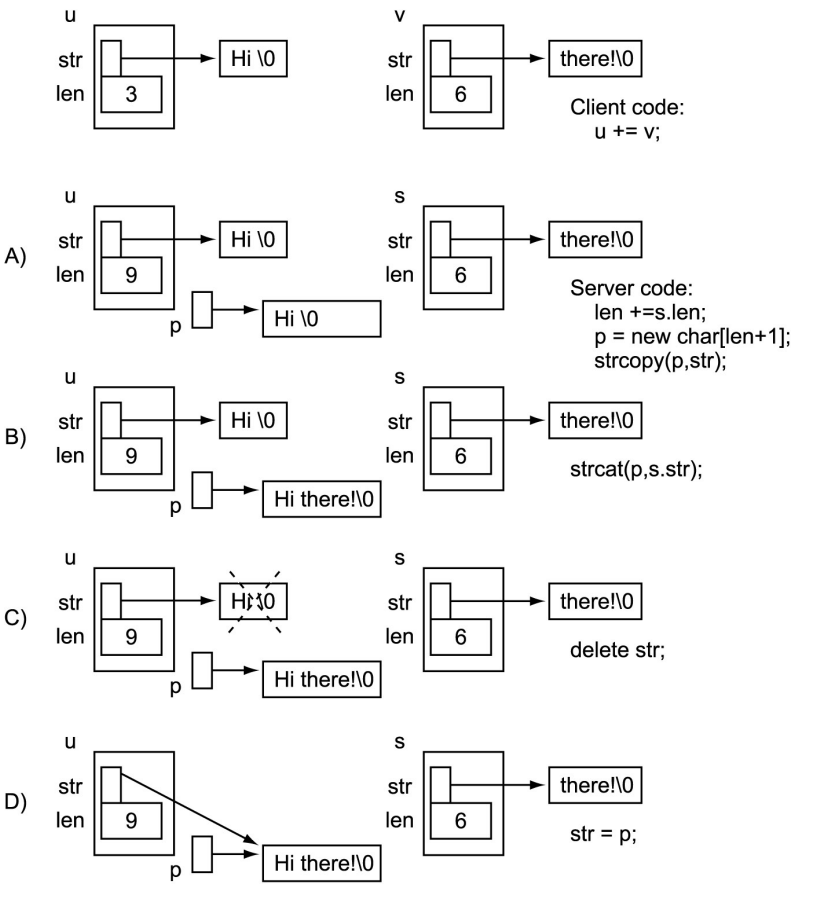
Jak już wspomniałem, na rysunku 11.7 pokazano, że pamięć zajęta przez tablicę znakową na stercie, a wskazywana przez wskaźnik `str` docelowego obiektu na początku wykonania funkcji, nie jest poprawnie zwracana do ponownego użytku. Gdy wskaźnik `str` zostaje skierowany tak, by wskazywał nowy, przyporządkowany segment pamięci (wskazywany przez lokalny wskaźnik `p`), ta stara pamięć staje się niedostępna (ang. *unaccessible*). To wyciek pamięci — powszechny błąd przy manipulowaniu wskaźnikiem w zarządzaniu pamięcią. Aby zapobiec takiemu wyciekowi pamięci, pamięć zajmowana przez starą tablicę znakową powinna zostać poprawnie zwrócona na stertę, zanim jeszcze wskaźnik `str` zostanie przestawiony i zacznie wskazywać nową przydzieloną pamięć i nową (wynikową) macierz znakową.

```
void String::operator += (const String s)   // obiekt jako parametr
{ char* p;                                  // lokalny wskaźnik
len = strlen(str) + strlen(s.str);          // całkowita długość
p = new char[len + 1];                      // przydział pamięci na stercie
if (p==NULL) exit(1);                       // sprawdź, czy się powiodło
strcpy(p,str);                              // kopiowanie pierwszej części
strcat(p,s.str);                            // konkatenacja drugiej części
delete str;                                 // zwrot poprzedniej dynamicznej pamięci
str = p; }                                  // ustaw str, by wskazywał nową pamięć
```

Rysunek 11.8 jest podobny do rysunku 11.7. Pokazuje schematycznie, jak tablica znakowa umieszczona na stercie, a wskazywana poprzez wskaźnik `str` docelowego obiektu komunikatu znika w wyniku zadziałania operatora `delete`. Dopiero potem wskaźnik `str` zostaje przestawiony tak, by wskazywał nową macierz znakową na stercie.

Po usunięciu tego wycieku pamięci należałoby się przyznać, że w tej dyskusji o funkcji dokonującej przeciążenia operatora autor mówił samą prawdę i tylko prawdę, ale nie powiedział jeszcze całej prawdy. Powodem było to, że najpierw należało się upewnić, że pokonaliśmy już mniejsze i łatwiejsze przeciwności, zanim staniemy oko w oko z bardziej skomplikowanymi i bardziej niebezpiecznymi problemami. Autor chciał utrzymać uwagę czytelnika w stanie niepodzielnym.

Ta dyskusja ma za zadanie zaprezentowanie swoistych szablonów postępowania i niebezpieczeństw, które powinniśmy rozpoznawać, gdy piszemy własne programy w C++. Istota tego problemu to, rzec można, ulubiony przeciwnik autora — przekazywanie obiektów jako parametrów poprzez wartość.



Rysunek 11.8. *Schemat pamięci dla skorygowanej funkcji operatorowej konkatenacji łańcuchów znakowych wobec klasy String*

Ochrona integralności programu

Gdy bieżący argument, obojętne — obiekt, czy też nie — jest przekazywany poprzez wartość, jego wartość zostaje skopiowana do utworzonej na stosie lokalnej automatycznej zmiennej. Taka kopia jest wykonywana metodą komponent po komponencie. To nie stanowi żadnego problemu dla danych wbudowanych typów elementarnych, ale jest pewną uciążliwością i powoduje nieznaczne pogorszenie efektywności w przypadku takich klas, jak `Rational` lub `Complex`. Stanowi to realny problem, z punktu widzenia efektywności działania kodu, dla większych klas, których obiekty wymagają większych ilości pamięci.

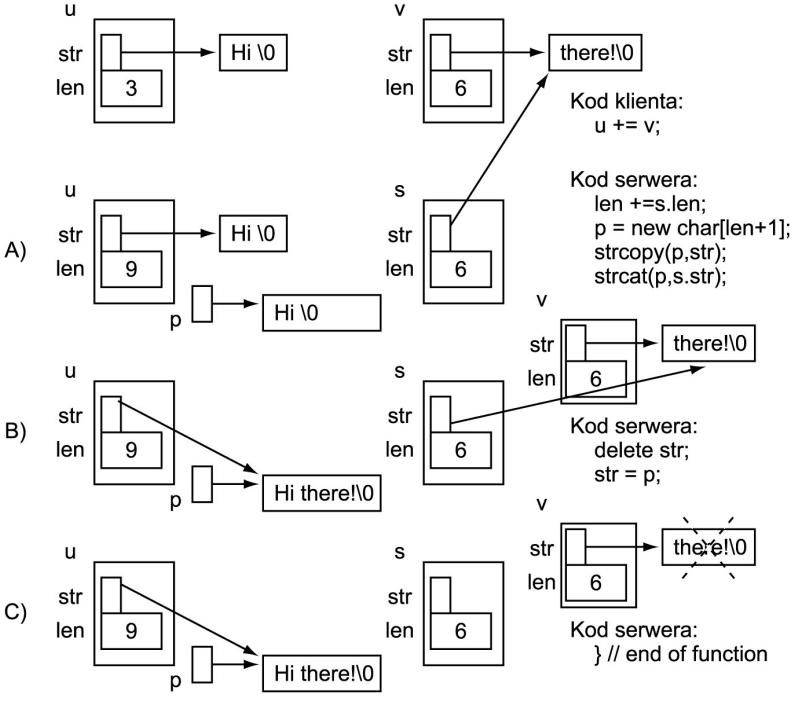
Co najważniejsze, stanowi to ogromny problem z punktu widzenia integralności programu, jeśli taka klasa zawiera wskaźniki wskazujące dynamicznie przydzieloną pamięć na stercie. Popatrzmy na wykonanie takiej funkcji z parametrem przekazywanym poprzez wartość w krytycznych momentach wykonania funkcji — na początku, w chwili wywołania funkcji i na końcu, gdy wykonanie funkcji zostaje zakończone. Autor lubi kojarzyć te momenty z klamrowym nawiasem otwierającym i z klamrowym nawiasem zamykającym, ograniczającymi kod ciała funkcji.

Gdy podczas przekazywania parametru poprzez wartość wykonywana jest kopia bieżącego argumentu-obiektu, następuje wywołanie dodanego przez kompilator konstruktora kopiującego. Ten konstruktor kopiuje pola danych bieżącego argumentu do odpowiednich pól danych jego lokalnej kopii — obiektu będącego parametrem formalnym funkcji. Gdy kopiowany jest wskaźnik `str`, wskaźnik w obiekcie stanowiącym parametr formalny (lokalnej kopii) otrzymuje skopiowaną zawartość wskaźnika `str` (adres) z bieżącego argumentu. Ten skopiowany wskaźnik wskazuje adres na stercie, gdzie przechowywana jest tablica znakowa reprezentowana przez obiekt będący bieżącym argumentem funkcji.

W efekcie wskaźniki w obu tych obiektach — bieżącym argumencie i jego lokalnej kopii — wskazują tę samą sekcję pamięci na stercie, a każdy z tych obiektów uważa, że ma tę pamięć do wyłącznego użytku.

Tę sytuację próbowano przedstawić schematycznie na rysunku 11.9. W istocie to, co dotąd zostało powiedziane, nie zmienia działania funkcji operatorowej dokonującej przeciążenia operatora (na razie). To dlatego wszystko to, co powiedziano do tej pory, było samą prawdą i tylko prawdą.

Rysunek 11.9, który mówi całą prawdę, zawiera (dodatkowo obejmuje) lokalny obiekt `s`, którego pola danych są inicjowane poprzez skopiowanie pól danych bieżącego argumentu funkcji — obiektu `v`. Rysunek 11.9a pokazuje, że ten lokalny obiekt `s` oraz bieżący argument `v` — odwołują się do tej samej sekcji pamięci na stercie. Rysunek 11.9b pokazuje, że po przyporządkowaniu i zainicjowaniu nowej pamięci na stercie ta nowa pamięć zastąpiła starą w docelowym obiekcie komunikatu (`u`), a lokalny obiekt `s` oraz bieżący argument `v` nadal odwołują się do tej samej sekcji pamięci na stercie.



Rysunek 11.9. *Schemat pamięci przy przekazaniu przez wartość obiektu klasy String*

Cała prawda powinna jeszcze obejmować zakończenie wykonania tej funkcji. Gdy wykonanie kodu funkcji dochodzi do zamykającego nawiasu klamrowego (to także koniec zakresu widoczności nazw w obrębie funkcji), lokalna kopia obiektu (`String s`) jest usuwana. Z punktu widzenia konwencjonalnej intuicji programistycznej oznacza to, że pamięć zajmowana przez taki lokalny obiekt (wskaźnik i liczba całkowita w tym przypadku) powinna zniknąć (zostać automatycznie zwolniona i zwrócona na stos). Ale w C++ nie ma czegoś takiego, jak skasowanie obiektu — i już. Każde usunięcie obiektu musi zostać poprzedzone wywołaniem destruktora.

Destruktor, gdy zostaje wywołany, czyni to, co wynika z kodu destruktora. Zwalnia i zwraca na stertę (jako wolny i dostępny do dalszego użytkowania) ten segment pamięci, który wskazywał wskaźnik `str` znajdujący się w usuwanym obiekcie.

```
String::~String() { delete [] str; }
// zwrot pamięci wskazywanej przez wskaźnik str
```

Rysunek 11.9c przedstawia stan lokalnego obiektu `s` i bieżącego argumentu `v` po wywołaniu destruktora, ale zanim lokalny obiekt zostanie usunięty. Rysunek pokazuje, że i lokalny obiekt, i bieżący argument straciły przyporządkowaną im pamięć na stercie (pamięć wskazywana przez wskaźnik `str` została zwolniona). To działanie oczywiście nie ma wpływu na stan obiektu docelowego `u`, ponieważ obiekt docelowy nie jest usuwany. Gdy zakończy się wykonanie funkcji przeciążającej operator, obiekt docelowy pozostanie w dokładnie tym samym stanie, co podczas poprzedniej dyskusji, odzwierciedlonej schematycznie na rysunku 11.8. Taki kod klienta zwraca poprawne rezultaty.

```
String u("Hi "); String v("there!");
u += v;
cout << " u = " << u.show() <<endl;        // komunikat: "Hi there!"
```

Tym niemniej pamięć zwolniona i zwrócona do systemu przez destruktor, gdy usuwany był formalny parametr — obiekt `s`, nie należała do obiektu docelowego. Należała (i nadal powinna należeć) do bieżącego argumentu funkcji, czyli do obiektu `v` zdefiniowanego w przestrzeni klienta. Po wywołaniu tej funkcji obiekt klienta, który był użyty jako bieżący argument przekazany przez wartość — został pozbawiony przydzielonej mu uprzednio dynamicznie „jego" pamięci na stercie. Jeśli kod klienta po wywołaniu tej funkcji spróbuje ponownie użyć tego obiektu — spowoduje to wystąpienie błędu.

```
String u("Hi "); String v("there!");
cout << " u = " << u.show() << endl;       // wydruk "Hi "
cout << " v = " << v.show() << endl;       // wydruk "there!"
u += v;
cout << " u = " << u.show() << endl;       // wydruk "Hi there!"
cout << " v = " << v.show() << endl;       // przypadkowe "śmieci"
```

Nie wygląda szczególnie elegancko ani mądrze powtórne kontrolowanie zawartości obiektu `v`, która była przed chwilą drukowana; sam obiekt całkiem niedawno był używany jako rwartość w wywołaniu funkcji operatorowej `operator+=()`. Następuje to tylko dlatego, że dokładnie wiemy, że tu właśnie występuje problem z taką implementacją. Jest jasne, że obiekt powinien reprezentować dokładnie tę samą zawartość, którą przed chwilą reprezentował, uczestnicząc w wyrażeniu `u += v`. Tak podpowiada intuicja programisty przyzwyczajonego do konwencjonalnego programowania. W większości przypadków w C++ ta intuicja znajdzie potwierdzenie, ale nie zawsze, i możliwie jak najszybciej powinniśmy rozwinąć u siebie inną intuicję. To wszystko zostało tu powiedziane, ponieważ ten niewinnie wyglądający kod klienta może

doprowadzić do sytuacji, gdy tekst reprezentowany przez obiekt v będzie absolutnie przypadkowy, a wszelka próba użycia tego obiektu przy założeniu, że jego stan pozostaje niezmienny, jest zwykłą lekkomyślnością.

No i jak ci się to podoba? Programowanie w C++ nie pozwala się nudzić. Niemniej jednak programista piszący w C++ musi rozumieć, co dzieje się w sposób niejawny, „pod stołem", nawet w takim prostym programie, jak ten ostatni przykładowy fragment kodu.

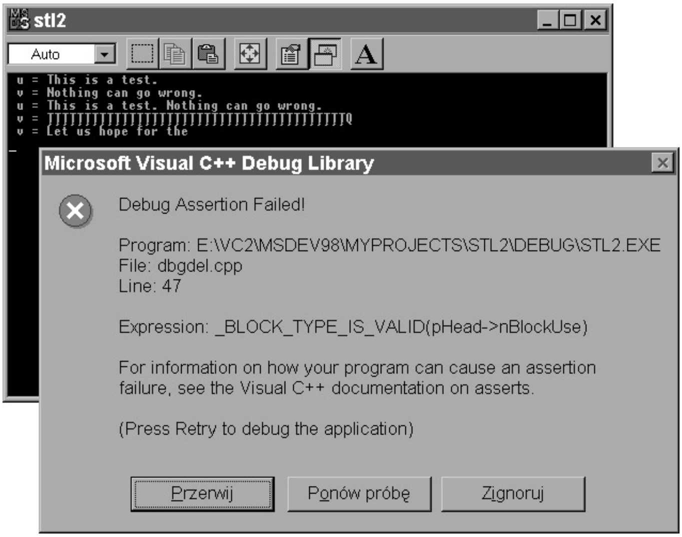
To jeszcze nie koniec tej opowieści. Takie efekty występują przy jeszcze jednym zamykającym nawiasie klamrowym — zamykającym zakres widoczności nazw. Zawsze zwracaj uwagę na nawiasy klamrowe ograniczające zakresy widoczności (dostępności). Powodują one wykonanie znaczącej części pracy. Gdy kod klienta dochodzi do nawiasu klamrowego zamykającego jego przestrzeń widoczności nazw i zamierza zakończyć swoje działanie, dla wszystkich lokalnych obiektów są wywoływane destruktory, włącznie z nieszczęsnym obiektem v, który był używany przy wywoływaniu funkcji operatorowej i tuż przed zakończeniem tej funkcji (powrotem z funkcji) został pozbawiony swojej dynamicznie przydzielonej pamięci. Destruktor próbuje zwolnić pamięć wskazywaną przez zawarty w tym obiekcie wskaźnik str, jednakże ta pamięć na stercie została już uprzednio uznana za nieważną i zwrócona do systemu. Gdybyśmy projektowali język programowania, moglibyśmy wprowadzić jakieś „no op" (NOP — ang. *no operation* — instrukcja nie powodująca żadnego działania). Ale tu nam się nie poszczęściło. Nie z C++ te numery, Brunner. W C++ powtórne użycie operatora delete w stosunku do tego samego wskaźnika jest zabronione. To jest błąd.

Niestety, stwierdzenie „to jest błąd" nie oznacza, że kompilator powstrzyma się od kompilacji i wydrukuje komunikat o błędzie składniowym tak, byśmy mogli to skorygować. Projektant kompilatora nie ponosi odpowiedzialności za śledzenie wykonania kodu i informowanie nas, że popełniliśmy błąd. Taki kod jest składniowo poprawny. Nie oznacza to także, że taki program skompiluje się, uruchomi, wykona i będzie wykazywać w sposób powtarzalny nieprawidłowe rezultaty. To znaczy tylko tyle, że rezultaty działania takiego kodu okażą się nieokreślone. W istocie rezultaty te są zależne od platformy uruchomieniowej, od tego, w jaki sposób aplikacja okaże się zależna od środowiska operacyjnego. System może się zawiesić, program może zadziałać w sposób nieprawidłowy (znienacka), może też przez pewien czas działać całkowicie poprawnie (do czasu).

Na listingu 11.3 pokazano kompletny program zawierający implementację tej wadliwej konstrukcji. Wyjściowy wydruk tego programu na monitorze autora przedstawiono na rysunku 11.10.

Listing 11.3. *Przeciążenie operatora konkatenacji z obiektem — parametrem przekazywanym poprzez wartość*

```
#include <iostream>
using namespace std;
class String
{
char *str;                        // dynamiczne zapamiętywanie tablicy typu char
int len;
public:
String (int length=0);            // konstruktor konwersji / domyślny
String(const char*);              // konstruktor konwersji
~String ();                       // przydział dynamicznej pamięci
void operator += (const String); // konkatenacja innego obiektu String
```



Rysunek 11.10. *Wydruk wyjściowy programu z listingu 11.3*

```
void modify(const char*);        // zmiana zawartości tablicy znakowej
const char* show() const;        // zwrot wskaźnika do tablicy
} ;
String::String(int length)
{ len = length;
str = new char[len+1];
if (str==NULL) exit(1);
str[0] = 0; }                    // pusty łańcuch o długości zerowej – to jest OK
String::String(const char* s)
{ len = strlen(s);               // pomiar długości tekstu wejściowego
str = new char[len+1];           // przydział wystarczającej ilości miejsca na stercie
if (str==NULL) exit(1);          // sprawdź, czy się powiodło
strcpy(str,s); }                 // kopiuj wejściowy tekst na stertę
String::~String()
{ delete str; }                  // zwrot pamięci na stercie (nie wskaźnika!)
void String::operator += (const String s)        // przez wartość
{ len = strlen(str) + strlen(s.str);             // całkowita długość
char *p = new char[len + 1];     // przydziel wystarczającą ilość miejsca
if (p==NULL) exit(1);            // sprawdź, czy się powiodło
strcpy(p,str);                   // kopiuj pierwszą część wynikowego tekstu
strcat(p,s.str);                 // dodaj drugą część tekstu
delete str;                      // ważna czynność
str = p; }                       // dopiero teraz p może zniknąć
const char* String::show() const                 // chroń dane przed zmianą
{ return str; }
void String::modify(const char a[])              // tu bez zarządzania pamięcią
{ strncpy(str,a,len-1);          // ochrona przed przepełnieniem
str[len-1] = 0; }                // poprawne zakończenie łańcucha znaków

int main()
{
String u("This is a test. ");
// po spolszczeniu: String u("To jest test. ");
String v("Nothing can go wrong.");
// po spolszczeniu: String v("Nic nie może pójść źle");
cout << " u = " << u.show() << endl;             // wynik jest OK
```

```
cout << " v = " << v.show() << endl;            // wynik jest OK
u += v; // u.operator+=(v);
cout << " u = " << u.show() << endl;            // wynik jest OK
cout << " v = " << v.show() << endl;            // tu wynik nie będzie OK
v.modify("Let us hope for the best.");          // uszkodzenie pamięci
// po spolszczeniu: String v("Nasza nadzieja w 'b'"); <- uszkodzenie
cout << " v = " << v.show() << endl;            // ????
return 0;
}
```

Zauważ, że te wszystkie okropne rzeczy dzieją się w chwili zakończenia działania funkcji. Pierwsze nieprzyjemne zdarzenie miało miejsce wtedy, gdy przeciążająca funkcja operatorowa `operator+=()` zbliżała się ku końcowi i musiało (zgodnie z zasadami) nastąpić wywołanie destruktora wobec formalnego parametru tej funkcji. Wtedy to bieżący argument tej funkcji, obiekt `v`, został pozbawiony przydzielonej mu na stercie dynamicznej pamięci. Drugie niemiłe zdarzenie miało miejsce, gdy funkcja-klient, `main()` zbliżała się ku końcowi i obiekt `v` musiał zostać usunięty, ponieważ kończył się zakres przestrzeni widoczności jego nazwy (obiekt wychodził poza zakres — ang. *out of scope*). Jego pamięć na stercie została wtedy zwolniona i zwrócona po raz wtóry.

W istocie, w C++ błędem jest powtórne zwracanie dynamicznej pamięci poprzez ponowne użycie operatora `delete` wobec tego samego niezerowego wskaźnika. Jeśli wskaźnik zawiera `NULL`, to nie jest błąd, lecz operacja pusta (ang. *no operation*). Niektórzy programiści próbują rozwiązać ten problem poprzez przypisanie wskaźnikowi do pamięci na stercie wartości `NULL` w obrębie destruktora[4].

```
String::~String()
{
delete str;                 // zwrot pamięci na stercie
str = 0;                    // ustaw na null, by uniknąć powtórnego zwrotu
}
```

To sympatycznie wyglądający pomysł, ale nie działa zgodnie z ich intencjami. Taki wskaźnik, który został ustawiony na zero, należy do obiektu, który to obiekt będzie usuwany w ciągu najbliższych mikrosekund. Nadal istnieje drugi wskaźnik, który wskazuje ten sam adres pamięci i mógłby zostać wyzerowany, ale nie jest dostępny dla takiego destruktora, wykonującego się przecież w odniesieniu do innego obiektu. Poza tym, nawet gdyby to zadziałało, stanowiłoby to tylko środek zapobiegający wykonaniu błędnej instrukcji, nie powodując przecież przywrócenia pamięci, która została omyłkowo skasowana.

Jak „stąd" przejść „tam"?

Czy autor przestraszył czytelnika? Jeśli tak, to taka właśnie była intencja. Jeśli nie, nie szkodzi, pamiętaj jednak zawsze, by zatroszczyć się o dynamiczne zarządzanie pamięcią w swoich programach. Nawet jeśli na twoim komputerze programy działają prawidłowo, to jeszcze nie jest oczywisty dowód, że dany program jest poprawny (dodajmy to do listy naszych zasad testowania programów).

[4] Taki wskaźnik przestaje wskazywać cokolwiek — *przyp. tłum.*

Program może działać bez żadnych zgrzytów miesiącami i latami, a następnie, np. po zainstalowaniu w systemie jakiejś innej, zupełnie z nim nie związanej aplikacji bądź po aktualizacji systemu operacyjnego do nowszej wersji Windows™ zmieni się sposób wykorzystywania pamięci i nasz program doprowadzi do katastrofy. Taki program może także dawać nieprawidłowe rezultaty, które nie zostaną dostrzeżone, ponieważ działał przecież poprawnie przez wiele miesięcy, a nawet przez wiele lat. Co się wtedy dzieje? Czy przeklinać Microsoft, bo właśnie zaktualizowaliśmy system operacyjny? Przecież to nie jest wina Microsoftu. To błąd programisty piszącego w C++, który popełnił grzech zaniechania, nie postawiwszy jednego znaku ampersand (&) w interfejsie funkcji dokonującej przeciążenia operatora `operator+=()`.

Oto jak powinna wyglądać ta funkcja. Jej obiekt-parametr nie jest przekazywany poprzez wartość, lecz poprzez referencję.

```
void String::operator += (const String &s)      // referencja do parametru
{
len = strlen(str) + strlen(s.str);              // całkowita długość
char *p = new char[len + 1];                    // przydziel wystarczającą ilość miejsca
if (p==NULL) exit(1);                           // sprawdź, czy się powiodło
strcpy(p,str);                                  // kopiuj pierwszą część
strcat(p,s.str);                                // kopiuj drugą część
delete str;                                     // bardzo ważna czynność
str = p; }                                      // teraz p może zniknąć
```

Na rysunku 11.11 pokazano wydruk wyjściowy programu z listingu 11.3 z funkcją operatora konkatenacji przy przekazaniu jej parametru poprzez referencję.

Rysunek 11.11. *Wydruk wyjściowy programu z listingu 11.3 z funkcją przeciążającą operator konkatenacji, w której parametr został przekazany poprzez referencję*

```
u = This is a test.
v = Nothing can go wrong.
u = This is a test. Nothing can go wrong.
v = Nothing can go wrong.
v = Let us hope for the b

Press any key to continue
```

Należy zdecydowanie uruchomić ten program, poeksperymentować z nim, by dokładnie zrozumieć te zagadnienia, które mogą sprawiać problemy. Nie ulegaj pokusie przekazywania obiektów jako parametrów poprzez wartość, chyba że jest to absolutnie niezbędne.

To prawdziwa okropność, że dodanie bądź usunięcie jednego tylko znaku w kodzie źródłowym (ampersanda — &) może zmienić zachowanie się programu w tak dramatycznym stopniu. Zwróć uwagę, że obie wersje kodu są ze składniowego punktu widzenia poprawne. Kompilator nie uprzedzi nas, że jest pewien problem, którym należałoby się martwić.

Przekazywanie obiektu jako parametru poprzez wartość przypomina kierowanie czołgiem. Zawsze dojedziemy tam, dokąd chcemy, ale po drodze możemy całkiem niezamierzenie dokonać wielu zniszczeń. Jak już powiedziano wcześniej, opieraj się pokusie przekazywania obiektów poprzez wartość, chyba że jest to absolutnie konieczne.

> Nie przekazuj obiektów do funkcji poprzez wartość. Jeśli obiekty zawierają wewnątrz wskaźniki i dynamicznie zarządzają pamięcią na stercie, nie można nawet myśleć o przekazaniu takich obiektów do funkcji poprzez wartość. Przekazuj obiekty do funkcji poprzez referencje i nie zapominaj o użyciu modyfikatora `const`, jeśli funkcja nie modyfikuje stanu obiektu-parametru i (lub) stanu docelowego obiektu komunikatu.

Więcej o konstruowaniu kopii obiektów

Popatrzmy wstecz, z pewnej perspektywy na tę sytuację. Istota problemu omawianego w poprzednim podrozdziale sprowadza się do kopiowania obiektu, który jako pola danych zawiera wskaźniki wskazujące segmenty pamięci na stercie.

Każdy taki obiekt powinien wskazywać obszar pamięci, który został przydzielony wyłącznie dla niego. Na przykład klasa `String` zawiera wskaźnik, który wskazuje segment pamięci na stercie zawierający tablicę znakową związaną z konkretnym obiektem klasy `String`.

Gdy pola danych zawarte w jednym obiekcie są kopiowane na pola danych innego obiektu, odpowiednie wskaźniki w obu tych obiektach będą mieć taką samą zawartość, będą zatem wskazywać ten sam obszar pamięci na stercie. Takie obiekty mogą być usuwane (kończyć swoje życie) w różnych momentach. Na przykład parametr formalny funkcji z listingu 11.3 znika, gdy ta funkcja zakończy swoje działanie, a jej rzeczywisty argument pozostaje przy życiu w przestrzeni klienta — w obrębie funkcji `main()`. Gdy obiekt ma przestać istnieć, jego destruktor zwalnia pamięć wskazywaną przez wskaźnik (czasem wskaźniki) zawarte w takim obiekcie. W ten sposób drugi spośród tych obiektów, nadal żyjący, po cichu traci związane z nim dane zapamiętane na stercie. Wszelka próba użycia potem takiego obiektu, który utracił swoje dane na stercie, jest niepożądana. To jest po prostu błąd.

Jeśli taka pamięć, zwrócona na stertę, nie zostanie natychmiast powtórnie użyta do innych celów, taki obiekt-fantom może zachowywać się tak, jakby ta zwrócona pamięć nadal była zastrzeżona dla niego. Nasze testy mogą utwierdzić nas w przekonaniu, że program jest poprawny.

Gdy znika drugi spośród tych obiektów, następuje wywołanie jego destruktora. Zwróć uwagę, że nie użyto tu sformułowania „destruktor jest wywoływany ponownie". Ten destruktor był wywoływany wcześniej, ale wobec innego obiektu (formalnego parametru funkcji), tego, który został już usunięty. Teraz ten sam destruktor jest wywoływany wobec drugiego spośród tych obiektów (rzeczywistego argumentu funkcji) i próbuje zwolnić i zwrócić do systemu ten sam segment pamięci na stercie. W C++ powoduje to powstanie sytuacji błędnej i zachowanie programu jest tu nieprzewidywalne. To taka grzeczna forma wypowiedzenia myśli, że w tym momencie program może zrobić wszystko, co mu się spodoba.

Sposób na zachowanie integralności programu

Jest wiele środków zapobiegawczych, których można użyć, by zapobiec problemom pojawiającym się wtedy, gdy obiekty z dynamicznie przyporządkowaną pamięcią są przekazywane jako parametry poprzez wartość.

Jednym ze sposobów jest wyeliminowanie destruktora, który zwalniałby i zwracał zbędną już pamięć na stercie do systemu. To nie jest ani dobre rozwiązanie na stałe, ani dobre rozwiązanie w ogóle. Możemy zdecydować, że zastosujemy je jako rozwiązanie tymczasowe, gdy program załamie się i musimy go uruchomić w celu przeprowadzenia sesji diagnostycznej z debugerem. Takie wyłączenie destruktora pozwoli na wykonanie programu od początku do końca.

Innym środkiem zapobiegawczym jest zastosowanie we wnętrzu obiektów tablic znakowych o stałych rozmiarach zamiast dynamicznego przyporządkowania pamięci. To nie jest eleganckie rozwiązanie, ale można je zastosować, jeśli rozmiar takiej tablicy zostanie wybrany szczodrze (z zapasem). Jest to szczególnie słuszne w przypadku tych programów, które posługują się stosunkowo niewielką liczbą obiektów i niezwykle rzadko muszą dokonywać obcinania danych, które nie zmieszczą się w tablicy o stałej wielkości, a poza tym jeśli jest to akceptowalne z punktu widzenia integralności danej aplikacji.

Odnośnie samego przekazywania parametrów, najlepszym środkiem zapobiegawczym jest tu przekazanie obiektów do funkcji jako parametrów poprzez referencje, a nie poprzez wartość. Eliminuje to problem tworzony przez kopiowanie obiektu. Przyspiesza to także wykonanie programu poprzez wyeliminowanie potrzeby tworzenia i usuwania tymczasowych obiektów, wywoływania konstruktorów i destruktorów.

Niestety, to rozwiązanie nie jest uniwersalne. Są przypadki kopiowania zawartości jednego obiektu do innego obiektu, które nie są związane z przekazywaniem parametrów do funkcji, kiedy to takie rozwiązanie nie może zostać zastosowane. Bywają przypadki, gdy jeden obiekt jest inicjowany przy użyciu innego obiektu tej samej klasy. Rozważmy następujący fragment kodu, w którym następuje przekazanie parametrów do funkcji `operator+=()` poprzez referencję.

```
// dla zgodności ze schematami pamięci – teksty oryginalne
String u("This is a test. "), v("Nothing can go wrong.");
// spolszczone: u("To jest test. "), v("Nic nie może pójść źle.");
cout << " u = " << u.show() << endl;        // wynik jest OK
cout << " v = " << v.show() << endl;        // wynik jest OK
u += v;                                     // u.operator+=(v); by reference
cout << " u = " << u.show() << endl;        // wynik jest OK
cout << " v = " << v.show() << endl;        // OK – poprzez referencję
v.modify("Let us hope for the best.");      // bez uszkodzenia pamięci
// "Nasza nadzieja w 'b'"
String t = v;                               // inicjowanie obiektu
cout << " t = " << t.show() << endl;        // wynik jest OK
t.modify("Nothing can go wrong.");          // modyfikuje i t, i v
cout << " t = " << t.show() << endl;        // wynik jest OK
cout << " v = " << v.show() << endl;        // v także zmodyfikowany
```

Ten kod tworzy dwa obiekty klasy `String` — `u` i `v`, inicjuje je za pomocą konstruktora konwersji i dokonuje konkatenacji reprezentowanych przez te obiekty łańcuchów znakowych. Skoro obiekt `v` jest przekazywany jako argument do funkcji poprzez referencję, nie ma tu zakłócenia działania pamięci i obiekt `v` zachowuje przyporządkowaną mu pamięć na stercie. Jeśli zmodyfikujemy obiekt `v`, zmieni się tylko jego zawartość, natomiast zawartość obiektu `u` pozostanie bez zmian. Następnie utworzymy jeszcze jeden obiekt `t` klasy `String`, który zainicjujemy poprzez skopiowanie dotychczasowej zawartości obiektu `v`. Gdy modyfikujemy zawartość obiektu `t`, intuicyjnie zakładamy, że zawartość obiektu `v` pozostanie bez zmian. Na rysunku 11.12 pokazano oczekiwane rezultaty wykonania podanego fragmentu kodu.

Rysunek 11.12. *Oczekiwany (a nie rzeczywisty) wydruk wyjściowy podanego fragmentu kodu klienta*

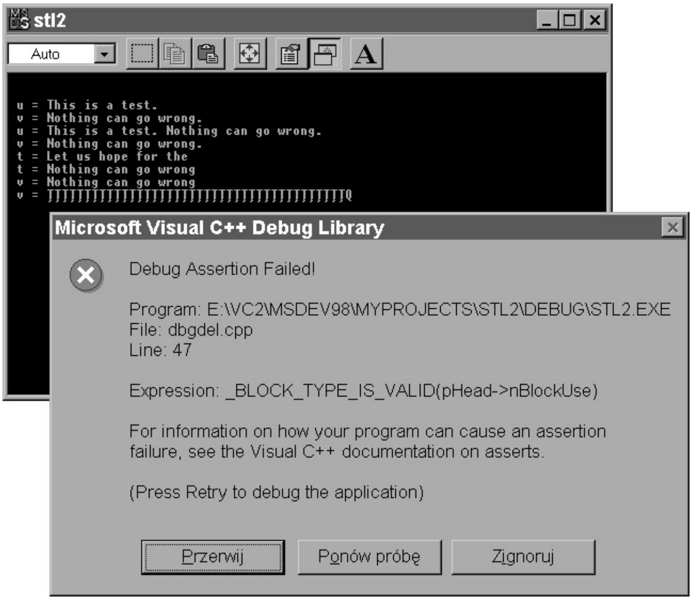
```
u = This is a test.
v = Nothing can go wrong.
u = This is a test. Nothing can go wrong.
v = Nothing can go wrong.
t = Let us hope for the b
t = Nothing can go wrong.
v = Nothing can go wrong.
```

W prawdziwym, realnym życiu programisty nie wszystko jednak przebiega zgodnie z naszymi oczekiwaniami. Na listingu 11.4 pokazano kod klasy `String` (z przekazywaniem parametru do funkcji `operator+=()` poprzez referencję) i kod klienta zawierający podany powyżej fragment kodu. Fragment ten na listingu zmodyfikowano w taki sposób, że obiekt `t` został utworzony w zagnieżdżonym zakresie *widoczności nazw* (ang. *nested scope*). Gdy ten zagnieżdżony segment kodu będzie się zbliżał do zakończenia i obiekt `t` zostanie usunięty, możemy zweryfikować stan obiektu `v` i skontrolować jego integralność. Na rysunku 11.13 pokazano rzeczywiste rezultaty wykonania programu z listingu 11.4.

Listing 11.4. *Inicjowanie jednego obiektu za pomocą danych z innego obiektu*

```
#include <iostream>
using namespace std;
class String
{
char *str;                          // dynamicznie zapamiętana tablica znakowa
int len;
public:
String (int length=0);              // konstruktor konwersji i domyślny
String(const char*);                // konstruktor konwersji
~String ();                         // przydziel dynamicznie pamięć
void operator += (const String&);   // konkatenacja innego obiektu
void modify(const char*);           // zmiana zawartości tablicy znakowej
const char* show() const;           // zwrot wskaźnika do tablicy
} ;
String::String(int length)
{ len = length;
str = new char[len+1];
if (str==NULL) exit(1);
str[0] = 0; }                       // pusty łańcuch o zerowej długości też jest OK
String::String(const char* s)
{ len = strlen(s);                  // pomiar długości tekstu wejściowego
str = new char[len+1];              // przydział pamięci na stercie
if (str==NULL) exit(1);             // sprawdź, czy się powiodło
strcpy(str,s); }                    // skopiuj wejściowy tekst na stertę
String::~String()
{ delete str; }                     // zwrot pamięci na stercie (a nie wskaźnika!)
void String::operator += (const String& s)          // parametr jako referencja
{ len = strlen(str) + strlen(s.str);                // całkowita długość
char* p = new char[len + 1];        // przydziel wystarczającą ilość miejsca
if (p==NULL) exit(1);               // sprawdź, czy się powiodło
strcpy(p,str);                      // kopiuj pierwszą część rezultatu
strcat(p,s.str);                    // dodaj drugą część rezultatu
delete str;                         // ważny krok
str = p; }                          // teraz tymczasowy wskaźnik p może zniknąć
const char* String::show() const    // chroń dane przed zmianami
{ return str; }
void String::modify(const char a[])                 // bez zarządzania pamięcią
{ strncpy(str,a,len-1);                             // ochrona przed przepełnieniem
str[len-1] = 0; }                                   // poprawne zakończenie tekstu
int main()
{ cout << endl << endl;
String u("This is a test. ");
String v("Nothing can go wrong.");
cout << " u = " << u.show() << endl;                // wynik jest OK
cout << " v = " << v.show() << endl;                // wynik jest OK
```

Rysunek 11.13.
Wyjściowy wydruk programu z listingu 11.4




```
u += v;                                        // to odpowiada: u.operator+=(s);
cout << " u = " << u.show() << endl;           // wynik jest OK
cout << " v = " << v.show() << endl;           // OK — poprzez referencję
v.modify("Let us hope for the best.");         // bez uszkodzenia pamięci
{ String t = v;                                // inicjowanie
cout << " t = " << t.show() << endl;           // wynik jest OK
t.modify("Nothing can go wrong.");             // zmieniamy obydwa — t oraz v
cout << " t = " << t.show() << endl;           // wynik jest OK
cout << " v = " << v.show() << endl; }         // v także jest zmodyfikowany
cout << " v = " << v.show() << endl;           // t znika, v pozbawiony pamięci
return 0;
}
```

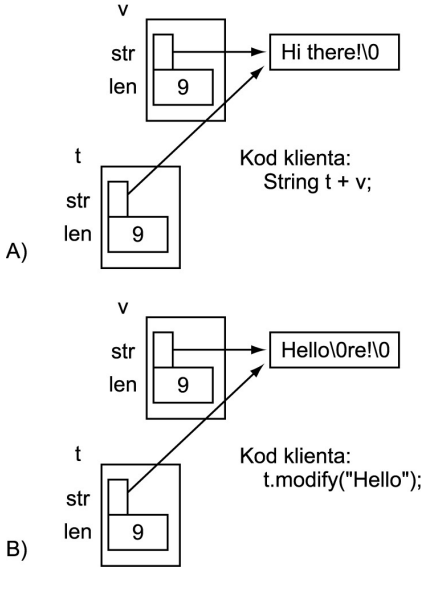
Gdy tworzony jest obiekt `t` klasy `String` (a jest on tworzony na stosie, ponieważ `t` jest lokalną zmienną automatyczną), zostaje mu przyporządkowana w pamięci przestrzeń wystarczająca dla zapamiętania wskaźnika typu `char*` i liczby całkowitej. Następnie wywoływany jest konstruktor. Po stronie kodu klienta widzimy znak operatora przypisania `=` , ale to nie oznacza operacji przypisania — ten symbol oznacza tu zainicjowanie obiektu. Jak już wspominano wcześniej, nie ulega żadnej wątpliwości, że po utworzeniu obiektu w pamięci[5] zawsze nastąpi wywołanie konstruktora. Pytanie tylko, który spośród konstruktorów zostanie wywołany w danym, konkretnym przypadku. Odpowiedź brzmi: to zależy od danych, które dostarczy kod klienta wtedy, gdy tworzony jest dany obiekt. Na listingu 11.4 funkcja-klient, `main()`, dostarcza konstruktorowi bieżący, rzeczywisty argument — istniejący już obiekt `v`. Skoro tak, wywołany tu zostanie konstruktor z jednym parametrem, będącym obiektem takiego samego typu, jak klasa, do której należy konstruktor — w naszym przypadku typu `String`.

[5] Tj. przyporządkowaniu pamięci dla samego obiektu — *przyp. tłum.*

Jak nazywa się taki konstruktor z jednym parametrem tego samego typu, co własna klasa konstruktora? Jak zapewne pamiętasz z rozdziału 9. „Klasy w C++ jako jednostki modularyzacji", to jest konstruktor kopiujący, ponieważ jego działanie polega na skopiowaniu danych z jednego obiektu do innego obiektu; ale klasa `String` nie zawiera konstruktora kopiującego. Czy to oznacza, że próba wywołania takiego nieistniejącego konstruktora kopiującego spowoduje komunikat o błędzie składniowym? Nie. Kompilator wygeneruje wywołanie domyślnego konstruktora kopiującego, który sam automatycznie dodał do specyfikacji klasy. Kompilator sam dodaje taki konstruktor i ten sam kompilator generuje jego wywołanie. Ten konstruktor skopiuje pola swojego obiektu-argumentu do nowego obiektu, który właśnie został utworzony. Dla klasy `String` taki dodawany automatycznie przez kompilator konstruktor kopiujący wygląda następująco:

```
// konstruktor kopiujący dodawany automatycznie przez kompilator
String::String(const String& s)
{ len = s.len;              // kopiuj długość obiektu tekstowego
str = s.str; }              // kopiuj wskaźnik do obiektu tekstowego
```

Na rysunku 11.14 pokazano, jak działa ten konstruktor. Gdy utworzony zostaje obiekt `t` klasy `String`, jego pole `len` przybiera wartość 9, a jego wskaźnik `str` jest ustawiany tak, by wskazywał ten sam obszar pamięci na stercie, który wskazuje wskaźnik `str` należący do obiektu `v`.



Rysunek 11.14. *Diagram pamięci przy inicjowaniu jednego obiektu klasy String za pomocą danych zawartych w innym obiekcie tejże klasy*

Podobnie jak w poprzedniej opowieści o przekazywaniu parametrów, te dwa obiekty, `t` oraz `v`, mają jeden wspólny segment pamięci na stercie, a nie dwa różne. Ten obszar pamięci został wcześniej przydzielony obiektowi `v`, ale teraz stał się wspólny dla niego i obiektu `t`, a każdy z tych obiektów uważa, że ten obszar pamięci na stercie jest do jego wyłącznej dyspozycji (i zachowuje się tak, jakby tak było). Taka sytuacja jest jeszcze gorsza niż przy przekazaniu poprzez wartość. Przy przekazaniu poprzez wartość bieżący, rzeczywisty argument występuje w przestrzeni *nazw klienta* (ang. *client scope*), a parametr formalny — w przestrzeni *nazw serwera* (ang. *server scope*). W każdym, dowolnie wybranym momencie wykonania programu tylko jeden z tych obiektów może być widoczny i dostępny. W tym przypadku obydwa obiekty znajdują się w tej samej przestrzeni klienta i można się do nich odwoływać, co więcej, obydwa mogą zostać zmodyfikowane w tym samym zakresie widoczności nazw.

Skoro te dwa obiekty posługują się tym samym obszarem pamięci na stercie, z punktu widzenia kodu klienta obydwa te obiekty są traktowane jako synonimy. Tak więc gdy obiekt `t` zostaje zmodyfikowany przez kod klienta, obiekt `v` także zostaje zmodyfikowany. Czy jest to wyraźnie widoczne na rysunku 11.13? Z punktu widzenia powszechnej intuicji programistycznej nie ma żadnego racjonalnego powodu, by obiekt `v` miał być zmodyfikowany w kodzie klienta — tym niemniej tak się dzieje.

Zastanówmy się nad tym. To wydaje się dziwne tylko z punktu widzenia popularnej intuicji. Na kursach programowania dla początkujących autor często spotyka słuchaczy, którzy mają kłopoty z prostym kodem posługującym się liczbami całkowitymi:

```
int v = 10; int t = v; t = 20;          // jaka jest teraz wartość v?
```

Większości programistów wydaje się zupełnie oczywiste, że wartość `v` nie zmieni się po tym, jak zmodyfikowana została wartość `t`, ponieważ zmienne `t` oraz `v` zajmują różne obszary pamięci. Niektórzy jednak stwierdzają: „OK. Zrobiliśmy tu najpierw założenie, że zmienne `v` oraz `t` oznaczają to samo. Skoro teraz zmodyfikowaliśmy wartość `t`, nie ma się co dziwić, że zmienia się również wartość `v`".

I mają rację. Należy im się punkt. Jeśli zmienne są synonimami, zmodyfikowanie wartości jednej z nich jest widziane przez drugą. Jak zapewne pamiętasz, to całkiem powszechna sytuacja, gdy jedna z takiej pary zmiennych jest zwykłą zmienną, a druga jest referencją.

```
int v = 10; int& t = v; t = 20;          // a teraz, jaka jest wartość v?
```

W tym przykładzie typowa intuicja programisty po prostu nas zawiedzie. Może dlatego, że to logika nowicjusza. Poczyniliśmy założenie, że dwie zmienne, `t` oraz `v`, są takie same. I co? Teraz lekkie zdziwienie, że stan obiektu `v` zmienia się po zmianie stanu obiektu `t`? Teraz obiekt `v` zawiera 20. To jest logika, do której muszą przywyknąć wszyscy użytkownicy C++, i nowicjusze, i eksperci. Co więcej, muszą się z nią poczuć komfortowo.

Semantyka referencji i semantyka wartości

W istocie są dwa rodzaje powszechnej programistycznej intuicji, które odpowiadają dwu różnym koncepcjom informatyki, a są nazywane *semantyką wartości* (ang. *value semantics*) oraz *semantyką referencji* (ang. *reference semantics*). Słowo „semantyka" jest tu używane w znaczeniu kopiowania danych.

Bardziej powszechna intuicja programistyczna posługuje się semantyką wartości. Każdy obiekt (w nieco szerszym znaczeniu, tj. albo zmienna elementarnego typu wbudowanego, albo obiekt klasy zdefiniowanej przez programistę) posiada swoją odrębną lokalizację w pamięci (własny, odseparowany obszar). Przyrównanie dwóch takich obiektów jest rozumiane jako powtórzenie w innym miejscu pamięci takiego samego układu bitów. W C++ (jak i w większości pozostałych języków programowania) taka semantyka wartości jest używana i wobec zmiennych elementarnych typów wbudowanych, i wobec obiektów klasy zdefiniowanej przez programistę.

```
// semantyka wartości – wartość v będzie wynosić 10
int v = 10; int t = v; t = 20;
```

Oto dlaczego takie intuicyjne podejście jest bardziej powszechne. Z takiego punktu widzenia, gdy dwa obiekty mają tę samą wartość, mają dwa odrębne zbiory bitów, a zatem zmiana wartości jednego z tych obiektów nie może mieć wpływu na istniejący już zbiór bitów odnoszący się do drugiego z tych obiektów.

Inna, nieco mniej powszechna intuicja programistyczna, posługuje się semantyką referencji. Według takiego wyobrażenia, gdy obiektowi zostaje przypisana wartość, otrzymuje on referencję (lub wskaźnik) do tej wartości. Przyrównanie dwóch obiektów oznaczałoby zatem ustawienie ich referencji (lub wskaźników) tak, by odwoływały się do tego samego adresu w pamięci. Jeśli tablica znakowa wskazywana przez wskaźnik w jednym z tych obiektów się zmieni, drugi z tych obiektów automatycznie dostrzega taką zmianę, ponieważ obydwa wskaźniki wskazują tę samą lokalizację w pamięci. W C++ taka semantyka referencyjna jest stosowana dla wskaźników i referencji, przy przekazywaniu parametrów poprzez referencję lub poprzez wskaźnik, dla tablic i wobec połączonych poprzez wskaźniki struktur danych.

```
// semantyka referencyjna – wartość v będzie wynosić 20
int v = 10; int& t = v; t = 20;
```

Znowu zdziwienie? Że semantyka referencji jest mniej powszechna? Jest ona stosowana głównie z powodu dążenia do wyższej efektywności (np. pozwala wyeliminować kopiowanie obiektów przy przekazywaniu parametrów do funkcji). Czasem takie referencyjne działania przychodzą same — bez żadnego zaproszenia, jak w tym przypadku, a my powinniśmy być przygotowani na ich rozpoznanie i na właściwe wobec nich postępowanie. Programista piszący w C++ powinien zawsze pamiętać o różnicy pomiędzy semantyką wartości (kopiowania) a semantyką referencji (wskazania).

To jeszcze nie koniec kłopotów z programem z listingu 11.4. Gdy wykonanie programu dochodzi do zamykającego nawiasu klamrowego zagnieżdżonego bloku instrukcji (zatem i zagnieżdżonego zakresu widoczności nazw), obiekt `t` powinien zostać usunięty, ponieważ jest zdefiniowany tylko wewnątrz tego zakresu (to ulubiony temat autora odnośnie dyskusji i analiz zachowania się kodów). Obiekt `v` jest zdefiniowany w obrębie całego ciała funkcji `main()` i powinien być dostępny dla dalszego użytkowania. Na listingu 11.4 autor podejmuje próbę wydrukowania zawartości obiektu `v` pod koniec kodu funkcji `main()`. Zwróć uwagę, że ta instrukcja wyprowadzenia danych jest taka sama, jak poprzednia, a te dwie instrukcje rozdziela jedynie nawias klamrowy zamykający zagnieżdżony zakres widoczności nazw. Powierzchowne wrażenie jest takie, że nic przecież nie wydarzyło się pomiędzy tymi dwiema instrukcjami w kodzie klienta, zatem te dwie instrukcje powinny dać taki sam wydruk wyjściowy. Tak jednak nie jest. Jeszcze raz tradycyjna intuicja programistyczna okazuje się niewystarczająca do zrozumienia programu napisanego w C++ i musimy rozwinąć naszą intuicję, by pozwoliła nam na czytanie i zrozumienie takich fragmentów kodu, jak ten.

Jak widać na rysunku 11.14, pierwsza instrukcja wyprowadza czytelny i zrozumiały wydruk. Nie jest to dokładnie to, czego normalnie można by się było spodziewać, ale przynajmniej jest. Druga instrukcja wyprowadza śmiecie. Co się wydarzyło pomiędzy tymi dwiema instrukcjami? Gdy wykonanie programu doszło do nawiasu klamrowego zamykającego zakres widoczności zagnieżdżonego bloku, wobec lokalnego obiektu `t` zdefiniowanego w tym zagnieżdżonym bloku został wywołany destruktor klasy `String`. Jak widać z listingu 11.4 i z rysunku 11.14, ten destruktor, posługując się operatorem `delete`, zwolnił i zwrócił do systemu pamięć wskazywaną na stercie przez wskaźnik `str` należący do likwidowanego obiektu `t`. Ta sekcja dynamicznej pamięci w istocie była przyporządkowana obiektowi `v`,

ale system o tym zapomniał. System pamięta jedynie, że pamięć wskazywana przez wskaźnik `str` powinna zostać zwrócona do systemu zgodnie z kodem destruktora klasy `String`. Obiekt `v` został pozbawiony swojej dynamicznej pamięci, ale nikt o tym nie wie. Obiekt `v` jest formalnie nadal w zakresie widoczności i dostępności i wydaje się pozostawać w dobrym zdrowiu. To jednak tylko pozory. Ten obiekt nie może być już wykorzystywany do niczego użytecznego w kodzie klienta.

To przypomina przekazanie parametrów przez wartość i podobnie, jak poprzednio, to jeszcze nie koniec całej historii. Gdy wykonanie programu dochodzi do nawiasu klamrowego zamykającego kod funkcji `main()`, obiekt `v` powinien zniknąć, zgodnie z regułami widoczności nazw. Bezpośrednio przedtem wywoływany jest destruktor, który próbuje zwolnić i zwrócić do systemu wcześniej już zwróconą pamięć na stercie. Taki program jest nieprawidłowy. Może zrobić, co zechce (nastąpi załamanie się programu).

Konstruktor kopiujący definiowany przez programistę

Podsumowując krótko zagadnienie dynamicznego zarządzania pamięcią: ten problem ma tylko jedno rozwiązanie — konstruktor kopiujący zdefiniowany przez programistę. Taki konstruktor powinien przydzielić miejsce w pamięci na stercie dla docelowego obiektu komunikatu, podobnie jak funkcja operatorowa operatora konkatenacji, opisana w poprzednim podrozdziale. Oto algorytm działania takiego konstruktora:

1. Skopiuj długość tablicy znakowej reprezentowanej przez obiekt — parametr do pola `len` docelowego obiektu komunikatu.

2. Przyporządkuj pamięć na stercie; ustaw wskaźnik `str` w docelowym obiekcie komunikatu tak, by wskazywał początek tej pamięci na stercie.

3. Sprawdź, czy przydział pamięci na stercie się powiódł[6]. Jeśli w systemie zabrakło pamięci, zakończ działanie.

4. Kopiuj znaki reprezentowane przez obiekt — parametr do nowej, przydzielonej na stercie pamięci.

Oto konstruktor kopiujący zdefiniowany przez programistę, stanowiący rozwiązanie naszego problemu.

```
// konstruktor kopiujący definiowany przez programistę
String::String(const String& s)
{
len = s.len;                    // długość tekstu źródłowego
str = new char[len+1];          // żądanie odrębnej pamięci na stercie
if (str == NULL) exit(1);       // sprawdź, czy się powiodło
strcpy(str,s.str);              // kopiuj tekst źródłowy
}
```

[6] Tj. czy wskaźnik wskazuje cokolwiek — *przyp. tłum.*

Zwróćmy uwagę, że parametr `s` zostaje tu przekazany poprzez referencję. To jest referencja do obiektu stanowiącego bieżący argument dla danego wywołania konstruktora kopiującego. Przy przekazywaniu argumentu do funkcji nie następuje żadne kopiowanie pól danych argumentu. Przeciwnie, dynamiczny przydział pamięci na stercie dla obiektu docelowego następuje w obrębie kodu ciała tego konstruktora. Zawartość dynamicznej pamięci związanej z kopiowanym obiektem-argumentem jest kopiowana do dynamicznej pamięci przydzielonej obiektowi docelowemu.

Taka technika jest mniej efektywna niż kopiowanie pól obiektu, jak na listingu 11.4. Semantyka wartości jest powolniejsza od semantyki referencji, ponieważ operuje na wartościach, a nie na referencjach czy wskaźnikach. Z drugiej jednak strony, semantyka wartości jest bezpieczna. Przypomnijmy kod klienta, który spowodował wszystkie te problemy.

```
String t = v;
// teraz – bez problemu, jeśli użyty będzie konstruktor kopiujący
```

Po wykonaniu tego kodu (teraz, po wprowadzeniu konstruktora kopii) wskaźniki `str` należące do dwóch obiektów, `t` oraz `v`, wskazują dwa różne obszary pamięci na stercie.

> Jeśli pośród pól danych klasy znajduje się wskaźnik (wskaźniki), a obiekt danej klasy posługuje się dynamicznym zarządzaniem pamięcią na stercie, projektant takiej klasy powinien zdecydować, czy ta klasa wymaga zastosowania semantyki wartości, czy też semantyki referencyjnej. Jeśli potrzebuje semantyki wartości i jeśli inicjuje jeden obiekt, posługując się zawartością innego obiektu, powinien się upewnić, że taka klasa zawiera zdefiniowany przez programistę, własny konstruktor kopiujący.

Na listingu 11.5 pokazano program z listingu 11.4, w wersji w której klasa `String` definiuje własny konstruktor kopiujący i posługuje się semantyką wartości do zainicjowania obiektu. Wydruk wyjściowy tego programu przedstawiono na rysunku 11.15. Jak widać, problem integralności programu zniknął. Obiekty `t` oraz `v` klasy `String` nie są już wzajemnymi synonimami. Gdy obiekt `t` się zmienia, obiekt `v` pozostaje niezmienny. Gdy kończy się zagnieżdżony zakres widoczności nazw i znika obiekt `t`, obiekt `v` istnieje nadal i może nadal być używany w kodzie klienta bez jakichkolwiek trudności. Autor radzi prześledzić uważnie zawartość tego listingu i wydruku wyjściowego programu, by mieć pewność, że wyraźnie widzi się relacje pomiędzy tymi dwoma obiektami.

Listing 11.5. *Zastosowanie konstruktora kopiującego do zainicjowania jednego obiektu przy wykorzystaniu danych z innego obiektu*

```
#include <iostream>
using namespace std;
class String
{
char *str;                          // dynamicznie zapamiętana tablica znakowa
int len;
char* allocate(const char* s)       // prywatna funkcja (metoda)
{ char *p = new char[len+1];         // przydział pamięci na stercie dla obiektu
if (p==NULL) exit(1);               // jeśli się nie powiodło – koniec
strcpy(p,s);                        // kopiuj tekst na stertę
return p; }                         // zwróć wskaźnik do pamięci na stercie
public:
String (int length=0);              // konstruktor konwersji / domyślny
String(const char*);                // konstruktor konwersji
```

Rysunek 11.15.
Wydruk wyjściowy programu z listingu 11.5

```
u = This is a test.
v = Nothing can go wrong.
u = This is a test. Nothing can go wrong.
v = Nothing can go wrong.
t = Let us hope for the b
t = Nothing can go wrong.
v = Let us hope for the b
v = Let us hope for the b
```

```
String(const String& s);           // konstruktor kopiujący
~String ();                        // destruktor ma za zadanie zwolnić dynamiczną pamięć
void operator += (const String&);  // dołącz (dodaj) inny obiekt
void modify(const char*);          // zmień zawartość pamięci
const char* show() const;          // zwróć wskaźnik do tablicy
} ;

String::String(int length)
{ len = length;
str = allocate(""); }              // kopiuj pusty łańcuch na stertę

String::String(const char* s)
{ len = strlen(s);                 // pomiar długości wejściowego tekstu
str = allocate(s); }               // przydziel pamięć, skopiuj tekst

String::String(const String& s)    // konstruktor kopiujący
{ len = s.len;                     // pomiar długości wejściowego tekstu
str = allocate(s.str); }           // przydziel pamięć, skopiuj tekst

String::~String()
{ delete str; }                    // zwrot pamięci na stercie (nie wskaźnika!)

void String::operator += (const String& s)         // parametr jako referencja
{ len = strlen(str) + strlen(s.str);               // całkowita długość
char* p = new char[len + 1];       // przydziel wystarczającą ilość miejsca na stercie
if (p==NULL) exit(1);              // sprawdź, czy się powiodło
strcpy(p,str);                     // kopiuj pierwszą część rezultatu
strcat(p,s.str);                   // dodaj drugą część rezultatu
delete str;                        // ważna czynność
str = p; }                         // teraz wskaźnik p może zniknąć
const char* String::show() const   // chroń dane przed zmianami
{ return str; }

void String::modify(const char a[])                // bez zarządzania pamięcią
{ strncpy(str,a,len-1);                            // ochrona przed przepełnieniem
str[len-1] = 0; }                                  // poprawne zakończenie łańcucha String

int main()
{
cout << endl << endl;
String u("This is a test. ");                      // "To jest test."
String v("Nothing can go wrong.");                 // "Nic nie może pójść źle."
cout << " u = " << u.show() << endl;               // rezultat jest OK
cout << " v = " << v.show() << endl;               // rezultat jest OK
u += v; // u.operator+=(v);
cout << " u = " << u.show() << endl;               // rezultat jest OK
cout << " v = " << v.show() << endl;               // OK – poprzez referencję
// a tu: "Nasza nadzieja w 'b'":
v.modify("Let us hope for the best.");             // bez uszkodzenia pamięci
```

```
{ String t = v;                                    // wywołanie konstruktora kopii
cout << " t = " << t.show() << endl;              // OK – poprawny rezultat
t.modify("Nothing can go wrong.");                // zmiana tylko t
cout << " t = " << t.show() << endl;              // OK – poprawny rezultat
cout << " v = " << v.show() << endl; }            // v się nie zmienił
cout << " v = " << v.show() << endl;              // t znika, v nienaruszony
return 0;
}
```

Na listingu 11.5 klasa `String` ma trzy konstruktory, które w przybliżeniu robią to samo. Wszystkie przydzielają dynamicznie pamięć na stercie i inicjują jej zawartość. W pierwszym konstruktorze konwersji dane początkowe (inicjujące) to pusty łańcuch znaków (tylko znak terminatora — zero). W drugim konstruktorze konwersji dane inicjujące to tablica znakowa dostarczona przez kod klienta w formie bieżącego argumentu konstruktora. W konstruktorze kopiującym dane inicjujące to tablica znakowa reprezentowana przez obiekt, dostarczony jako argument przez kod klienta. Skoro taka tablica znakowa zostaje umieszczona na stercie, nie musi ona mieć własnej nazwy i można się do niej odwoływać przy użyciu wskaźnika `str` wskazującego tę tablicę. Skoro obiekt-parametr `s` należy do tej samej klasy `String`, co docelowy obiekt komunikatu, który podlega zainicjowaniu, konstruktor kopiujący ma prawo dostępu do prywatnego pola danych — wskaźnika `str` tego obiektu, posługując się jego pełną nazwą — `s.str`.

Jest zresztą naturalne, że różne konstruktory wykonują podobne algorytmy, ponieważ wynikowy obiekt powinien zawsze wyglądać niemal dokładnie tak samo, bez względu na to, który z tych konstruktorów był wywoływany wtedy, gdy tworzony był dany obiekt. Jeśli klasa zawiera jeden lub dwa konstruktory, sensowne jest po prostu dokładne powtórzenie kodu, „słowo w słowo". W miarę wzrostu liczby zastosowań takich wspólnych algorytmów (i wszystko wskazuje na to, że do końca jeszcze daleko), programiści często dokonują ich hermetyzacji poprzez umieszczenie ich wewnątrz prywatnych metod, które są następnie wywoływane przez różne funkcje (już często publiczne) należące do tej samej klasy. Taka funkcja z algorytmem powinna być prywatna, ponieważ kodu klienta nie interesują szczegóły obsługi pamięci przez obiekt. Takie szczegóły niskiego poziomu nie powinny zakłócać realizacji algorytmu kodu klienta ani zaskakiwać programisty tworzącego kod klienta. Taką prywatną funkcję można obejrzeć na listingu 11.5. Gdy ta funkcja kopiuje swój parametr do przydzielonej mu pamięci na stercie, wykorzystywany jest zamiast nazwy wskaźnik `p` wskazujący lokalizację w pamięci, ponieważ taka dynamicznie tworzona macierz nie ma własnej nazwy.

```
char* allocate(const char* s)      // prywatna metoda
{ char *p = new char[len+1];       // przydziel pamięć na stercie dla obiektu
if (p==NULL) exit(1);              // jeśli się nie powiodło – zakończ
strcpy(p,s);                       // kopiuj tekst na stertę
return p; }                        // zwróć wskaźnik do pamięci na stercie
```

Kod z listingu 11.5 demonstruje, jak pierwszy konstruktor konwersji przekazuje do funkcji `allocate()` (umieść na stercie) pusty łańcuch znaków, drugi konstruktor konwersji przekazuje do tejże funkcji własny argument, tablicę znakową, natomiast konstruktor kopii przekazuje do funkcji `allocate()` tablicę znakową wskazywaną poprzez wskaźnik `s.str` jego własnego argumentu — obiektu.

Gdy jeden obiekt inicjuje drugi obiekt, wywoływany jest konstruktor kopiujący. To nieuniknione. Jest tylko kwestia, jaki konstruktor zostanie wywołany. Jeśli klasa nie zawiera własnej, indywidualnej wersji konstruktora kopiującego, kompilator wygeneruje wywołanie automatycznie dodanego, domyślnego konstruktora kopiującego, który skopiuje pola danych

obiektu. Jeśli obiekt danej klasy nie wykorzystuje dynamicznej pamięci na stercie, to wystarczy. Jeśli obiekty wykorzystują indywidualne segmenty pamięci na stercie (semantyka wartości), użycie domyślnego konstruktora kopiującego dodawanego automatycznie przez kompilator stanowi jakby podłożenie miny pod integralność aplikacji. Aby uchronić integralność programu, klasa powinna zawierać implementację własnego konstruktora kopiującego, który zapewni docelowemu obiektowi przydział jego własnej pamięci na stercie.

W poprzednim zdaniu określenie „klasa powinna zawierać implementację" podkreśla normalne relacje klient-serwer pomiędzy różnymi segmentami kodu programu i podział na różne strefy koncentracji uwagi ze strony człowieka. Kod klienta wyraża swoje potrzeby, posługując się obiektami do realizacji celu aplikacji (w tym np. inicjując jeden obiekt za pomocą drugiego obiektu). Kod serwera obsługuje potrzeby kodu klienta poprzez zaimplementowanie stosownych metod, które są wywoływane z kodu klienta. Konstruktory są wywoływane w sposób niejawny, ale to w niczym nie zmienia relacji klient-serwer.

Jeśli aplikacja wymaga zastosowania semantyki wartości (kopii), klasy z dynamicznym zarządzaniem pamięcią mogą zostać zmuszone do zapewnienia konstruktorów kopiujących przeznaczonych do użycia w innych kontekstach, gdy jeden obiekt inicjuje inny obiekt. Jeden z takich możliwych kontekstów to przekazywanie obiektu jako parametru poprzez wartość. Jeśli tylko odpowiednia wersja konstruktora kopiującego znajduje się na swoim miejscu, nasza pierwsza wersja funkcji przeciążającej operator konkatenacji, `operator+=()` z listingu 11.3, jest absolutnie w porządku.

```
void String::operator += (const String s)       // poprzez wartość
{ len = strlen(str) + strlen(s.str);            // całkowita długość
char *p = new char[len + 1];                    // przydziel wystarczającą ilość miejsca
if (p==NULL) exit(1);                           // sprawdź, czy się powiodło
strcpy(p,str);                                  // kopiuj pierwszą część rezultatu
strcat(p,s.str);                                // dodaj drugą część rezultatu
delete str;                                     // bardzo ważna czynność
str = p; }                                      // teraz p może sobie znikać
```

Gdy wywoływana jest ta funkcja i tworzona jest kopia jej rzeczywistego, bieżącego argumentu, następuje wywołanie konstruktora kopiującego zdefiniowanego przez programistę. Ten konstruktor kopiujący przydziela pamięć na stercie dla parametru formalnego tej funkcji — obiektu `s` klasy `String`. Gdy wykonanie tej funkcji dobiega końca i wywołany zostaje destruktor wobec tego formalnego parametru, przyporządkowana mu na stercie pamięć zostaje zwolniona i zwrócona do systemu, natomiast nie następuje zwolnienie ani zwrot pamięci na stercie przydzielonej rzeczywistemu, bieżącemu argumentowi. Problem integralności programu znika. Pozostaje natomiast problem efektywności wykonania takiego kodu. Gdy parametr jest przekazywany poprzez wartość, wywołanie funkcji operatorowej przy wykonaniu operatora konkatenacji łańcuchów znaków obejmuje utworzenie obiektu, wywołanie konstruktora kopiującego, przydział pamięci na stercie, kopiowanie łańcucha znaków z pamięci jednego obiektu do pamięci drugiego obiektu, wywołanie destruktora i zwolnienie pamięci na stercie. Wywołanie poprzez referencję nie wymaga żadnej czynności z tej listy. Semantyka referencyjna eliminuje obniżenie wydajności poprzez wykluczenie niekoniecznego kopiowania.

> Nie przekazuj obiektów do funkcji poprzez wartość. Jeśli obiekty zawierają wewnątrz wskaźniki i dynamicznie zarządzają pamięcią na stercie, nie przekazuj takich obiektów poprzez wartość. Jeśli już koniecznie musisz przekazywać takie obiekty poprzez wartość, zdefiniuj konstruktor kopiujący, który eliminuje problem integralności aplikacji. Upewnij się, że takie kopiowanie nie pogorszy efektywności działania programu.

Zwrot poprzez wartość

Innym kontekstem wymagającym semantyki wartości jest zwrot obiektu z funkcji poprzez wartość. Zagadnienie to było już dyskutowane w rozdziale 10. w odniesieniu do klas, które nie obsługują dynamicznego zarządzania pamięcią. Ponieważ problem zwrotu obiektu z funkcji polega dokładnie na tym samym, co problem inicjowania jednego obiektu przez inny obiekt, tylko to krótko przypomnę.

Na listingu 11.6 pokazano jeszcze jedną wersję klasy `String`. Wewnątrz kodu każdego z konstruktorów umieszczono wydruk komunikatów w celu śledzenia wykonania. Dodano funkcję dokonującą przeciążenia operatora porównania zaimplementowaną jako metoda należąca do klasy. Oprócz tego dodano po stronie klienta globalną funkcję `enterData()` (wprowadź dane) i uproszczono kod funkcji `main()`. Ten program prosi użytkownika o wpisanie nazwy miasta i poszukuje wprowadzonej nazwy we własnej bazie danych. Dla uproszczenia tę bazę danych zakodowano w sposób sztywny w obrębie funkcji `main()` w formie tablicy złożonej z łańcuchów znaków i użyto najprostszego wyszukiwania sekwencyjnego do odnajdywania miasta podanego przez użytkownika. Wydruk wyjściowy demonstrujący wyniki wykonania tego programu zamieszczono na rysunku 11.16.

Listing 11.6. *Zastosowanie konstruktora kopiującego do zwrotu obiektu z funkcji*

```
#include <iostream>
using namespace std;

class String
{
char *str;                          // dynamicznie zapamiętywana macierz znakowa
int len;
char* allocate(const char* s)       // funkcja prywatna
{ char *p = new char[len+1];         // przydział pamięci na stercie – dla obiektu
if (p==NULL) exit(1);               // jeśli się nie powiodło – zakończ
strcpy(p,s);                        // kopiuj tekst na stertę
return p; }                         // zwróć wskaźnik do tekstu na stercie
public:
String (int length=0);              // konstruktor konwersji / domyślny
String(const char*);                // konstruktor konwersji
String(const String& s);            // konstruktor kopiujący
~String ();                         // zwolnij dynamiczną pamięć
void operator += (const String&);   // konkatenacja
void modify(const char*);           // zmień zawartość tablicy
bool operator == (const String&) const;       // porównaj zawartość
const char* show() const;                     // zwróć wskaźnik do tablicy
} ;

String::String(int length)
{ len = length;
str = allocate("");                 // kopiuj pusty łańcuch znaków na stertę
cout << " Zapoczątkowany: '" << str <<"'\n"; }

String::String(const char* s)
{ len = strlen(s);                  // pomiar długości wejściowego tekstu
str = allocate(s);                  // przydziel pamięć, kopiuj tekst
cout << " Utworzony: '" << str <<"'\n"; }
```

Rysunek 11.16.
Wydruk wyjściowy programu z listingu 11.6

```
Zapoczątkowany:  ''
Zapoczątkowany:  ''
Zapoczątkowany:  ''
Zapoczątkowany:  ''
Utworzony:  'Atlanta'
Utworzony:  'Boston'
Utworzony:  'Chicago'
Utworzony:  'Denver'
Podaj miasto do odszukania: Boston
Utworzony:  'Boston'
Miasto Boston zostało znalezione
```

```
String::String(const String& s)     // konstruktor kopiujący
{ len = s.len;                      // pomiar długości wejściowego tekstu
str = allocate(s.str);              // przydziel pamięć, kopiuj tekst
cout << " Skopiowany: '" << str <<"'\n"; }

String::~String()
{ delete str; }                                 // zwrot zwolnionej pamięci na stercie
void String::operator += (const String& s)      // parametr jako referencja
{ len = strlen(str) + strlen(s.str);            // całkowita długość
char* p = new char[len + 1];                    // wystarczająca ilość miejsca na stercie
if (p==NULL) exit(1);                           // czy się powiodło?
strcpy(p,str);                                  // kopiuj pierwszą część rezultatu
strcat(p,s.str);                                // dodaj drugą część rezultatu
delete str;                                     // ważna czynność
str = p; }                                      // teraz p może sobie znikać

bool String::operator==(const String& s) const   // porównaj zawartość
{ return strcmp(str,s.str)==0; }                 // strcmp() zwraca 0, jeśli takie samo

const char* String::show() const                 // chroń dane przed zmianami
{ return str; }

void String::modify(const char a[])              // bez zarządzania pamięcią
{ strncpy(str,a,len-1);                          // ochrona przed przepełnieniem
str[len-1] = 0; }                                // prawidłowe zakończenie łańcucha znaków
String enterData()
{ cout << " Podaj miasto do odszukania: ";       // komunikat: wpisz miasto
char data[200];                                  // rozwiązanie dość niezręczne
cin >> data;                                     // wczytaj dane od użytkownika
return String(data); }                           // wywołaj konstruktor

int main()
{
enum { MAX = 4} ;
String data[4];                     // baza danych, tablica złożona z obiektów
char *c[4] = { "Atlanta", "Boston", "Chicago", "Denver" };
for (int j=0; j<MAX; j++)
{ data[j] += c[j]; }                // to znaczy — data[j].operator+=(c[j]);
String u = enterData();             // katastrofa bez konstruktora
int i;
for (i=0; i < MAX; i++)             // i deklarowane poza pętlą
{ if (data[i] == u) break; }        // przerwij, jeśli znalazłeś
if (i == MAX)                       // jeśli nie znalazłeś i doszedłeś do końca
cout << " Miasto " << u.show() << " nie zostało znalezione\n";
else
```

```
cout << " Miasto " << u.show() << " zostało znalezione\n";
return 0;
}
```

Gdy w funkcji `main()` zostaje utworzona tablica składająca się z obiektów, dla każdego z elementów tej tablicy następuje wywołanie domyślnego konstruktora zawartego w specyfikacji klasy `String` (to znaczy — pierwszego konstruktora konwersji z domyślną wartością argumentu). Ten konstruktor umieszcza w pamięci pusty łańcuch znaków o zerowej długości i wyprowadza na ekran komunikat „Zapoczątkowany". Gdy wywoływana jest funkcja operatorowa `operator+=()`, aby dołączyć na zasadzie *przyrostkowej* (ang. *append*) nazwy miast do zawartości reprezentowanej przez poszczególne obiekty, tablica znakowa (tj. łańcuch znaków) jest przekazywana, jako parametr, do funkcji operatorowej realizującej przeciążenie operatora. . Przeciążony operator oczekuje parametru typu `String`, zatem w tym momencie wywołany zostaje drugi spośród konstruktorów konwersji i ten właśnie konstruktor wyprowadza na ekran komunikat „Utworzony" dla każdego obiektu — elementu tablicy.

Następnie wywołana zostaje funkcja `enterData()`. Prosi ona użytkownika o wpisanie nazwy miasta. Wczytuje i zapamiętuje dane od użytkownika i przekazuje wpisaną nazwę miasta jako argument do konstruktora konwersji klasy `String`. Z tego powodu na ekranie ponownie widzimy komunikat `Utworzony` wyprowadzony przez ten konstruktor konwersji. Ponieważ obiekt `u` jest jedynym obiektem klasy `String` w obrębie funkcji `main()`, gdy zostaje wywołana funkcja `enterData()`, wywołanie konstruktora konwersji wewnątrz kodu funkcji `enterData()` następuje w odniesieniu do obiektu `u` jako wywołanie konstruktora dla tegoż obiektu. Konstruktor kopiujący nie zostaje wywołany. Nawet jeśli obiekty klasy `String` dynamicznie zarządzają pamięcią, integralność programu jest tu chroniona. Konstruktor kopiujący nie ma tu nic do rzeczy odnośnie implementacji semantyki wartości. Taki konstruktor konwersji ma za zadanie przydzielić obiektowi `u` w funkcji `main()` jego własną, odrębną pamięć na stercie. Tak jak w dowcipie o małpie i krokodylu. Krokodyl chce — to gra na pianinie. Krokodyl chce — to śpiewa. A małpa nie ma tu nic do rzeczy.

Byśmy poczuli się bardziej komfortowo posługując się obiektami, które dynamicznie zarządzają pamięcią, wprowadzimy w obrębie funkcji `enterData()` niewielką modyfikację — dodamy tylko jeden lokalny obiekt służący nam do przechowywania danych wprowadzonych przez użytkownika.

```
String enterData()
{
cout << " Podaj miasto do odszukania: ";
char data[200];                    // rozwiązanie niezręczne
cin >> data;                       // wczytanie danych od użytkownika
String x = data;                   // konstruktor konwersji
return x; }                        // konstruktor kopiujący
```

Zmiana jest niewielka. Gdyby `x` był zmienną typu wbudowanego, w ogóle nie byłoby o czym mówić. W przypadku obiektów z dynamicznym zarządzaniem pamięcią pojawia się tu zupełnie inny problem. Gdy utworzony zostaje lokalny obiekt `x`, wywoływany jest wobec niego konstruktor konwersji. Gdy natomiast funkcja kończy swoje działanie, obiekt `u` w kodzie funkcji `main()` zostaje zainicjowany przy użyciu konstruktora kopiującego. Jeśli nie został zaimplementowany konstruktor kopiujący zdefiniowany indywidualnie przez programistę, stosowany jest tu domyślny konstruktor kopiujący dodany automatycznie przez kompilator. Ten konstruktor kopiuje pola danych obiektu `x` na pola danych obiektu `u`, ale nie dokonuje

przydziału pamięci na stercie. Wskaźniki `str` zawarte w obiektach `x` oraz `u` wskazują ten sam adres pamięci na stercie. Gdy kończy się działanie funkcji `enterData()`, a obiekt `x` zostaje usunięty, wywoływany jest wobec niego destruktor klasy `String`, który zwalnia pamięć na stercie wskazywaną przez wskaźnik `str` obiektu `x` i zwraca ją do systemu. Oznacza to, że obiekt `u` narodził się jako upośledzony od urodzenia — jego pamięć na stercie została zwolniona w chwili, gdy ten obiekt powstał.

Jakie są tego konsekwencje? Takie same, jak poprzednio. Komputer autora się zawiesił. Może się okazać, że komputer czytelnika będzie działał nadal, ale to wszystko kwestia przypadku. Program jest nieprawidłowy. Ten program wymaga zastosowania konstruktora kopiującego zdefiniowanego przez programistę.

Gdy dostarczony zostanie konstruktor kopiujący zdefiniowany przez programistę, wszystko odbywa się poprawnie. Przykładowy wydruk wyjściowy przedstawiający rezultaty działania takiego programu pokazano na rysunku 11.17.

Rysunek 11.17. *Wydruk wyjściowy programu z listingu 11.6 ze zmodyfikowaną wersją funkcji enterData() oraz z konstruktorem kopiującym*

```
Zapoczątkowany:  ''
Zapoczątkowany:  ''
Zapoczątkowany:  ''
Zapoczątkowany:  ''
Utworzony:  'Atlanta'
Utworzony:  'Boston'
Utworzony:  'Chicago'
Utworzony:  'Denver'
Podaj miasto do odszukania: Moskwa
Utworzony:  'Moskwa'
Skopiowany:  'Moskwa'
Miasto Moskwa nie zostało znalezione
```

Te, wspomagające śledzenie programu, wydruki komunikatów wyjściowych pokazują, że po wprowadzeniu przez użytkownika *wiersza danych wejściowych* (ang. *input line*) następuje wywołanie konstruktora konwersji wobec lokalnego obiektu `x` w obrębie kodu ciała funkcji `enterData()`, a następnie wywołany zostaje konstruktor kopiujący wobec lokalnego obiektu `u` w obrębie kodu funkcji `main()`. Krokodyl gra na pianinie, a małpa śpiewa. Ta wersja jest nieco powolniejsza niż poprzednia, ale nie to jest tu najważniejsze. To, co jest rzeczywiście istotne, to fakt, iż ta wersja zachowuje się inaczej niż poprzednia. Jeszcze ważniejsze jest to, że gdyby zmienne `x` oraz `u` były zmiennymi typu elementarnego, taka zmiana nie miałaby wpływu na zachowanie się programu. No i tak to jest z naszym doświadczeniem z typami wbudowanymi, które ukształtowało naszą intuicję programisty. Pomimo wszystkich tych wysiłków, elementarne typy wbudowane i typy danych definiowane przez programistę są traktowane w C++ w sposób różny. Praca z obiektami wymaga zmian w intuicji programisty. Z tego powodu autor poświęcił tyle czasu na wyliczanie tych sekwencji zdarzeń, by pomóc czytelnikowi w rozwoju takiej nowej intuicji programisty. Czytelnik powinien zyskać pewność, że czuje się swobodnie w zagadnieniach występujących na styku kodu klienta z metodami należącymi do klas, które są wywoływane w sposób niejawny.

Ograniczenia skuteczności konstruktorów kopiujących

Jesteśmy już prawie u celu. Autor chciałby tu dokonać jeszcze jednej niewielkiej zmiany w programie, tym razem po stronie kodu klienta. Zamiast definiować obiekt `u` w obrębie funkcji

`main()` i natychmiast go inicjować, obiekt ten zdefiniuje się przy użyciu konstruktora domyślnego, a następnie zostaną mu przyporządkowane te dane, które wprowadzi użytkownik w trakcie wykonania kodu wywołanej funkcji `enterData()`.

```
int main()
{
enum { MAX = 4} ;              // tablica z nazwami miast

// String u = enterData(); <- bez konstruktora kopiującego się zawiesza

String u;                      // konstruktor domyślny
u = enterData();               // crash! nie pomoże konstruktor kopiujący

// dalej – poszukiwanie miasta i wydruk rezultatów

return 0;
}
```

Po wprowadzeniu tej zmiany system na komputerze autora się zawiesił. Można sobie oszczędzić oglądania jeszcze jednego okienka dialogowego z bezużytecznymi informacjami o przyczynach tego problemu. W końcu to tylko przykładowe wykonanie na konkretnym komputerze w konkretnym systemie operacyjnym. Istotne jest to, że ten program jest nieprawidłowy. Nawet jeśli kompilacja jego kodu przebiegła poprawnie, jego zachowanie pozostaje nieprzewidywalne i taki program nie może być uruchamiany. Skoro kompilator nie powiedział nam, że ten program jest błędny, nasza intuicja programisty powinna nam pomóc w zrozumieniu tego, co w niejawny sposób odbywa się w tle podczas wykonania tego programu.

Przeciążenie operatora przypisania

Przy wielu różnych okazjach powtarzano tu, że w C++ zainicjowanie obiektów i przypisywanie wartości obiektom — to dwie różne rzeczy. Gdy mamy do czynienia z elementarnymi, wbudowanymi typami danych, taka różnica ma często znaczenie czysto akademickie. Na przykład rozważmy następujący fragment kodu klienta:

```
int v = 5;
int u = v;                // zmienna u jest inicjowana
```

i porównajmy go z następującym fragmentem:

```
int v = 5;
int u;
u = v;                    // przypisanie wartości zmiennej u
```

W pierwszym przykładzie zmienna `u` jest inicjowana w momencie deklaracji (jest to zatem jednocześnie definicja tej zmiennej). W przypadku zmiennych typów elementarnych końcowy rezultat będzie taki sam. Jeśli natomiast takie zmienne będą obiektami typu zdefiniowanego przez programistę, które obsługują własną pamięć, różnica stanie się istotna.

```
String v = "Hello"; String u = v;          // obiekt u inicjowany
String v = "Hello"; String u; u = v;       // obiekt u przypisany
```

Pierwszy wiersz podanego kodu może postawić nas w kłopotliwej sytuacji, jeśli dana klasa nie będzie zawierać konstruktora kopiującego. Drugi wiersz podanego kodu natomiast okaże się problematyczny, jeśli klasa nie będzie zawierać funkcji dokonującej przeciążenia operatora przypisania. W drugim wierszu nie będzie wywoływany konstruktor kopiujący.

Problem z dodaną przez kompilator obsługą operatora przypisania

Jeśli klasa zawiera funkcję dokonującą wobec niej przeciążenia operatora przypisania, to ta funkcja zostanie wywołana do obsługi operatora w drugim wierszu podanego fragmentu kodu klienta. Jeśli natomiast klasa nie zawiera przeciążenia operatora przypisania, kompilator automatycznie doda własną, domyślną funkcję operatorową, wykonującą tę operację wobec obiektów danej klasy. Taka funkcja operatorowa jest bardzo podobna do konstruktora kopiującego. Funkcja kopiuje pola danych (zawartość) obiektu stanowiącego prawostronny operand operatora przypisania do pól danych obiektu stanowiącego lewy operand operatora przypisania.

Podobnie jak w przypadku dodawanego automatycznie przez kompilator domyślnego konstruktora kopiującego, ta dodawana przez kompilator funkcja dokonująca przeciążenia operatora przypisania jest zawsze dostępna. Dla tych klas, które nie wykonują dynamicznego zarządzania pamięcią (np. takich klas, jak `Complex`, `Rational`, `Rectangle` itp.), taka automatycznie dodana funkcja operatorowa przeciążająca operator przypisania jest odpowiednia. W przypadku tych klas, które zarządzają pamięcią w sposób dynamiczny, taki dodany przez kompilator operator przypisania powoduje kłopoty.

Gdy wobec obiektów klasy `String` wykonywana jest operacja przypisania, pola danych są kopiowane metodą pole po polu. Wskaźnik `str` zawarty w obiekcie znajdującym się po lewej stronie operatora przypisania wskazuje tę samą lokalizację w pamięci, co wskaźnik `str` zawarty w obiekcie znajdującym się po prawej stronie operatora przypisania. Takie dwa obiekty stają się synonimami. Jeśli zmodyfikujemy jeden z tych obiektów, powiedzmy obiekt `u`, zmiana ta jest widziana przez drugi z tych obiektów, w tym przypadku — obiekt `v`.

Gdy jeden z tych obiektów jest usuwany poprzez *reguły widoczności nazw* (ang. *scope rules*) lub poprzez operator `delete` (np. obiekt `u`), wobec tego obiektu zostaje wywołany destruktor i pamięć wskazywana poprzez wskaźnik `str` zawarty w tym obiekcie zostaje zwolniona. W rezultacie drugi z tych obiektów (w tym przypadku `v`) zostaje pozbawiony przyporządkowanej mu dynamicznie na stercie pamięci, nawet jeśli po stronie kodu klienta wydaje się pozostawać zupełnie normalnym obiektem. Wszelka próba użycia takiego obiektu stanowi błąd. Gdy ten obiekt także ma zostać usunięty, zostaje wobec niego wywołany destruktor, który próbuje zwolnić pamięć na stercie wskazywaną przez wskaźnik `str` tego obiektu. Ale przecież ta pamięć została już zwolniona! Jak już wyjaśniono wcześniej, próba zwolnienia pamięci, która już uprzednio została zwolniona, powoduje wytworzenie stanu, w którym dalsze działanie programu staje się nieprzewidywalne. Nawet jeśli nie jest to błąd składniowy — jest to nieprawidłowe z semantycznego punktu widzenia.

Wyśledzenie przyczyny takiego problemu jest trudne, ponieważ nie ma tu bezpośredniego odniesienia do rezultatów działania programu, a istnienie indywidualnego konstruktora kopiującego nie jest w stanie zapobiec temu problemowi, ponieważ wtedy gdy wykonywana jest operacja przypisania, nie następuje wywołanie żadnego konstruktora. W C++ przypisanie i zainicjowanie — to nie jest to samo.

Przeciążenie przypisania — wersja pierwsza (z wyciekiem pamięci)

Rozwiązaniem tego problemu jest dokonanie przeciążenia operatora przypisania dla danej klasy. Taki poddany przeciążeniu operator przypisania musi się upewnić, że obiekty stanowiące lewostronny i prawostronny operand nie przestały wskazywać tego samego obszaru pamięci na stercie.

Wbudowany operator przypisania w C++ jest operatorem dwuargumentowym, z dwoma operandami — lewostronnym (docelowym) i prawostronnym (źródłowym). Tak samo musi być w przypadku przeciążonego operatora przypisania zdefiniowanego przez programistę. Skoro tak, interfejs takiej funkcji operatorowej jest podobny do interfejsu konstruktora kopiującego. Obiekt będący operandem lewostronnym jest docelowym obiektem komunikatu, natomiast obiekt będący operandem prawostronnym jest parametrem funkcji operatorowej.

```
u = v;                  // to oznacza wywołanie:  u.operator=(v);
```

Oznacza to, że przeciążony operator przypisania, który jest nam potrzebny dla klasy `String`, powinien mieć następujący interfejs:

```
// prototyp funkcji przeciążającej operator przypisania:
void String::operator = (const String& s);
```

Funkcja dokonująca przeciążenia operatora przypisania powinna skopiować z obiektu stanowiącego jej bieżący argument pola danych, poza wskaźnikiem, do obiektu docelowego komunikatu, a następnie przydzielić na stercie wystarczającą ilość miejsca w pamięci, by skopiować z pamięci na stercie przyporządkowanej obiektowi będącemu argumentem funkcji dane do pamięci przydzielonej obiektowi docelowemu. Te działania przypominają sposób postępowania konstruktora kopii:

1. Skopiuj długość macierzy znakowej reprezentowanej przez obiekt — parametr do pola `len` docelowego obiektu komunikatu.
2. Przyporządkuj pamięć na stercie; ustaw wskaźnik `str` w docelowym obiekcie komunikatu tak, by wskazywał początek tej pamięci na stercie.
3. Sprawdź, czy przydział pamięci na stercie się powiódł[7]. Jeśli w systemie zabrakło pamięci, zakończ działanie.
4. Kopiuj znaki reprezentowane przez obiekt-parametr do nowej, przydzielonej na stercie pamięci.

> Jeśli trzeba przypisać jeden obiekt drugiemu obiektowi, a obiekty te dokonują dynamicznego zarządzania pamięcią na stercie, upewnij się, że klasa zawiera funkcję dokonującą przeciążenia operatora przypisania. Sam konstruktor kopiujący w tej sytuacji nie wystarczy.

[7] Tj. czy wskaźnik wskazuje cokolwiek — *przyp. tłum.*

Oto wersja funkcji dokonującej przeciążenia operatora przypisania implementująca podany algorytm. Choć działa wolniej niż dodawany przez kompilator domyślny operator przypisania, zachowuje semantykę wartości i powoduje, że obydwa obiekty poddane działaniu tego operatora pozostaną wzajemnie niezależne.

```
void String::operator = (const String& s)
{ len = s.len;                        // kopiuj dane, ale bez wskaźnika
str = new char[len + 1];              // przydziel miejsce na stercie
if (str == NULL) exit(1);             // sprawdź, czy się powiodło
strcpy(str,s.str); }                  // kopiuj dane na stertę
```

To całkiem sympatycznie zaimplementowany operator przypisania, traktuje on jednak docelowy obiekt komunikatu dokładnie tak samo, jak czyni to konstruktor kopiujący — zupełnie tak, jakby obiekt wyszedł właśnie z fabryki i nie miał żadnej historii. To całkiem uprawnione podejście w przypadku konstruktora kopiującego, ale to nie jest typowa sytuacja, z którą miewa do czynienia operator przypisania. Obiekt docelowy `u` został utworzony wcześniej. Oznacza to, że wtedy, gdy ten obiekt został utworzony, wywoływany był wobec niego któryś z konstruktorów i w trakcie wykonania kodu tego konstruktora wskaźnik `str` należący do tego obiektu został już raz ustawiony tak, by wskazywał pewien adres pamięci na stercie. Operator przypisania pomija i lekceważy tę przyporządkowaną na stercie pamięć. Operator ustawia wskaźnik `str` tak, by wskazywał inną lokalizację w pamięci na stercie. Przez to pamięć przyporządkowana temu obiektowi poprzednio zostaje zgubiona (nie jest stosowana, a nie zostaje zwrócona do systemu). Taki operator przypisania powoduje wyciek pamięci. To drugi rodzaj niebezpieczeństwa czyhającego w programach pisanych w C++ oprócz ryzyka zwalniania dwukrotnie tej samej pamięci.

Jaki jest na to sposób? W przeciwieństwie do konstruktora kopiującego, funkcja dokonująca przeciążenia operatora przypisania musi zwolnić zasoby (tu — pamięć), których używał docelowy obiekt danej operacji przypisania, zanim rozpoczęło się wykonanie bieżącej operacji. Ten brak można stosunkowo łatwo uzupełnić. Trzeba tylko wiedzieć, że należy to zrobić. Oto ulepszona wersja tej samej funkcji dokonującej przeciążenia operatora przypisania.

```
// ulepszona wersja tej samej funkcji operatorowej:
void String::operator = (const String& s)
{ delete str;                    // tego nie musi robić konstruktor kopiujący
len = s.len;                     // kopiuj dane, ale bez wskaźnika
str = new char[len + 1];         // przydziel własną pamięć
if (str == NULL) exit(1);        // sprawdź, czy pamięć przydzielono
strcpy(str,s.str); }             // kopiuj dane na stertę
```

Przeciążenie przypisania — wersja następna (samoprzypisanie)

Podany kod operatora przypisania jest właściwy i adekwatny. Będzie obsługiwał nasze operatory przypisania w kodzie klienta w sposób poprawny w znakomitej większości przypadków. Jest tu jednak pewien problem, którego zapewne często nie bierze się pod uwagę. Taka funkcja operatorowa nie jest w stanie poprawnie obsłużyć wyrażenia przypisania, zapisanego w kodzie klienta w następujący sposób:

```
u = u;                // u.operator=(u);  nieczęsto robi się coś takiego, prawda?
```

To całkowicie bezużyteczna instrukcja, ale jest to w pełni legalna konstrukcja C++ dla zmiennych elementarnych typów wbudowanych. Nie ma żadnego powodu, by nie mogła to być również konstrukcja legalne dla zmiennych typów zdefiniowanych przez programistę. W istocie jest całkowicie legalna i kompilator nie zasygnalizuje tej instrukcji jako błędu składniowego. Tyle tylko, że pierwsza instrukcja z kodu naszej funkcji operatorowej `operator=()` zwalnia pamięć na stercie przyporządkowaną obiektowi-argumentowi. Gdy dochodzi do wywołania funkcji bibliotecznej `strcpy()`, funkcja ta będzie w istocie kopiowała znaki z nowej, przyporządkowanej właśnie na stercie pamięci — do tej samej pamięci. Rezultat takiego kopiowania zawartości pomiędzy wzajemnie nakładającymi się obszarami pamięci (ang. *overlaped memory areas*) jest nieprzewidywalny. I oto mamy jeszcze jeden powód do bólu głowy. Jednak nawet gdyby taki rezultat był całkowicie przewidywalny, poprzednia zawartość pamięci na stercie przyporządkowanej naszemu obiektowi — i tak przepadła bez wieści.

Jakby się to nie wydawało dziwne, takie przypisanie obiektu samemu sobie nie jest aż taką rzadkością. Takie operacje często występują w algorytmach sortowania i w algorytmach manipulujących wskaźnikami. Aby zapobiec próbie kopiowania zawartości pamięci do tego samego obszaru pamięci, funkcja operatorowa może najpierw sprawdzić, czy referencja do obiektu-argumentu wskazuje w pamięci to samo miejsce (adres), gdzie zlokalizowany jest docelowy obiekt komunikatu. Dobrym sposobem odwołania się do lokalizacji docelowego obiektu komunikatu może być posłużenie się wskaźnikiem `this`.

```
// pierwszy wiersz kodu zapobiega utracie pamięci w razie
// próby kopiowania "samego siebie na samego siebie"
void String::operator = (const String& s)
{
if (&s == this) return;          // jeśli to "samoobsługa" – zatrzymaj
delete str;                      // tego nie robi konstruktor kopiujący
len = s.len;                     // kopiowanie danych, ale bez wskaźnika
str = new char[len + 1];         // przydziel własną pamięć
if (str == NULL) exit(1);        // sprawdź, czy się powiodło
strcpy(str,s.str);               // kopiuj dane na stertę
}
```

Taki test można oczywiście przeprowadzić po stronie kodu klienta przed wywołaniem funkcji operatorowej dokonującej przeciążenia operatora przypisania, ale takie postępowanie spowodowałoby w rezultacie przenoszenie odpowiedzialności w górę, do kodu klienta, zamiast w dół, do kodu serwera.

Jeszcze jednym rozwiązaniem może tu być sprawdzenie, czy wskaźniki `str` należące odpowiednio do obiektu-argumentu i do docelowego obiektu komunikatu wskazują ten sam obszar pamięci na stercie. Taki test w kodzie funkcji operatorowej powinien wyglądać następująco:

```
// czy to ten sam obszar na stercie?
if (str == s.str) return;
```

Obydwa te środki zapobiegawcze są równoważne, ale z niewiadomych powodów ten pierwszy sposób stosowany jest częściej. Powodem może być to, że wskaźnik `this` ma dla programistów pracujących w C++ pewne dodatkowe walory estetyczne.

Przeciążenie przypisania — jeszcze jedna wersja (wyrażenia łańcuchowe)

Ta wersja poddanego przeciążeniu operatora przypisania działa poprawnie i powinna być stosowana we wszystkich klasach, które w sposób dynamiczny zarządzają swoją pamięcią

na stercie i wymagają obsługi operacji przypisania. Mimo to taka wersja operatora przypisania nie nadaje się do obsługi wyrażeń łańcuchowych, w których wartość zwracana przez operator przypisania (tj. funkcję operatorową) zostaje wykorzystana w następnej operacji przypisania.

```
// funkcja typu void nie obsłuży takich wyrażeń, jak przedstawione:
t = u = v;
```

Nie jest do końca jasne, na ile istotna jest obsługa łańcuchowych operacji przypisania. W końcu zawsze możemy w kodzie klienta zastosować sekwencję operacji przypisania z zastosowaniem operatora dwuargumentowego.

```
u = v;              // operator dwuargumentowy:  u.operator=(v);
t = u;              // operator dwuargumentowy:  t.operator=(u);
```

Tu jednak także cała istota zagadnienia sprowadza się do problemu jednakowego traktowania zmiennych elementarnych typów wbudowanych i zmiennych typów zdefiniowanych przez programistę. W przypadku zmiennych elementarnych typów wbudowanych stosowanie w kodach C++ wyrażeń łańcuchowych jest dopuszczalne. Skoro tak, powinno to być także dopuszczalne w kodach C++ wobec zmiennych typów definiowanych przez programistę.

Operator przypisania jest prawostronnie łączny, zatem znaczenie wyrażenia łańcuchowego jest następujące:

```
t = (u = v);        // t.operator = (u.operator = (v));
```

Oznacza to, że funkcja operatorowa przeciążająca operator przypisania musi zwrócić wartość, która jest odpowiednia, by mogła zostać użyta jako bieżący argument w następnym wywołaniu tej samej funkcji operatorowej (innymi słowy, w kolejnym komunikacie). To z kolei oznacza, że taka funkcja operatorowa powinna zwrócić wartość (obiekt) takiego typu, jak klasa, do której należy dana funkcja operatorowa.

```
String String::operator = (const String& s) // zwraca obiekt
{
if (&s == this) return *this;          // ochrona przed "samoobsługą"
delete str;                            // tego nie robi konstruktor kopiujący
len = s.len;                           // kopiuj dane, ale bez wskaźnika
str = new char[len + 1];               // przydziel pamięć na stercie
if (str == NULL) exit(1);              // sprawdź, czy się powiodło
strcpy(str,s.str);                     // kopiuj dane na stertę
return *this;
}
```

Na listingu 11.7 pokazano zmodyfikowaną wersję programu z listingu 11.6. Dodano funkcję dokonującą przeciążenia operatora przypisania. Ta metoda operatorowa posługuje się wywołaniem prywatnej metody `allocate()` (dosł. przydziel pamięć, rozmieść w pamięci), by zażądać miejsca w pamięci na stercie i sprawdzić, czy ta próba przydziału pamięci zakończyła się powodzeniem. Aby zmniejszyć objętość diagnostycznych wydruków wyjściowych, z wnętrza kodu konstruktora domyślnego usunięto instrukcję wyprowadzania komunikatu `Zapoczątkowany`. Zamiast tego dodano wydruk komunikatu `Przypisany`, który będzie wyprowadzany na ekran zawsze, gdy nastąpi wywołanie funkcji operatorowej przeciążającej operator przypisania. Usunięto z kodu także wywołanie operatora konkatenacji w pętli programowej po stronie kodu klienta, która ładowała zawartość do bazy danych, i zastąpiono to wywołanie operatorem przypisania. Wydruk wyjściowy tego programu pokazano na rysunku 11.18.

Listing 11.7. *Klasa String z przeciążonym operatorem przypisania*

```
#include <iostream>
using namespace std;
class String
{
char *str;                                  // dynamicznie allokowana tablica znakowa
int len;
char* allocate(const char* s)               // prywatna metoda
{ char *p = new char[len+1];                // przydział pamięci dla obiektu
if (p==NULL) exit(1);                       // jeśli się nie powiodło – zatrzymaj
strcpy(p,s);                                // kopiuj tekst na stertę
return p; }                                 // zwrot wskaźnika do pamięci na stercie
public:
String (int length=0);                      // konstruktor konwersji / domyślny
String(const char*);                        // konstruktor konwersji
String(const String& s);                    // konstruktor kopii
~String ();                                 // destruktor – zwolnienie dynamicznej pamięci
void operator += (const String&);           // konkatenacja
String operator = (const String&);          // operator przypisania
void modify(const char*);                   // zmiana zawartości tablicy
bool operator == (const String&) const;     // porównanie zawartości
const char* show() const;                   // zwrot wskaźnika do tablicy
} ;

String::String(int length)
{ len = length;
str = allocate(""); }                       // kopiuj pusty łańcuch na stertę

String::String(const char* s)
{ len = strlen(s);                          // pomiar długości tekstu wejściowego
str = allocate(s);                          // przydziel pamięć, skopiuj tekst
cout << " Utworzony: '" << str <<"'\n"; }

String::String(const String& s)             // konstruktor kopii
{ len = s.len;                              // pomiar długości tekstu wejściowego
str = allocate(s.str);                      // przydziel pamięć, skopiuj tekst
cout << " Skopiowany: '" << str <<"'\n"; }

String::~String()
{ delete str; }                             // zwrot pamięci na stercie
void String::operator += (const String& s)          // parametr jako referencja
{ len = strlen(str) + strlen(s.str);                // całkowita długość
char* p = new char[len + 1];                // przydział wystarczającej ilości miejsca
if (p==NULL) exit(1);                       // sprawdź, czy się powiodło
strcpy(p,str);                              // kopiuj pierwszą część rezultatu
strcat(p,s.str);                            // dodaj drugą część rezultatu
delete str;                                 // ważna czynność
str = p; }                                  // teraz p może zniknąć

String String::operator = (const String& s)
{ if (&s == this) return *this;             // czy to "samoobsługa"?
delete str;                                 // tego nie robi konstruktor kopiujący
len = s.len;                                // kopiuj dane, ale bez wskaźnika
str = allocate(s.str);                      // przydziel pamięć, skopiuj tekst
cout << " Przypisany: '" << str <<"'\n";            // w celu śledzenia
return *this; }                             // zwróć obiekt docelowy do kodu klienta
```

Rysunek 11.18.
Wydruk wyjściowy programu z listingu 11.7

```
Utworzony:  'Atlanta'
Przypisany:  'Atlanta'
Skopiowany:  'Atlanta'
Utworzony:  'Boston'
Przypisany:  'Boston'
Skopiowany:  'Boston'
Utworzony:  'Chicago'
Przypisany:  'Chicago'
Skopiowany:  'Chicago'
Utworzony:  'Denver'
Przypisany:  'Denver'
Skopiowany:  'Denver'
Podaj miasto do odszukania: Denver
Utworzony:  'Denver'
Przypisany:  'Denver'
Skopiowany:  'Denver'
Miasto Denver zostało znalezione
```

```
bool String::operator==(const String& s) const     // porównanie zawartości
{ return strcmp(str,s.str)==0; }                   // zwraca 0, jeśli takie samo
const char* String::show() const                   // ochrona danych przed zmianą
{ return str; }
void String::modify(const char a[]) // bez zarządzania pamięcią
{ strncpy(str,a,len-1);                // ochrona przed przepełnieniem
str[len-1] = 0; }                      // poprawne zakończenie łańcucha znaków

String enterData()
{ cout << " Podaj miasto do odszukania: ";         // pytamy użytkownika
char data[200];                                    // niezgrabne rozwiązanie
cin >> data;                           // wczytujemy miasto podane przez użytkownika
return String(data); }                 // konstruktor konwersji

int main()
{
cout << endl << endl;
enum { MAX = 4} ;
String data[4];                        // macierz złożona z obiektów (baza danych)
char *c[4] = { "Atlanta", "Boston", "Chicago", "Denver" };
for (int j=0; j<MAX; j++)
{ data[j] = c[j]; }                    // przypisanie:
                                       // data[j].operator=(c[j]);
String u; int i;
// następujące wyrażenie wymaga przypisania,
// a nie wywołania konstruktora kopii
u = enterData();
for (i=0; i<MAX; i++)
{ if (data[i] == u) break; }           // instrukcje warunkowe
                                       // (data[i].operator==(u))
if (i == MAX)
cout << " Miasto " << u.show() << " nie zostało znalezione\n";    // nie ma
else
cout << " Miasto " << u.show() << " zostało znalezione\n";        // jest
return 0;
}
```

Jak widać, problem integralności programu zniknął. Możemy postępować z obiektami klasy `String` dokładnie w taki sam sposób, jak ze zmiennymi elementarnych, wbudowanych typów numerycznych. Możemy utworzyć takie obiekty bez ich zainicjowania, możemy je zainicjować, posłużywszy się do tego celu macierzą znakową, możemy także zainicjować takie obiekty, posługując się innym, utworzonym wcześniej, obiektem tego samego typu `String`. Możemy jeden obiekt klasy `String` przypisać operatorem = drugiemu obiektowi klasy `String` tak, jakby były to liczby. Zwróć uwagę, że C++ nie pozwoli na to w odniesieniu do macierzy. Macierze w C++ posługują się semantyką referencji, a nie semantyką wartości.

Możemy do klasy `String` dodać tyle funkcji przeciążających operatory arytmetyczne, ile tylko się zmieści (dodawanie obiektów klasy `String`, odejmowanie, mnożenie itp.). Powinniśmy jednakże pamiętać tu o losie programisty — serwisanta kodów. Nie powinniśmy doprowadzać do tego, by zadanie polegające na zrozumieniu naszych kodów było trudniejsze, niż jest to rzeczywiście konieczne.

Wbudowana w C++ możliwość przeciążania operatorów stanowi znaczący wkład do estetyki programowania komputerów.

Elastyczność C++ ma swoją cenę. Jak zawsze — coś za coś. Jeśli chcemy zainicjować jeden obiekt, posłużywszy się w tym celu innym obiektem (przy definiowaniu obiektu, przy przekazywaniu go jako parametru poprzez wartość lub przy zwrocie obiektu poprzez wartość z funkcji), powinniśmy zdecydowanie dodać do klasy konstruktor kopii. Jeśli chcemy dokonać przypisania jednego obiektu innemu obiektowi, powinniśmy zdecydowanie dodać do specyfikacji klasy funkcję operatorową dokonującą przeciążenia operatora przypisania.

Problemy z zachowaniem integralności programu, które mogą wystąpić z powodu stosowania nieklarownego dynamicznego zarządzania pamięcią, są na tyle niebezpieczne, że wielu programistów implementuje konstruktor kopiujący i funkcję operatorową operatora przypisania dla każdej klasy, która dynamicznie zarządza pamięcią. Programiści robią to często nawet w takich klasach, które nie zarządzają pamięcią w sposób dynamiczny. W końcu napisanie tych funkcji nie wymaga wiele wysiłku — zawsze lepiej to zrobić, choćby tylko tak, na wszelki wypadek.

Autor uważa, że to jest problem z kategorii „dmuchania na zimne". Zamiast dodawać do kodu wiele kompletnie bezużytecznych funkcji, projektant powinien uważnie przeanalizować rzeczywiste wymagania kodu klienta i zrozumieć konsekwencje różnorodnych decyzji projektowych.

Z dostarczaniem klas z ilością metod przewyższającą tę, której klasa rzeczywiście potrzebuje, wiążą się liczne problemy. Jednym z nich jest rozdęty ponad miarę projekt. To nie jest mało znacząca konsekwencja. Gdy serwisant kodów (lub programista piszący kod klienta) przegląda kody bezużytecznych funkcji, nie zwraca uwagi na inne istotne szczegóły.

Inną kwestią jest tu efektywność działania kodu. Jak widać na rysunku 11.18, problem ten może stać się zupełnie realny. Dla każdego przypisania wejściowego łańcucha znaków w obrębie pętli programowej potrzebne są dwa wywołania funkcji oraz wywołanie funkcji operatorowej przeciążającej operator przypisania:

1. Wywołanie konstruktora konwersji wobec argumentu funkcji operatorowej `operator=()`.
2. Wywołanie samej funkcji operatorowej `operator=()`.

3. Wywołanie konstruktora kopii w celu zwrócenia obiektu poprzez wartość z wnętrza funkcji operatorowej.

Pomimo tych wysiłków nadal występują znaczne różnice pomiędzy traktowaniem obiektów klas definiowanych przez programistę a traktowaniem zmiennych typów wbudowanych. Gdyby tablice `data[]` oraz `c[]` składały się z elementów typu elementarnego, wewnątrz naszej pętli programowej mogłaby się znajdować tylko jedna, pojedyncza instrukcja. Przy takiej konstrukcji klasy `String` wnętrze tej pętli programowej reprezentuje coś zupełnie innego — aż trzy wywołania funkcji.

```
for (int j=0; j<MAX; j++)
{ data[j]=c[j]; }            // przypisanie: data[j].operator=(String(c[j]));
```

Zwróć uwagę, że każda z takich operacji jest dość kosztowna (w sensie czasu wykonania). Poza tym oprócz wywołań funkcji każda z tych operacji pociąga za sobą konieczność przydziału pamięci na stercie, skopiowania parametru — łańcucha znaków do pamięci przydzielonej na stercie, a następnie, podczas wywołania destruktora, zwolnienia tej pamięci i jej zwrotu do systemu. Wykonanie tych wszystkich działań jest nieuniknione jeden raz — dla operatora przypisania, który posługuje się semantyką wartości w celu utrzymywania pamięci na stercie odrębnie dla swoich dwu operandów. Jednak robić to jeszcze dwa razy? Raz — wobec parametru funkcji operatorowej przeciążającej operator przypisania. Drugi raz — wobec wartości zwracanej z tej funkcji. Wygląda na to, że to zbyt wiele. Aby wyczerpać ten kłopotliwy temat do końca dodajmy, że obiekt tworzony przez konstruktor kopiujący nie jest używany przez kod klienta (przypomnę, że zwrot obiektu z funkcji został wprowadzony tylko po to, by poprawnie obsługiwać łańcuchowe operacje przypisania). Zostaje całkowicie pozostawiony własnemu losowi, a następnie usunięty po wywołaniu destruktora.

Pierwszy środek zapobiegawczy — więcej przeciążania

Są dwa sposoby, by poprawić efektywność działania takiego poddanego przeciążeniu operatora przypisania. Zmiana typu parametru funkcji operatorowej z obiektu klasy `String` na macierz znakową (łańcuch znaków) pozwala na wyeliminowanie wywołania konstruktora konwersji.

```
String String::operator=(const char s[])       // tablica jako parametr
{ delete str;                                  // tego nie robi konstruktor kopiujący
len = strlen(s);
str = allocate(s);                             // przydziel pamięć, skopiuj tekst
cout << "Przypisany: '" << str <<"'\n";        // do śledzenia wykonania
return *this; }
```

Jeśli chcemy, by poddany przeciążeniu wobec klasy `String` operator przypisania obsługiwał i przypisanie obiektowi tablicy znakowej, i przypisanie obiektu klasy `String`, musimy dwukrotnie poddać operator przypisania przeciążeniu — oddzielnie dla parametru typu tablica znakowa, oddzielnie dla typu `String`. Wydruk wyjściowy programu z listingu 11.7 po dodaniu do specyfikacji klasy drugiego przeciążonego operatora pokazano na rysunku 11.19. W komunikacie diagnostycznym wydrukowanym z wnętrza funkcji obsługującej drugi z operatorów przypisania zostało dodanych kilka spacji, by łatwo było optycznie odróżnić komunikaty drukowane przez funkcję obsługującą pierwszy operator przypisania (funkcja z parametrem typu `String`) od komunikatu wydrukowanego przez funkcję obsługującą drugi operator przypisania (z parametrem typu macierz znakowa).

Rysunek 11.19.
Wydruk wyjściowy programu z listingu 11.7 po dodaniu drugiej funkcji przeciążającej operator przypisania

```
Przypisany:  'Atlanta'
Skopiowany:  'Atlanta'
Przypisany:  'Boston'
Skopiowany:  'Boston'
Przypisany:  'Chicago'
Skopiowany:  'Chicago'
Przypisany:  'Denver'
Skopiowany:  'Denver'
Podaj miasto do odszukania: Atlanta
Utworzony:  'Atlanta'
Przypisany:  'Atlanta'
Skopiowany:  'Atlanta'
Miasto Atlanta zostało znalezione
```

Drugi środek zapobiegawczy — zwrot wartości poprzez referencję

Drugim sposobem, by poprawić efektywność działania programu, jest wyeliminowanie nadmiarowych wywołań konstruktora kopiującego. Aby to osiągnąć, powinniśmy zastąpić zwrot poprzez wartość zwrotem poprzez referencję. Oto przykład takiego zastąpienia w kodzie funkcji przeciążającej operator przypisania, której parametrem jest macierz znakowa.

```
String& String::operator = (const char s[])    // zwraca referencję
{ delete str;                                  // tego nie robi konstruktor kopiujący
len = strlen(s);
str = allocate(s);                             // przydziel pamięć, skopiuj tekst
cout << " Przypisany: '" << str <<"'\n";       // dla śledzenia
return *this; }
```

To samo powinniśmy zrobić w pierwszej funkcji przeciążającej operator przypisania z parametrem typu `String`. Gdy z funkcji zwracane są referencje (więcej na ten temat napisano w dyskusji o funkcjach w rozdziale 9.), powinniśmy postępować ostrożnie i zawsze się upewnić, że referencja ciągle wskazuje na ważny (funkcjonujący) obiekt, który pozostanie przy życiu, gdy dana funkcja zakończy swoje działanie. W tym przypadku nie ma takiego niebezpieczeństwa. Referencja, która zostaje zwrócona z funkcji operatorowej, jest referencją do obiektu stanowiącego lewostronny operand operatora przypisania w przestrzeni klienta, np. `data[i]` w podanym przykładzie pętli programowej. Ten obiekt pozostaje przy życiu po zakończeniu działania funkcji operatorowej, ponieważ został zdefiniowany w przestrzeni klienta (zakresie widoczności nazw klienta). Bądź ostrożny przy próbach zwrotu referencji do obiektów zdefiniowanych w przestrzeni nazw serwera, które znikają po powrocie z wywołania funkcji serwera. Wiele kompilatorów wydrukuje jedynie *komunikat ostrzegawczy* (ang. *warning*) i taki błąd może ci ujść bezkarnie.

Wydruk wyjściowy programu z listingu 11.7, w wersji z zastosowaniem dwóch funkcji przeciążających operator przypisania zwracających referencje do obiektu, pokazano na rysunku 11.20.

To wygląda tak, jakbyśmy zamęczyli już biedny operator przypisania na śmierć, ale to nie jest jego dumny koniec. Niektórzy puryści upieraliby się, że to jeszcze nie wystarczy, ponieważ taka konstrukcja nie chroni autora kodu klienta przed koniecznością wykonywania czynności, które przecież nie są niezbędne, jak np. modyfikowanie zawartości zwróconego obiektu

```
Przypisany:  'Atlanta'
Przypisany:  'Boston'
Przypisany:  'Chicago'
Przypisany:  'Denver'
Podaj miasto do odszukania: Denver
Utworzony:  'Denver'
Przypisany:  'Denver'
Miasto Denver zostało znalezione
```

Rysunek 11.20. *Wydruk wyjściowy programu z listingu 11.7 z zastosowaniem dwóch funkcji przeciążających operator przypisania, zwracających referencje do obiektu klasy String*

reprezentującego łańcuch znaków przed usunięciem tego obiektu. Na przykład następujący fragment kodu jest konstrukcją legalną w C++ w odniesieniu do operatorów przypisania w wersji z listingu 11.7.

```
// komunikat: miasto, o którym nikt nigdy nie słyszał
for (int j=0; j<MAX; j++)
{ (data[j] = c[j]).modify("Miasto, o którym nikt nie słyszał."); }       // legalne
```

Ten kod przypisuje jeden obiekt drugiemu obiektowi, zwraca referencję do obiektu docelowego i natychmiast wysyła komunikat nakazujący zmodyfikowane tegoż obiektu. Przypisana wartość nie zostaje nigdy użyta. Nie ma to szczególnego sensu, zatem powinno zostać zasygnalizowane jako błąd składniowy. Jeśli chcemy, by kompilator rzeczywiście wygenerował komunikat o błędzie składniowym, powinniśmy tę zwracaną referencję zadeklarować jako referencję do stałej (przy użyciu słowa `const`).

```
const String& String::operator = (const char s[])      // zbyt wiele?
{ delete str;               // tego nie robi się w konstruktorze kopiującym
len = strlen(s);
str = allocate(s);          // przydział pamięci, skopiowanie tekstu
cout << " Przypisany: '" << str <<"'\n";               // do śledzenia
return *this; }
```

Autor nie ma przekonania, czy warto się tu upierać, że koniecznie należałoby to zrobić, jednak puryści uzyskali tu punkt. Jeśli coś nie ma sensu, nie powinno być dozwolone, by nie mogło wystąpić jako legalna część kodu.

Rozważania praktyczne — jak chcielibyśmy to zaimplementować?

Dynamiczne zarządzanie pamięcią powinno być obsługiwane w oparciu o wiedzę i zrozumienie. Odstąpienie na krok od reguł w jakimkolwiek kierunku powoduje ryzyko albo spadku efektywności działania, albo utraty integralności programu.

Wielu programistów uważa, że zawsze gdy projektujemy klasę, która dynamicznie zarządza pamięcią, musimy wyposażyć tę klasę w pełny zestaw pomocniczych metod:

- konstruktor domyślny,
- konstruktor (konstruktory) konwersji,
- konstruktor kopiujący,

- przeciążony operator przypisania,
- destruktor.

Autor nie jest przekonany, czy należy w sposób automatyczny przestrzegać takich zaleceń. W zależności od wymagań kodu klienta możemy potrzebować tylko części spośród tych funkcji. Jeśli wyposażymy klasę w funkcje dokonujące przeciążenia operatorów z nieprawidłowymi interfejsami, wyeliminujemy w ten sposób problem integralności programu, ale jednocześnie pogorszymy efektywność działania kodu programu bez żadnych racjonalnych powodów. Autor jest natomiast pewien, że musimy rozumieć zagadnienia omówione w niniejszym rozdziale. To zrozumienie pozwoli nam na dobór metod zgodnie z oczekiwanymi zadaniami do wykonania (tj. zgodnie z wymaganiami kodu klienta) oraz na zaprojektowanie klasy, która jednocześnie jest i efektywna, i poprawna. Jeśli automatycznie wyposażamy klasę w całą tę maszynerię, kod klienta wykonuje się poprawnie, ale tracimy ostrość widzenia zagadnień i zapominamy o różnicy pomiędzy inicjowaniem a przypisaniem. To jest wręcz niebezpieczne.

Należy zawsze się upewnić, że w odniesieniu do klas stosuje się właściwe narzędzia. Jeśli pojawią się problemy, należy przeanalizować sytuację, zastosować komunikaty ułatwiające śledzenie działania programu, narysować schematyczne diagramy, ale nie przeciążać klas komponentami, które nie są im niezbędne. Upewnij się, że dobrałeś odpowiednie narzędzia do pracy wymagającej wykonania. Nie krąż wokół bezproduktywnie z młotkiem i gwoździami w rękach (konstruktory, funkcje operatorowe przypisania i inne narzędzia), szukając oparcia dopiero na przeciwległej ścianie. Pamiętaj, że konstruktor kopiujący i operator przypisania służą do rozwiązywania różnych zadań i nie mogą być stosowane zamiennie. W przenośni można o nich powiedzieć, że służą jakby do zawieszania obrazków na przeciwległych sobie ścianach.

Często kod klienta nie potrzebuje możliwości zainicjowania jednego obiektu przy użyciu innego obiektu ani przypisania jednego obiektu drugiemu obiektowi. Załóżmy, że klasa, którą mamy zaimplementować, reprezentuje okno. Dla uproszczenia rozpatrzmy tylko jedno pole danych, reprezentujące tekst, który ma być wyświetlany w tym oknie. Taka klasa, powiedzmy `Window`, jest podobna do klasy `String`. Zawiera tablicę znakową umieszczaną na stercie w dynamicznie przydzielanej pamięci, destruktor i funkcję dokonującą przeciążenia operatora konkatenacji, akceptującą jako argument tablicę znakową, która ma zostać wyświetlona w oknie i dodaje ten tekst do zawartości okna.

```
class Window
{
char *str;                                  // dynamicznie allokowana tablica znakowa
int len;
public:
Window()
{ len = 0; str = new char; str[0]= 0 ; }    // pusty łańcuch
~Window()
{ delete str; }                             // zwrot pamięci na stertę
void operator += (const char s[])           // tablica jako parametr
{ len = strlen(str) + strlen(s);
char* p = new char[len + 1];                // przydziel wystarczającą ilość miejsca
if (p==NULL) exit(1);
strcpy(p,str); strcat(p,s);                 // utwórz dane ze składników
delete str; str = p; }                      // ustaw wskaźnik na nowe dane
const char* show() const
{ return str; } } ;                         // wskaźnik do zawartości
```

Aby mieć „pełnosprawne” okno, potrzebowalibyśmy więcej pól danych i metod należących do tej klasy, ale taki projekt wystarcza do zaprezentowania związanych z tym zagadnień.

Oczywiście, w aplikacji jest nieco mniej obiektów klasy `Window` niż obiektów klasy `String`. Poza tym gdy tworzony jest obiekt klasy `Window`, zostaje on zainicjowany tak, by jego zawartość była pustym łańcuchem znaków (puste okno), a dane są dodawane w miarę wykonania kodu programu.

Jak pokazano na początku niniejszego rozdziału, to dokładnie ten rodzaj klas, których obiekty nie mogą być przekazywane poprzez wartość. Co się stanie, gdy programista piszący kod klienta przekaże obiekt klasy `Window` jako parametr poprzez wartość albo po prostu zapomni o wpisaniu operatora `&` i w ten sposób utworzy przekazywanie poprzez wartość niechcący?

```
void display(const Window window)            // nie rób tak!
{ cout << window.show(); }
```

Nie ma sensu inicjowanie jednego okna przy użyciu innego ani przypisywanie jednemu obiektowi typu „okno” innego obiektu tego typu.

```
// Welcome... – Witamy drogiego Klienta!
Window w1; w1 += "Welcome, Dear Customer!";    // rozsądne
Window w2 = w1;                                // nierozsądne
w2 = w1;                                       // jeszcze bardziej nierozsądne
display(w2);                                   // przekazanie przez wartość – powolne
```

Nie. Drugi i trzeci wiersz w podanym fragmencie kodu — zdecydowanie nie mają sensu. Znakomita większość ludzi nie napisałaby czegoś takiego. Co więcej, argument do funkcji `display()` zostaje przekazany poprzez wartość. Większość ludzi (szczególnie spośród tych, którzy czytają tę książkę) nie napisałaby tak. Skoro większość ludzi nie pisałaby programu w taki sposób, czy oznacza to, że klasa `Window` mogłaby zostać zbudowana bez konstruktora kopiującego albo bez przeciążenia operatora przypisania? Jeśli ktoś (kto najwyraźniej nie czytał tej książki) napisał kod podobny do podanego fragmentu, to byłby w stanie spowodować jednocześnie problemy z zachowaniem i integralności, i efektywności działania programu. Choć przecież ten kod jest formalnie legalny w C++.

Czy do naszej klasy `Window` powinniśmy dodać obszerne komentarze? „Szanowny Programisto, piszący kod klienta, nie inicjuj, proszę, obiektów klasy `Window` przy użyciu innych obiektów klasy `Window`. No i, bardzo, bardzo proszę, nie przekazuj obiektów klasy `Window` poprzez wartość jako parametrów do funkcji ani nie zwracaj tych obiektów poprzez wartość z funkcji. Inaczej Twój program będzie miał kłopoty”. Miło byłoby zrobić coś więcej w celu ochrony kodu klienta.

Jednym ze sposobów jest dodanie do klasy konstruktora kopiującego i funkcji operatorowej przeciążającej operator przypisania. Jeśli teraz programista tworzący kod klienta napisze niewłaściwy kod, przynajmniej nie będzie on powodował problemów z zachowaniem integralności programu.

Innym sposobem jest doprowadzenie do takiej sytuacji, by niepożądany kod był niedopuszczalny ze składniowego punktu widzenia. To bardzo ciekawa koncepcja.

Istota sprawy polega na tym, by zaprojektować naszą klasę w taki sposób, że próba niewłaściwego użycia obiektów tej klasy w kodzie klienta będzie wychwytywana przez kompilator jako błąd składniowy. To projektant klasy decyduje, jakie zastosowanie jest nieprawidłowe. Potem już nie potrzebujemy żadnego komentarza w stylu „Szanowny Programisto, piszący kod klienta…".

Tyle tylko, że to nie jest takie łatwe. Możemy usunąć konstruktor kopiujący i funkcję przeciążającą operator przypisania zdefiniowane przez programistę, ale kompilator automatycznie doda do naszej klasy domyślne wersje konstruktora kopiującego i operatora przypisania. A to są właśnie te funkcje, metody dodawane automatycznie przez system, które powodują powstawanie problemów z zachowaniem integralności w przypadku klas z dynamicznym zarządzaniem pamięcią. Aby temu zapobiec, dodajmy do klasy konstruktor kopiujący i funkcję dokonującą przeciążenia operatora przypisania zdefiniowane przez programistę. Trzeba to jednakże zrobić w taki sposób, by kod klienta nie mógł ich zastosować i by każda próba ich wywołania powodowała wystąpienie błędu składniowego.

Czy już widać, dokąd zmierza i prowadzi nas autor? Autor namawia oto czytelnika do napisania funkcji, której kod klienta nie może wywołać. Jak można napisać taką funkcję, by kod klienta nie mógł jej wywoływać? Jednym z możliwych rozwiązań jest zadeklarowanie tej funkcji jako metody niepublicznej poprzez umieszczenie jej w sekcji prywatnej (lub chronionej).

Na listingu 11.8 przedstawiono takie właśnie rozwiązanie. Konstruktor kopiujący i metoda dokonująca przeciążenia operatora przypisania zostały zadeklarowane jako prywatne. Po takiej deklaracji nie muszą nawet zostać zaimplementowane. Jeśli podany jest tylko prototyp funkcji, a funkcja zostaje wywołana przez kod klienta, jest to *błąd konsolidacji* (ang. *linker error*). W tym przypadku konsolidator nie znajdzie kodu funkcji. Kompilator zaprotestuje, że trzy ostatnie wiersze kodu w obrębie funkcji `main()` są błędne. Jeśli deklaracje tej metody dokonującej przeciążenia operatora i konstruktora kopiującego poprzedzimy znakiem komentarza (w ten sposób wyłączając je z kodu), kompilator zaakceptuje ten sam kod klienta, aprobując w ten sposób formalnie i przygotowując do wykonania ten naprawdę nierozsądny kod.

Listing 11.8. *Przykład prywatnych prototypów w celu wykluczenia nieprawidłowego posługiwania się obiektami po stronie klienta*

```
#include <iostream>
using namespace std;

class Window
{
char *str;                              // dynamicznie umieszczana tablica znakowa
int len;
Window(const Window& w);                // prywatny konstruktor kopiyjący
Window& operator = (const Window &w);   // prywatny operator przypisania
public:
Window()
{ len = 0; str = new char; str[0]= 0 ; } // pusty łańcuch
~Window()
{ delete str; }                         // zwrot pamięci na stertę
void operator += (const char s[])       // tablica jako parametr
{ len = strlen(str) + strlen(s);
char* p = new char[len + 1];            // przydziel wystarczającą ilość miejsca
if (p==NULL) exit(1);
strcpy(p,str); strcat(p,s);             // utwórz dane ze składników
```

```
delete str; str = p; }                    // ustaw wskaźnik na nowe dane
const char* show() const
{ return str; } } ;                       // zwrot wskaźnika do danych

// nie przekazuj obiektów poprzez wartość
void display(const Window window)
{ cout << window.show(); }

int main()
{
Window w1; w1 += "Welcome, Dear Customer!\n";   // rozsądne
Window w2 = w1;                                 // nierozsądne – błąd składni
w2 = w1;                                  // jeszcze bardziej nierozsądne – błąd składni
display(w2);                              // przekazanie poprzez wartość – błąd składni
return 0;
}
```

To doskonała metoda, by zapobiec niewłaściwemu stosowaniu naszych klas przez programistę piszącego kod klienta. Oczywiście jeśli taki kod, który w obrębie funkcji `main()` na listingu 11.8 został opatrzony delikatnym komentarzem „nierozsądne", musi być poprawnie obsługiwany (z jakichkolwiek powodów), a nie ma ograniczeń ze strony efektywności działania, klasa musi zawierać konstruktor kopiujący i funkcję przeciążającą operator przypisania lub kilka wersji operatorów przypisania, jeśli możliwe jest stosowanie wielu typów wyrażeń występujących po prawej stronie operatora przypisania. Dopóki rozważany jest taki operator konwersji (operatory konwersji), powinniśmy dołączyć do klasy stosowne funkcje, skoro obiekty danej klasy mają być inicjowane za pomocą prostych zmiennych zawierających dane, a nie za pomocą obiektów tego samego typu. Innym uzasadnieniem, by dodać do klasy operatory konwersji, jest chęć uniknięcia konieczności stosowania wielokrotnych funkcji dokonujących przeciążenia operatorów. W ten sposób zmniejszamy ilość funkcji zawartych w obrębie klasy, ale w zamian mamy dodatkowe wywołania konstruktora i operacje przydziału dynamicznej pamięci.

Podsumowanie

W niniejszym rozdziale przyglądaliśmy się ciemnym stronom potęgi C++. Autor nie zamierzał przestraszyć czytelnika, lecz raczej uświadomić mu powagę sytuacji i skalę odpowiedzialności programisty piszącego w C++, od którego zależy i efektywność działania programu, i zachowanie jego integralności.

Autor wytoczył kolejne ważkie argumenty przeciwko przekazywaniu parametrów poprzez wartość i ma nadzieję, że w naszych programach nie będziemy skłonni pójść na żaden kompromis. Przekazujmy parametry poprzez referencje i stosujmy modyfikator `const` do wskazania, że określony parametr nie zostanie zmodyfikowany podczas wykonania danej funkcji.

Autor przytaczał także argumenty przeciwko zwrotowi obiektów z funkcji poprzez wartość. Jeśli musimy zwrócić z funkcji obiekt, zwróćmy referencje do tego obiektu, upewniwszy się jednak, że jest to referencja do takiego obiektu, który nie zniknie natychmiast po powrocie z danego wywołania funkcji.

Jeśli jesteśmy zdecydowanie przekonani, że kod klienta nie powinien przekazywać obiektów naszej klasy poprzez wartość, zadeklarujmy konstruktor kopiujący jako prywatny poprzez umieszczenie jego prototypu w prywatnej sekcji specyfikacji klasy. W takiej sytuacji nie ma potrzeby implementowania takiego konstruktora.

Jeśli nasza klasa dynamicznie zarządza pamięcią, upewnijmy się, że wyposażyliśmy tę klasę w destruktor, który zwalnia i zwraca do systemu pamięć na stercie.

Jeśli obiekty naszej klasy mają być po stronie kodu klienta wykorzystywane do inicjowania jednego obiektu przy użyciu innego obiektu tej samej klasy, dodajmy do specyfikacji klasy konstruktor kopiujący, którego implementacja stosuje semantykę wartości wobec naszej klasy i wyposaża każdy obiekt w jego własny, odrębny segment pamięci na stercie. W kodzie przeciążającym operator przypisania upewnijmy się, że zapobiegliśmy wyciekom pamięci poprzez zwrot do systemu tego segmentu pamięci na stercie, którym dysponował docelowy obiekt przed przypisaniem mu nowej wartości skopiowanej ze źródłowego obiektu-parametru. Upewnijmy się, że ta pamięć nie zostaje zwrócona przed sprawdzeniem, czy nie mamy tu do czynienia z samokopiowaniem (czyli przypisaniem obiektowi jego własnej wartości). Zdecydujmy, czy chcemy obsługiwać operacje łańcuchowe. Często nasi klienci nie będą mieć takich potrzeb.

Zastosowanie konstruktorów konwersji pozwala na znaczące rozluźnienie rygorystycznych reguł kontroli zgodności typów w C++. Jako rzeczywisty, bieżący argument możemy przekazywać dane innego typu niż typ wymagany przez daną klasę, a mimo to kod będzie uznawany za formalnie poprawny. To wspaniałe, ale stosujmy takie techniki ostrożnie. Dodatkowe wywołania konstruktorów konwersji są kosztowne (w sensie czasu wykonania), szczególnie wtedy, gdy musimy stosować semantykę wartości.

Oczywiście — jeszcze jedno. Upewnijmy się, że rozróżniamy, w którym miejscu kod klienta wywołuje konstruktor kopiujący, a w którym — funkcję operatorową przeciążającą operator przypisania. W obu przypadkach operacja po stronie klienta jest oznaczana tym samym symbolem równości, ale powoduje to wywołanie różnych funkcji po stronie kodu serwera. Powinniśmy wiedzieć, która z nich jest stosowana w którym miejscu.

Autor radzi często wracać do materiału, który obejmuje niniejszy rozdział. Rysujmy diagramy wykorzystania pamięci, eksperymentujmy z przykładowymi kodami. Pamiętajmy zawsze, że dynamiczne zarządzanie pamięcią w C++ może łatwo przerodzić się w coś w rodzaju słonia w składzie porcelany albo wycieczki czołgiem po sąsiedzkich ogródkach. To źle, że do tradycyjnych kategorii błędów programistycznych, błędów składniowych i błędów semantycznych (w ruchu, ang. *run-time errors*) C++ dodaje jeszcze jedną kategorię błędów. Program może być formalnie poprawny składniowo i równocześnie poprawny semantycznie, a mimo to — nadal być nieprawidłowy. Żaden inny język programowania nie obciąża programisty tak ogromną odpowiedzialnością. Miejmy zawsze pewność, że przyjmujemy tę odpowiedzialność z należnym respektem.

Powodzenia.