helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

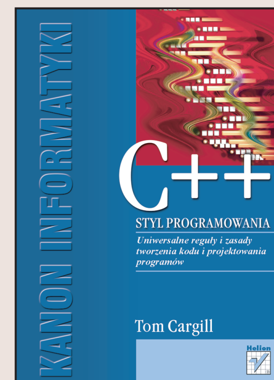
TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

C++. Styl programowania

Autor: Tom Cargill
Tłumaczenie: Adam Majczak
ISBN: 83-7361-321-8
Tytuł oryginału: C++ Programming Style
Format: B5, stron: 224



C++ wspomaga programowanie w dużej skali, pozwalając na precyzyjne wyrażenie współzależności pomiędzy różnymi częściami programu. Dlatego zakres pojęciowy techniki i stylu programowania w C++ wykracza poza tradycyjne jego pojmowanie w odniesieniu do programowania w małej skali, sprowadzającego się do szczegółów kodowania wiersz po wierszu.

Autor dowodzi, że nieprzemyślane stosowanie złożonych i zaawansowanych technik programowania może prowadzić do tworzenia chaotycznych, niezrozumiałych i mętnych konstrukcji, stanowiących zarazem często rozwiązania mniej efektywne, niż prostsze i zrozumiałe konstrukcje alternatywne. Tom Cargill dokonuje przeredagowania licznych programów, stosując techniki pozwalające na udoskonalenie kodu, począwszy od poprawy spójności, po usuniecie zbędnego, nadmiarowego dziedziczenia. Sposób prezentacji zagadnień rozpoczyna się od przeglądu oryginalnego kodu, który możesz samodzielnie ocenić i przeanalizować, rozważając możliwe alternatywne podejścia do przedstawionych zagadnień programistycznych. Te własne przemyślenia możesz następnie porównać z analizami i wnioskami Autora.

Na podstawie przykładów formułowane są uniwersalne reguły i zasady tworzenia kodu i projektowania programów. Zrozumienie i umiejętne stosowanie tych reguł pomoże profesjonalnym programistom projektować i pisać lepsze programy w C++.

Kolejne rozdziały poświęcone są następującym zagadnieniom:

- Abstrakcja — pojęcia i modele abstrakcyjne
- Spójność
- Zbędne dziedziczenie
- Funkcje wirtualne
- Przeciążanie operatorów
- Nakładki typu „wrapper"
- Efektywność

Po wprowadzeniu i zilustrowaniu reguł programowania w pierwszych siedmiu rozdziałach, Tom Cargill prezentuje praktyczne studium, w trakcie którego pojedynczy przykładowy program przechodzi kolejne transformacje, które pozwalają poprawić jego ogólną jakość przy jednoczesnym zredukowaniu wielkości kodu. Konkluzję książki stanowi rozdział poświęcony wielokrotnemu dziedziczeniu.

Książka Toma Cargilla to nie tylko cenne źródło wiedzy dla zaawansowanych programistów — przyda się ona również studentom informatyki i pokrewnych kierunków, zainteresowanych zdobyciem profesjonalnych umiejętności programistycznych.

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

Spis treści

Przedmowa......7

Wprowadzenie......9

1.

Abstrakcja......15

Przykład techniki programowania: ceny komputerów......15
Poszukiwanie wspólnego pojęcia abstrakcyjnego......19
Różnice pomiędzy klasami......22
Powtórnie wprowadzamy do programu dziedziczenie......26
Wyeliminowanie typów wyliczeniowych......27
Podsumowanie......30
Bibliografia......31
Ćwiczenie......31

2.

Spójność kodu......35

Przykład techniki programowania: klasa string......36
Precyzyjnie zdefiniowany stan obiektu......37
Spójność fizycznego stanu obiektu......38
Niezmienne warunki, czyli inwariancja klas......39
Konsekwentne stosowanie dynamicznego zarządzania pamięcią......41
Zwolnienie dynamicznie przyporządkowanej pamięci......43
Przykład techniki programowania: drugie podejście do tego samego problemu......44
Podsumowanie......50
Bibliografia......51
Ćwiczenia......51

3.

Zbędne dziedziczenie......55

Przykład techniki programowania: stos......55
Reguły widoczności nazw w procesie dziedziczenia......59
Relacje tworzone poprzez dziedziczenie......60

Enkapsulacja ... 65
Interfejs a implementacja ... 67
Zastosowanie szablonów ... 71
Podsumowanie ... 73
Bibliografia ... 73
Ćwiczenie ... 74

4.

Funkcje wirtualne ... 75

Przykład techniki programowania: problem pojazdów i garaży ... 75
Spójność kodu ... 79
Destruktory klas bazowych ... 81
Dziedziczenie ... 82
Sprzężenie ... 85
Podsumowanie ... 91
Bibliografia ... 92
Ćwiczenie ... 92

5.

Przeciążanie operatorów ... 93

Przeciążanie operatorów — podstawy ... 93
Przykład techniki programowania: tablica plików FileArray ... 98
Dziedziczenie implementacji ... 105
Programistyczny dylemat — jak lepiej: przeciążone operatory czy metody ... 110
Podsumowanie ... 112
Bibliografia ... 112
Ćwiczenia ... 112

6.

Klasy otaczające (ang. wrappers) ... 113

Biblioteka C ... 113
Przykład techniki programowania: klasa otaczająca dla struktury dirent w C++ ... 114
Wiele obiektów klasy Directory ... 116
Gdy zawodzi konstruktor ... 119
Publiczny dostęp do stanu obiektu ... 121
Dodatkowy argument służący obsłudze błędów ... 123
Podsumowanie ... 127
Bibliografia ... 128
Ćwiczenie ... 128

7.

Efektywność ... 129

Przykład techniki programowania: klasa BigInt ... 130
Egzaminujemy klasę BigInt ... 136
Długość dynamicznych łańcuchów znaków ... 138
Liczba dynamicznie zapamiętywanych łańcuchów znaków ... 140
Kod klienta ... 144
Modyfikujemy klasę BigInt ... 145

Podsumowanie ... 151
Bibliografia ... 152
Ćwiczenia ... 152

8.

Studium praktyczne ... 153

Przykład techniki programowania: Finite State Machine, czyli automat stanów skończonych ... 153
Zainicjowanie ... 158
Sprzężenia ... 165
Spójność ... 169
Moduły kontra abstrakcyjne typy danych ... 172
Wartość kontra zachowanie ... 175
Uogólnienie ... 180
Bibliografia ... 184
Ćwiczenia ... 185

9.

Dziedziczenie wielokrotne ... 187

Niejednoznaczności w mechanizmie wielokrotnego dziedziczenia ... 187
Skierowane niecykliczne grafy hierarchii dziedziczenia ... 189
Eksplorujemy wirtualne klasy bazowe ... 193
Przykład techniki programowania: klasa Monitor ... 200
Przykład techniki programowania: wirtualna klasa bazowa ... 204
Protokół powiadomień zwrotnych i przydatne wielokrotne dziedziczenie ... 210
Podsumowanie ... 213
Bibliografia ... 213
Ćwiczenia ... 213

10.

Zbiorcza lista reguł ... 215

Rozdział 1.: „Abstrakcja” ... 215
Rozdział 2.: „Spójność kodu” ... 215
Rozdział 3.: „Zbędne dziedziczenie” ... 216
Rozdział 4.: „Funkcje wirtualne” ... 216
Rozdział 5.: „Przeciążanie operatorów” ... 216
Rozdział 6.: „Klasy otaczające (ang. wrappers)” ... 216
Rozdział 7.: „Efektywność” ... 217
Rozdział 8.: „Studium praktyczne” ... 217
Rozdział 9.: „Dziedziczenie wielokrotne” ... 217

Skorowidz ... 219

3

Zbędne dziedziczenie

Chociaż w rozdziale 2. uważnie rozróżnialiśmy interfejs klasy i implementację klasy, nie czyniliśmy tego w kontekście dziedziczenia. Aby dokładnie zrozumieć relację dziedziczenia pomiędzy klasą pochodną a jej klasą bazową, musimy rozpatrywać część interfejsową i część implementacyjną klasy w sposób odrębny. W tym rozdziale przeanalizujemy przypadek, który z pozoru wydaje się być naturalną kandydaturą do zastosowania dziedziczenia. Mimo to, dokładniejsza analiza dziedziczenia i implementacji klas bazowej i pochodnej spowoduje w efekcie znaczącą modyfikację kodu.

Przykład techniki programowania: stos

Na listingu 3.1 pokazano program, w którym zostały zdefiniowane klasy `CharStack` (dosł. stos przeznaczony na znaki) oraz `IntStack` (dosł. stos przeznaczony na liczby typu `int`). Przeanalizujmy i oceńmy te klasy. Czy widać tu uogólnienie pojęć abstrakcyjnych? Czy dziedziczenie działa w sposób poprawny? Czy dziedziczenie zostało wykorzystane we właściwy sposób?

LISTING 3.1.
Oryginalna wersja klas Stack, CharStack oraz IntStack

```
#include <assert.h>
#include <string.h>

class Stack { // klasa bazowa "stos"
  int top;             // wierzchołek stosu
  int size;            // rozmiar
protected:             // dostępne tylko dla klas pochodnych
  void **vec;          // tablica wskaźników (wektor)
public:                // publiczny interfejs klasy bazowej
  Stack(int sz);       // konstruktor ogólny
  ~Stack();            // destruktor klasy bazowej
  void* push();        // metoda 'odłóż na stos'
  void* pop();         // metoda 'zdejmij ze stosu'
};
```

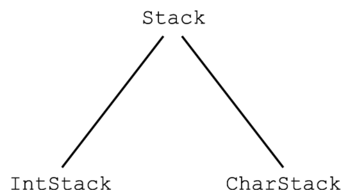
```
Stack::Stack(int sz)   // definicja konstruktora
{
  vec = new void*[size=sz];
  top = 0;
}

Stack::~Stack()
{
  delete [] vec;       // zwalnia pamięć zajmowaną przez tablicę wskaźników
}

void* Stack::push()
{
  assert(top < size);
  return vec[top++];
}

void* Stack::pop()
{
  assert(top > 0);
  return vec[--top];
}

const int defaultStack = 128;    // domyślny rozmiar stosu

class CharStack : public Stack {
  char *data;                    // prywatny wskaźnik do danych
public:
  CharStack();                   // konstruktor domyślny
  CharStack(int size);
  CharStack(int size, char *init);
  ~CharStack();

  void push(char);
  char pop();
};

CharStack::CharStack() : Stack( defaultStack )
{
  data = new char[defaultStack];
  for( int i=0; i<defaultStack; ++i)
    vec[i] = &data[i];
}

CharStack::CharStack(int size) : Stack(size)
{
  data = new char[size];
  for( int i=0; i<size; ++i)
    vec[i] = &data[i];
}
```

```
CharStack::CharStack(int size, char *init) : Stack(size)
{
  data = new char[size];
  for( int i=0; i<size; ++i)
    vec[i] = &data[i];
  for(i=0; i<strlen(init); ++i)
    *((char*) Stack::push()) = init[i];
}

CharStack::~CharStack()
{
  delete [] data;
}

void CharStack::push(char d)
{
  *((char*) Stack::push()) = d;
}

char CharStack::pop()
{
  return *((char*) Stack::pop());
}

class IntStack : public Stack {
  int *data;                // prywatny wskaźnik do danych
public:
  IntStack();               // konstruktor domyślny
  IntStack(int size);
  ~IntStack();

  void push(int);
  int  pop();
};

IntStack::IntStack() : Stack( defaultStack )
{
  data = new int[defaultStack];
  for( int i=0; i<defaultStack; ++i)
    vec[i] = &data[i];
}

IntStack::IntStack(int size) : Stack(size)
{
  data = new int[size];
  for( int i=0; i<size; ++i)
    vec[i] = &data[i];
}

IntStack::~IntStack()
{
```

```
  delete [] data;
}

void IntStack::push(int d)
{
  *((int*) Stack::push()) = d;
}

int IntStack::pop()
{
  return *((int*) Stack::pop());
}
```

Natychmiastowa, odruchowa reakcja wielu Czytelników na samo hasło `CharStack` oraz `IntStack` może polegać na stwierdzeniu, że takie klasy powinny zostać skonstruowane przy zastosowaniu typów sparametryzowanych, czyli szablonów (ang. *templates*), zgodnie z tym, co proponuje standard ANSI C++ (patrz: Ellis i Stroustrup). Ignorujemy tę możliwość i skoncentrujemy się na programie w takiej postaci, w jakiej został napisany. A to z dwóch powodów. Po pierwsze, ważne jest zrozumienie, przy wykorzystaniu najbardziej podstawowych konstrukcji języka C++, w jaki sposób powstała struktura tego programu, zanim jeszcze zaczniemy rozważać, w jaki sposób można by go udoskonalić, wykorzystując rozszerzenie języka. Pewne błędy w rozumowaniu, które doprowadziły do powstania tego programu, mogłyby łatwo wystąpić w bardziej kłopotliwej sytuacji, gdy użycie typów parametryzowanych (szablonów) nie dałoby nam żadnej alternatywy. Po drugie, w chwili pisania tej książki, implementacje typów parametryzowanych nie były jeszcze szeroko rozpowszechnione i dostępne. Co więcej, należało się liczyć z tym, że jeszcze przez pewien czas szablony nie zostaną poprawnie zaadaptowane do języka C++. Przedwczesne byłoby rozważanie technik programowania w odniesieniu do typów parametryzowanych przed nagromadzeniem stosownego praktycznego doświadczenia, co pozwoliłoby na poradnictwo, jak właściwie i efektywnie stosować te techniki. Wersja kodu napisana z zastosowaniem szablonów została podana na końcu tego rozdziału.

Istota tej konstrukcji kodu sprowadza się do tego, że obydwie klasy: `CharStack` oraz `IntStack` reprezentują stos (pierwsza dla znaków, druga dla liczb całkowitych), zatem obie mają wspólną klasę bazową `Stack` (po prostu „Stos", bez skonkretyzowania szczegółów). Taka hierarchia dziedziczenia została schematycznie przedstawiona na rysunku 3.1.



RYSUNEK 3.1. Hierarchia dziedziczenia w programie z listingu 3.1

Musimy popatrzeć na ten program znacznie uważniej, by dostrzec, że takie dziedziczenie nie jest niezbędne. Co więcej, okazuje się mylące i wręcz powinno zostać wyeliminowane. W istocie, zastosowanie dziedziczenia z specyfikatorem dostępu `public:` w tym programie tworzy niebezpieczną lukę w enkapsulacji takich stosów.

Reguły widoczności nazw w procesie dziedziczenia

Dziedziczenie może następować z użyciem specyfikatorów dostępu `public`, `private`, `protected`. Sposób dziedziczenia decyduje o widoczności i dostępności nazw (ang. *inheritance scope rules*). Publiczny interfejs klasy bazowej `Stack` jest następujący:

```
public:
  Stack(int sz);
  ~Stack();
  void* push();
  void* pop();
};
```

Patrząc na klasy pochodne, widzimy tam metody o tych samych nazwach (`pop()` oraz `push()`), co w klasie bazowej. Zwróćmy uwagę, że metody `Stack::push()` oraz `Stack::pop()` nie są funkcjami wirtualnymi. Zauważmy także, że typy argumentów tych metod w klasach pochodnych nie są zgodne z typami argumentów odpowiednich metod w klasie bazowej. Na przykład, metoda `Stack::push()` jest bezargumentowa, podczas gdy metoda `IntStack::push( int )` pobiera jeden argument typu `int`. W przypadku takich funkcji reguły widoczności nazw C++ mówią, że metoda zawarta w klasie pochodnej przesłania metodę o tej samej nazwie zawartą w klasie bazowej, ponieważ klasa pochodna wprowadza nowy poziom widoczności nazw (ang. *new scope level*). Rezultat tych reguł widoczności można wyraźnie dostrzec na następującym przykładzie:

```
class Base {      // klasa bazowa
public:
  void f(float);
  void g(float);
};
class Derived : public Base { // klasa pochodna
public:
  void f(int);
};

int main()
(
  Derived d;
  d.f(1.5);       // wywołanie:  Derived::f(1.5);
  d.g(1.5);       // wywołanie:  Base::g(1.5);
```

Wyraźnie widać, że w tym przypadku (przy dziedziczeniu) funkcja — metoda o nazwie `f()` nie jest przeciążona (ang. *not overloaded*). Klasa pochodna definiuje funkcję o nazwie `f()`, a zatem wyrażenie `d.f(1.5)` musi jednoznacznie wywołać metodę `f()` zdefiniowaną w klasie pochodnej, a nie w klasie bazowej. Natomiast wyrażenie `d.g(1.5)` wywołuje metodę z klasy bazowej, ponieważ w klasie pochodnej nie została zdefiniowana własna wersja metody o nazwie `g()`. Gdy kompilator C++ poszukuje w celu wywołania metody `g()`, najpierw przeszukiwana jest klasa pochodna, jeśli w klasie pochodnej nie zostanie odszukana żadna „pasująca" metoda, w drugiej kolejności poszukiwanie nastąpi w klasie bazowej. Zatem wywołanie metody `g()` wobec obiektu klasy pochodnej spowoduje w istocie zastosowanie wersji `Base::g()`z klasy bazowej, ale wywołanie metody `f()` będzie oznaczać `Derived::f()`, a nie `Base::f()`, która została przesłonięta i stała się w ten sposób niewidoczna.

Publiczny interfejs klasy IntStack jest następujący:

```
public:
  IntStack();
  IntStack(int size);
  ~IntStack();
  void push(int);
  int  pop();
};
```

Zgodnie z zasadami widoczności (dostępności) nazw opisanymi powyżej, kod klienta może manipulować obiektami klas IntStack lub CharStack przy użyciu wywołań ich własnych metod push() oraz pop(), a nie poprzez metody odziedziczone po klasie bazowej Stack. Klasy pochodne IntStack oraz CharStack dziedziczą publiczny interfejs po klasie bazowej Stack, ale odziedziczone metody przesłaniają własnymi wersjami tych funkcji. Powrócimy jeszcze do tego zagadnienia w dalszej części tego rozdziału.

Relacje tworzone poprzez dziedziczenie

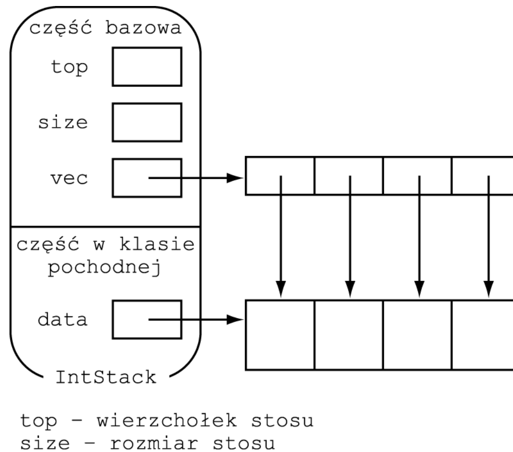
Przyjrzyjmy się na razie bliżej relacjom tworzonym poprzez dziedziczenie. Klasa bazowa Stack zapewnia, że w każdej jej klasie pochodnej będzie zaimplementowany mechanizm indeksujący (ang. *indexing service*). Odpowiedzialna jest za to jednowymiarowa tablica (czyli wektor) o nazwie vec. Tablica ta jest typu chronionego i składa się ze wskaźników do typu void. Klasa Stack zarządza także prywatnym polem danych int top; (wierzchołek stosu) pozwalającą na obsługę tablicy w odpowiedzi na wywołania metod push() oraz pop() z klas pochodnych. Klasy pochodne tworzą własne tablice — wektory wskazywane poprzez wskaźniki data dla wartości przechowywanych na stosie i inicjują te wektory tak, że dla wszystkich wartości indeksu tablicy i (oczywiście, w granicach wektora) element vec[i] jest wskaźnikiem do elementu data[i]. Wskaźniki typu void zwracane przez metody Stack::push() oraz Stack::push() mówią klasie pochodnej, na którym elemencie tablicy data mają być wykonywane odpowiednio operacje push (odłóż na stos) oraz pop (zdejmij ze stosu).

Struktura danych została przedstawiona na rysunku 3.2. Zwróćmy uwagę na spójność układu wskaźników. Taka struktura danych zawiera, niestety, niewiele informacji. Konstruktory klas pochodnych zawierają i wykonują instrukcję:

```
vec[i] = &data[i];
```

dla każdego z elementów wskazywanych w tablicy data[], aby poprawnie zainicjować wskaźniki wchodzące w skład tablicy vec[]. Po takim zadziałaniu konstruktora i zainicjowaniu, te wskaźniki nigdy nie ulegają zmianie. Klasa bazowa odsyła te wskaźniki z powrotem, by „powiedzieć” klasie pochodnej, gdzie (pod jakim adresem) znajdują się dane, do których można się odwoływać. Coś tu jest jednak nie tak. Chyba trochę za dużo wskaźników w porównaniu z niewielką ilością rzeczywistej, użytecznej informacji.

Gdy skoncentrujemy się na wymuszonej konwersji typu (ang. type *casting*) w kodzie metody Stack::pop(), zobaczymy tę samą sytuację z innej perspektywy.



RYSUNEK 3.2.
Relacja wskaźników z wektora vec do elementów tablicy data

```
int IntStack::pop()
{
  return *((int*) Stack::pop());
}
```

Ta wymuszona konwersja powoduje przekształcenie wskaźnika do typu `void` zwróconego przez metodę `Stack::pop()` na wskaźnik do typu `int` w celu odwołania się do elementu tablicy wskazywanej przez wskaźnik `IntStack::data`. Aby uniknąć niespodzianek i niebezpieczeństw kryjących się w mechanizmach rzutowania, zawsze lepiej programować bez wymuszania konwersji typu. Ta konwersja może być łatwo wyeliminowana. Gdy to zrobimy, wyjdzie na jaw problem, który sprawia wektor `vec`. Ogólnie rzecz biorąc, logika i kolejność działań jest tu następująca:

- metoda `Stack::pop()` zwraca wskaźnik `vec[i]`,
- wskaźnik `vec[i]` wskazuje element tablicy `data[i]`,
- metoda `IntStack::pop()` zwraca `data[i]`.

Żadna konwersja typów nie byłaby tu potrzebna, gdyby funkcja `Stack::pop()` zwracała sam indeks `i`, bezpośrednio, zamiast wskaźnika `vec[i]`. Mając wartość indeksu `i`, metoda `IntStack::pop()` bezpośrednio odwoływałaby się do elementu tablicy `data[i]`, bez żadnej konwersji typu. Metody `push()` oraz `pop()` należące do klasy bazowej `Stack` mogłyby zwracać pojedynczą liczbę całkowitą (indeks), której obiekt klasy pochodnej rzeczywiście potrzebuje, zamiast wskaźnika typu `void`, który przed użyciem musi zostać poddany konwersji typu. Taka prostsza wersja klasy `Stack` została pokazana na listingu 3.2. Zwróćmy uwagę, że metody `push()` oraz `pop()` należące do klas pochodnych `IntStack` oraz `CharStack` w tej wersji już nie stosują wymuszonej konwersji typu. Zauważmy także, że nazwa `Stack` została zmieniona na `StackIndex`, co lepiej opisuje to pojęcie abstrakcyjne.

LISTING 3.2.
Uproszczona klasa abstrakcji „Indeksacja stosu”

```
#include <assert.h>
#include <string.h>

class StackIndex {       // klasa bazowa "indeksacja stosu"
                         // prywatne pola danych
```

```
  int top;              // top - to 'wierzchołek stosu'
  int size;             // size - to rozmiar stosu
public:                 // publiczny interfejs klasy bazowej
  StackIndex(int sz);   // konstruktor ogólny
  ~StackIndex();        // destruktor klasy bazowej
  int  push();          // metoda 'odłóż na stos'
  int  pop();           // metoda 'zdejmij ze stosu'
};

StackIndex::StackIndex(int sz)
{
  size = sz;
  top = 0;
}

StackIndex::~StackIndex() {}     // "pusty" destruktor

int StackIndex::push()
{
  assert(top < size);
  return top++;
}

int StackIndex::pop()
{
  assert(top > 0);
  return --top;
}

const int defaultStack = 128;   // domyślny rozmiar stosu

class CharStack : public StackIndex {
  char *data;                   // prywatny wskaźnik do danych
public:
  CharStack();                  // konstruktor domyślny
  CharStack(int size);
  CharStack(int size, char *init);
  ~CharStack();

  void push(char);
  char pop();
};

CharStack::CharStack() : StackIndex( defaultStack )
{
  data = new char[defaultStack];
}

CharStack::CharStack(int size) : StackIndex(size)
{
  data = new char[size];
}
```

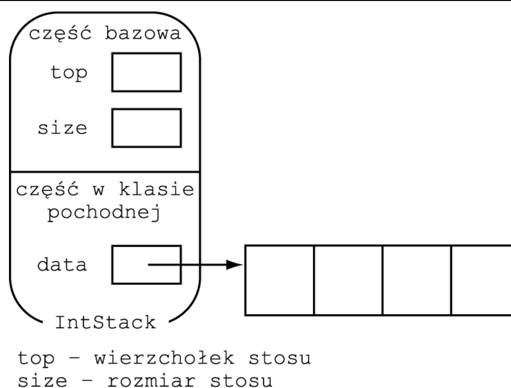
```
CharStack::CharStack(int size, char *init) : StackIndex(size)
{
  data = new char[size];
  for(int i=0; i<strlen(init); ++i)
    data[StackIndex::push()] = init[i];
}

CharStack::~CharStack()
{
  delete [] data;
}

void CharStack::push(char d)
{
  data[StackIndex::push()] = d;
}

char CharStack::pop()
{
  return data[StackIndex::pop()];
}

class IntStack : public StackIndex {
  int *data;          // prywatny wskaźnik do danych
public:
  IntStack();         // konstruktor domyślny
  IntStack(int size);
  ~IntStack();

  void push(int);
  int  pop();
};

IntStack::IntStack() : StackIndex( defaultStack )
{
  data = new int[defaultStack];
}

IntStack::IntStack(int size) : StackIndex(size)
{
  data = new int[size];
}

IntStack::~IntStack()
{
  delete [] data;
}

void IntStack::push(int d)
{
  data[StackIndex::push()] = d;
}
```

```
int IntStack::pop()
{
  return data[StackIndex::pop()];
}
```

Zastosowany tu nowy interfejs klasy bazowej `StackIndex` lepiej odpowiada abstrakcyjnemu pojęciu „indeksacja stosu", czyli tym funkcjom, które rzeczywiście zapewnia. Wspólna część abstrakcyjna — to obsługa indeksacji, mówiąca klasie pochodnej, gdzie znajdują się dane, do których można się odwoływać. Informacja o tym, w jaki sposób działa i zachowuje się stos, pozostaje ukryta przed klasą bazową `Stack`. Informacja o elementach znajdujących się rzeczywiście na poszczególnych stosach (obiektach klas pochodnych) zostaje udostępniona klasom pochodnym. Jedyna komunikacja następuje tu w zakresie indeksacji. Bez żadnego ograniczenia możliwości funkcjonalnych, taki abstrakcyjny stos może zostać zaimplementowany prościej. Jego zakres odpowiedzialności zostanie ograniczony do obsługi indeksacji w taki sposób, że inna, odrębna klasa może zapewnić mechanizm obsługi stosu.

► Poszukujmy prostych pojęć abstrakcyjnych.

Tablica wskaźników `Stack::vec` jest nadmiarowa (stanowi redundancję), dlatego zniknęła z nowej wersji klasy `StackIndex`. Każda z klas pochodnych takiej klasy bazowej otrzymuje wszystkie informacje, których potrzebuje w postaci indeksu zwracanego poprzez wywołane metody należące do klasy bazowej: `StackIndex::push()` oraz `StackIndex::pop()`. Nie ma żadnego racjonalnego uzasadnienia, by w klasie bazowej przechowywać wskaźniki do danych znajdujących się w tablicach deklarowanych w klasach pochodnych. Struktura danych wygląda teraz tak, jak pokazano na rysunku 3.3. Takie uproszczenie programu powoduje jednocześnie, że staje się on bardziej efektywny z punktu widzenia wykorzystania pamięci. Wyeliminowanie tablicy `Stack::vec` gwałtownie zmniejsza ilość pamięci wymaganą do utworzenia stosu. Przy typowej, 32-bitowej architekturze, gdy liczby całkowite typu `int` zajmują po 4 bajty pamięci każda i przy 4-bajtowych wskaźnikach, obiekt klasy `IntStack` zajmuje teraz na stercie dwukrotnie mniej pamięci, niż było to w wersji oryginalnej. Na każdy element przypadają 4 bajty zamiast 8. Obszar pamięci zajmowany na stercie przez obiekt klasy `CharStack` należy podzielić przez współczynnik 5. Zamiast 5 bajtów na każdy znak na stosie przypada teraz tylko 1 bajt.



RYSUNEK 3.3. Uproszczona struktura danych

Enkapsulacja

Taka nowa reprezentacja abstrakcyjnego pojęcia stosu jest prostsza, a sam program jest krótszy i wykorzystuje mniej pamięci. Relacje wprowadzone poprzez mechanizm dziedziczenia pomiędzy klasą bazową StackIndex a jej klasami pochodnymi nie zostały jeszcze do końca wyjaśnione. Wcześniej w tym rozdziale zasygnalizowano, że metody push() oraz pop() należące do klas pochodnych przesłaniają metody o tych samych nazwach należące do klasy bazowej. Jeśli jednak stosuje się pełne, rozwinięte nazwy funkcji, np.: StackIndex::push() i odpowiednio StackIndex::pop(), te metody mogą nadal być wywołane w zakresie widoczności nazw utworzonym przez klasy pochodne, przez obiekty klas IntStack lub CharStack. W istocie, metody StackIndex::push() oraz StackIndex::pop() są właśnie w taki sposób wywoływane w kodzie przez odpowiednie metody należące do klas pochodnych.

Metody z klasy bazowej, StackIndex::push() oraz StackIndex::pop(), mogą zostać wywołane nawet bez potrzeby sięgania do operatora widoczności ::. Jeśli do obiektu klasy pochodnej IntStack lub CharStack odwołujemy się za pośrednictwem wskaźnika do obiektu klasy bazowej StackIndex albo za pośrednictwem referencji do obiektu klasy StackIndex, to push() i pop() oznaczają wywołania metod należących do klasy bazowej. Problem dostępności metod należących do klas bazowych to bardzo poważne zagadnienie (a w tym przypadku — naruszenie) z punktu widzenia enkapsulacji stosów. Przy takiej konstrukcji programu możliwe jest zamierzone bądź omyłkowe wprowadzenie obiektu „stos” w stan niespójny logicznie poprzez odwołanie z innych części programu. Poniższa funkcja prezentuje trzy sposoby naruszenia hermetyzacji obiektu klasy IntStack.

```
void Violate()
{
  IntStack s;
  s.StackIndex::push();   // publiczne pole publicznej klasy bazowej
  StackIndex *p = &s;
  p->push();              // wywołanie StackIndex::push()
  StackIndex &r = s;
  r.push();               // wywołanie StackIndex::push()
}
```

Poprzez bezpośrednie wywołanie publicznych składowych klasy bazowej z kodu klienta, funkcja Violate() (dosł. „naruszająca”) zwiększa indeks stosu bez jednoczesnego dostarczenia jakiejkolwiek wartości przeznaczonej do umieszczenia na stosie. Efekt jest taki, że stos pozornie rośnie o jeden nowy element, ale skoro w chwili wykonania operacji „push” nie zostaje dostarczona żadna wartość do umieszczenia na stosie, to wartość zwracana przez metodę IntStack::pop() pozostaje nieokreślona. Ta wartość pozornie umieszczona na stosie pozostanie tą wartością, która znajdowała się w pamięci przeznaczonej dla danego elementu tablicy — poprzednio, w chwili wykonania operacji push.

Jak widać, kod klienta może bezpośrednio manipulować implementacją stosu, co może doprowadzić obiekt — stos — do stanu niezdefiniowanego. Enkapsulacja stosu (rozumiana jako jego odizolowanie od kodu klienta) zostaje naruszona. Ten problem nie powstał nagle, w momencie naszej modyfikacji kodu w celu wyeliminowania tablicy zawierającej wskaźniki w klasie bazowej. Ta luka w enkapsulacji występowała już w oryginalnym kodzie wyjściowym. Należałoby ją zlikwidować.

Jest tam jeszcze inny problem związany z dziedziczeniem. Destruktor klasy bazowej nie został zadeklarowany jako funkcja wirtualna. Jeśli obiekt klasy IntStack zostanie utworzony w sposób dynamiczny, a następnie zostanie usunięty operatorem delete, kasującym wskaźnik do klasy bazowej, to zostanie wywołany tylko destruktor klasy bazowej. Pod tym względem destruktory

zachowują się tak samo, jak inne metody. Rodzaj destruktora (tj. jego przynależność do określonej klasy) jest determinowany poprzez typ wskaźnika (typ wartości wskazywanej przez dany wskaźnik użyty w momencie deklaracji).

```
IntStack *p = new IntStack;
// ...
StackIndex *q = p;
// ...
delete q;      // spowoduje wywołanie tylko StackIndex::~StackIndex()
```

Skoro destruktor klasy pochodnej nie może zostać wywołany, programy (kod klienta) posługujące się klasą `IntStack` lub `CharStack` są potencjalnie zagrożone występowaniem „wycieków pamięci". Wyciek pamięci omawiany w rozdziale 2. na przykładzie klasy `string` występował dlatego, że pominięto pożądany operator `delete` w kodzie metody. W tym przypadku „wyciek pamięci" wystąpi nawet wtedy, gdy destruktor w klasie pochodnej będzie całkowicie poprawny. Operator `delete` użyty w kodzie destruktora klasy pochodnej może zostać wykonany tylko wtedy, gdy ten destruktor zostanie wywołany. Ten destruktor nie zostanie wywołany, jeśli obiekt klasy pochodnej będzie usuwany poprzez skasowanie wskaźnika do klasy bazowej (wskaźniki do klas bazowych mogą wskazywać także obiekty klas pochodnych i o taką sytuację tu chodzi). Jeśli wektor przydzielony dla danych stosu nie zostanie poprawnie usunięty, a jego pamięć zwolniona i zwrócona do systemu, w miarę działania programu będzie wzrastać ilość zajętej, choć bezużytecznej pamięci na stercie, co może doprowadzić nawet do przepełnienia pamięci przeznaczonej na stertę. Wycieki pamięci są trudno wykrywalne we wczesnych stadiach życia oprogramowania. Niewielkie testy mogą nie zaangażować wystarczająco dużej ilości pamięci, by można było uzyskać zauważalne rezultaty. W rozdziale 7. pokazano, w jaki sposób wyposażyć program w monitorowanie wykorzystania pamięci na stercie i jak dzięki temu wykrywać ewentualne wycieki pamięci.

Ten potencjalny wyciek pamięci może zostać wyeliminowany poprzez zadeklarowanie destruktora w klasie bazowej jako funkcji wirtualnej, choć jest to metoda doraźnego łatania dziury, która nie rozwiązuje rzeczywistego strukturalnego problemu występującego w tym kodzie źródłowym. W przypadku takich klas można znaleźć lepsze rozwiązania tego problemu. Do zagadnienia wirtualnych destruktorów w klasach bazowych wrócimy w rozdziałach 4. i 9.

Istnieją dwa środki zapobiegawcze. Każdy z tych środków pozwala na jednoczesne „wyleczenie" i problemu luki w enkapsulacji, i problemu wycieku pamięci. Pierwsze lekarstwo polega na tym, by `StackIndex` stała się prywatną klasą bazową. Dziedziczenie z użyciem specyfikatora dostępu `private:` zapobiega przenoszeniu (propagacji) publicznego interfejsu klasy bazowej na publiczne interfejsy klas pochodnych. W ten sposób wyklucza także tę możliwość, by wskaźnik do klasy bazowej lub referencja do klasy bazowej stały się referencjami (wskazaniami) do obiektów klas pochodnych. Jeśli `StackIndex` stanie się prywatną klasą bazową (innymi słowy, jeśli dziedziczenie będzie następować z użyciem specyfikatora `private:`), próba użycia funkcji takiej jak `Violate()` po stronie kodu klienta spowoduje wygenerowanie całej serii błędów kompilacji.

```
class CharStack : private StackIndex {
//...
};
class IntStack : private StackIndex {
//...
};
void Violate()
{
```

```
  IntStack s;
  s.StackIndex::push();    // komunikat o błędzie
  StackIndex *p = &s;      // komunikat o błędzie
  p->push();
  StackIndex &r = s;       // komunikat o błędzie
  r.push();
}
```

Próba odwołania się do prywatnych pól klasy bazowej jest formalnie nielegalna, podobnie jak próba wskazania obiektu klasy pochodnej za pomocą wskaźnika do klasy bazowej lub referencji do klasy bazowej. Jednocześnie problem z destruktorem został także skorygowany. Skoro wskaźnik do klasy bazowej formalnie nie może wskazywać obiektu klasy pochodnej, to niepożądany efekt działania operatora `delete` przy zastosowaniu takiego wskaźnika, jak opisano wyżej został wyeliminowany formalnie, „z definicji".

Interfejs a implementacja

Dlaczego i czy w ogóle musimy tu używać dziedziczenia? Jeśli przeanalizujemy dokładniej relacje dziedziczenia, przekonamy się, że to dziedziczenie może zostać całkowicie wyeliminowane. Drugim, alternatywnym rozwiązaniem może być zastosowanie obiektu, jako elementu klasy, zamiast dziedziczenia.

Jak opisano to w rozdziale 2., klasa w C++ składa się z dwóch głównych części: publicznego interfejsu, który charakteryzuje zachowanie się obiektów danej klasy oraz z prywatnej implementacji mechanizmów realizujących to zachowanie. Większość przypadków dziedziczenia to dziedziczenie po publicznej klasie bazowej (innymi słowy, z specyfikatorem `public:`). Klasa pochodna dziedziczy wtedy i interfejs, i implementację po klasie bazowej. Możliwe jest jednak również bardziej selektywne (wybiórcze) dziedziczenie, w którym klasa pochodna dziedziczy albo wyłącznie interfejs, albo wyłącznie implementację. Po prywatnej klasie bazowej (inaczej: przy dziedziczeniu z specyfikatorem `private:`) klasa pochodna dziedziczy całą (prywatną) implementację, ale równocześnie nie dziedziczy ani jednego elementu publicznego interfejsu klasy bazowej. Po publicznej, abstrakcyjnej klasie bazowej klasa pochodna dziedziczy cały interfejs, ale równocześnie odziedziczona implementacja może okazać się niekompletna lub niespójna.

Każde zastosowanie dziedziczenia powinno zostać poddane uważnej ocenie. Czy dziedziczony jest interfejs, implementacja, czy i jedno, i drugie? W tym programie klasy pochodne `CharStack` oraz `IntStack` dziedziczą tylko implementację. Interfejs klasy pochodnej `CharStack` jest podobny, lecz różni się jednak od interfejsu klasy pochodnej `IntStack`. Metody należące do obu tych klas pochodnych mają takie same nazwy, ale jedne operują na wartościach typu `char`, drugie na wartościach typu `int`. Klasy pochodne `CharStack` oraz `IntStack` nie w mają żadnym stopniu wspólnego interfejsu. Kod klienta nie może traktować obiektów tych klas w sposób jednakowy ani zamiennie. Choć jednak pojęcia abstrakcyjne reprezentowane poprzez klasy `IntStack` oraz `CharStack` stanowią specjalizację w stosunku do pojęć abstrakcyjnych odnoszących się do ogólnego pojęcia stosu, nie stanowią one specjalizacji w stosunku do abstrakcyjnego pojęcia „mechanizmu indeksacji" definiowanego przez ich klasę bazową `StackIndex`. Z punktu widzenia obiektów klienta każda spośród tych trzech klas: `IntStack`, `CharStack`, `StackIndex` oferuje odrębne usługi. Klasa `StackIndex` oferuje obsługę mechanizmu indeksacji, klasa `IntStack` oferuje zarządzanie stosem liczb całkowitych, a klasa `CharStack` umożliwia obsługę stosu znakowego. Relacja specyfikacji — uogólnienia, którą można by słownie wyrazić jako „A jest szczególnym przypadkiem B" (ang. „*is a kind of*") nie ma w odniesieniu

do tych klas sensu z punktu widzenia kodu klienta (użytkownika tych klas). Dlatego w tym przypadku dziedziczenia zastosowanie dziedziczenia publicznego interfejsu nie jest rozwiązaniem właściwym.

Relacja pomiędzy klasą pochodną a jej prywatną klasą bazową przypomina relację „klasa klienta-klasa serwera". Jeśli klasa bazowa `StackIndex` będzie prywatną klasą bazową, klasy pochodne `IntStack` oraz `CharStack` odziedziczą jej prywatną implementację, w celu wykorzystania jej mechanizmu obsługi indeksacji. W ten sposób klasy pochodne staną się „klientami", a klasa bazowa będzie ich „serwerem". Alternatywnym sposobem wyrażenia w C++ takiej relacji „klient-serwer", bez stosowania dziedziczenia, jest zagnieżdżenie prywatnej części (implementacji), czyli obiektu klasy `StackIndex` wewnątrz każdego obiektu klas `IntStack` oraz `CharStack`.

Zamiast pozostawać klasą bazową, `StackIndex` może stać się obiektem składowym wewnątrz `CharStack` oraz `IntStack`. To zmieni strukturę programu w bardzo niewielkim stopniu — pokazano to na listingu 3.3. Ta sama funkcjonalność w obu przypadkach jest realizowana poprzez obiekt takiego samego typu (klasy). Różnica polega na tym, że w tej wersji serwerem jest wewnętrzny obiekt należący do klasy, a nie prywatna część klasy bazowej. Zatem taki serwer jest identyfikowany poprzez nazwę (identyfikator) składowej klasy, a nie poprzez nazwę klasy. W tym konkretnym przypadku wybór jednego z tych rozwiązań: albo prywatnej klasy bazowej, albo prywatnego składowego obiektu wewnątrz klasy jest zdecydowanie kwestią gustu. Te dwie konstrukcje są z funkcjonalnego punktu widzenia równoważne. Mimo to, rozwiązanie stosujące obiekt składowy może być preferowane w celu uproszczenia kodu, ponieważ składnia i semantyka stosowania składowych obiektów jest banalnie prosta w porównaniu z semantyką stosowaną podczas dziedziczenia.

► Wykorzystajmy dziedziczenie do przekazania implementacji klasy bazowej, a nie jej interfejsu; w takich przypadkach stosujmy prywatne klasy bazowe albo (lepiej) obiekty składowe klas.

LISTING 3.3.
Obiekt jako składowa klasy zastępuje dziedziczenie

```
#include <assert.h>
#include <string.h>

class StackIndex {
  int top;
  int size;
public:
  StackIndex(int sz);
  int  push();
  int  pop();
};

StackIndex::StackIndex(int sz)
{
  size = sz;
  top = 0;
}

int StackIndex::push()
{
```

```
  assert(top < size);
  return top++;
}

int StackIndex::pop()
{
  assert(top > 0);
  return --top;
}

const int defaultStack = 128;

class CharStack {
  StackIndex index;
  char *data;
public:
  CharStack();
  CharStack(int size);
  CharStack(int size, char *init);
  ~CharStack();

  void push(char);
  char pop();
};

CharStack::CharStack() : index( defaultStack )
{
  data = new char[defaultStack];
}

CharStack::CharStack(int size) : index(size)
{
  data = new char[size];
}

CharStack::CharStack(int size, char *init) : index(size)
{
  data = new char[size];
  for(int i=0; i<strlen(init); ++i)
    data[index.push()] = init[i];
}

CharStack::~CharStack()
{
  delete [] data;
}

void CharStack::push(char d)
{
  data[index.push()] = d;
}
```

```
char CharStack::pop()
{
  return data[index.pop()];
}

class IntStack {
  StackIndex index;
  int *data;
public:
  IntStack();                 // konstruktor domyślny
  IntStack(int size);
  ~IntStack();

  void push(int);
  int  pop();
};

IntStack::IntStack() : index( defaultStack )
{
  data = new int[defaultStack];
}

IntStack::IntStack(int size) : index(size)
{
  data = new int[size];
}

IntStack::~IntStack()
{
  delete [] data;
}

void IntStack::push(int d)
{
  data[index.push()] = d;
}

int IntStack::pop()
{
  return data[index.pop()];
}
```

Przeciążanie funkcji w porównaniu ze stosowaniem domyślnych argumentów

Zanim przejdziemy do ostatniego przykładu, zwróćmy uwagę na podobieństwa w poddanych przeciążeniu funkcji (ang. *overloading*) konstruktorach w obydwu klasach `CharStack` oraz `IntStack`. Podobnie jak w przypadku przeciążonego konstruktora klasy `string`, opisanym w rozdziale 2.,

zastosowanie domyślnych wartości argumentów powinno było zastąpić stosowanie dwóch konstruktorów, czyli przeciążanie funkcji w podobny sposób, jak pokazano to wobec klasy CharStack na listingu 3.4. Dla każdej spośród tych klas wiele konstruktorów sprowadza się do jednego (bardziej złożonego) konstruktora, co upraszcza obsługę takiego kodu. Zastąpienie przeciążania funkcji użyciem domyślnych wartości argumentów nie zawsze jest takie proste, niemniej jednak — powinno zawsze zostać rozważone, jako rozwiązanie alternatywne. Przypomnijmy stosowną regułę:

► Rozważ użycie domyślnych wartości argumentów funkcji jako rozwiązanie alternatywne wobec przeciążania funkcji.

Na koniec, zwróćmy jeszcze uwagę, że konstruktor CharStack::CharStack na listingu 3.4 wywołuje funkcję biblioteczną strlen() podczas każdego cyklu iteracyjnego pętli programowej. Jeśli okaże się, że to ten właśnie konstruktor tworzy „wąskie gardło", powodując znaczące pogorszenie prędkości działania programu, usunięcie tego błędu powinno okazać się łatwe.

LISTING 3.4.
Zastosowanie domyślnych wartości argumentów pozwala zastąpić przeciążanie funkcji

```
class CharStack {
  StackIndex index;
  char *data;
public:
  CharStack(int size = defaultStack, char *init = "");
  ~CharStack();

  void push(char);
  char pop();
};

CharStack::CharStack(int size, char *init) : index(size)
{
  data = new char[size];
  for(int i=0; i<strlen(init); ++i)
    data[index.push()] = init[i];
}
```

Zastosowanie szablonów

Wspólne własności klas IntStack oraz CharStack mogą zostać wyrażone w sposób zupełnie inny przy użyciu szablonów w C++ (ang. *templates*), nazywanych także typami parametryzowanymi. Odpowiedni szablon klasy (ang. *class template*) dla klasy reprezentującej stos został pokazany na listingu 3.5.

Szablon klasy Stack definiuje rodzinę klas. Gdy szablon klasy Stack zostaje użyty do zadeklarowania obiektu, w deklaracji nazwa typu T musi zostać zastąpiony konkretnym typem danych. Na przykład, stos przeznaczony na elementy typu char:

```
Stack<char> stackOfChar(10);
```

LISTING 3.5.
Stos w postaci szablonu klasy Stack

```
const int defaultStack = 128;

template <class T>
class Stack {
  int size;
  int top;
  T *data;
public:
  Stack(int size = defaultStack);
  ~Stack();

  void push(T);
  T    pop(void);
};

template <class T>
Stack<T>::Stack(int s)
{
  size = s;
  top = 0;
  data = new T[size];
}

template <class T>
Stack<T>::~Stack()
{
  delete [] data;
}

template <class T>
Stack<T>::push(T d)
{
  assert(top<size);
  data[top++] = d;
}

template <class T>
T Stack<T>::pop()
{
  assert(top>0);
  return data[--top];
}
```

Powyższa deklaracja powoduje utworzenie obiektu o nawie `stackOfChar` będącego stosem do przechowywania maksymalnie 10 elementów typu `char`, natomiast:

```
Stack<int> stackOfInt(20);
```

spowoduje utworzenie obiektu o nazwie `stackOfInt`, czyli stosu do przechowywania maksymalnie 20 wartości typu `int`. Typ argumentu metody odkładającej na stos `push()` oraz typ wartości zwracanej metody zdejmującej ze stosu `pop()` są także określone za pomocą nazwy typu `T`.

Podstawowym i zasadniczym motywem, który spowodował dodanie szablonów do C++ była zapewniana przez szablony obsługa ogólnych klas — kolekcji. W ten sposób można utworzyć nie tylko stos liczb całkowitych i znaków, lecz równie dobrze można zbudować stosy złożone z liczb zmiennoprzecinkowych, ze wskaźników do znaków, itd.

Zachowanie się obiektów utworzonych na podstawie takiego szablonu klas w jeden, dość subtelny sposób różni się od tych obiektów, które były uprzednio tworzone na podstawie oryginalnych definicji klas `IntStack` oraz `CharStack`. Oryginalny konstruktor z klasy `CharStack` mógłby obierać drugi argument, który mógłby określać łańcuch znaków przeznaczony do umieszczenia (`push`) na stosie. Jednakże w klasie `IntStack` nie ma żadnego konstruktora, który mógłby pobierać taki (analogiczny) argument. Jeśli zastosujemy szablon do opisu obu tych klas jednocześnie, nie ma żadnego sposobu, by wyrazić takie zróżnicowanie.

Podsumowanie

Pierwszą i najważniejszą zmianą, której dokonaliśmy w pierwszym programie było zmodyfikowanie abstrakcyjnej konstrukcji zdefiniowanej poprzez klasę `Stack`. Oryginalny interfejs był zdefiniowany w kategoriach wskaźników do elementów tablicy użytkownika. Abstrakcyjna konstrukcja stosu została uproszczona poprzez wyrażenie interfejsu za pomocą innej kategorii — liczby całkowitej reprezentującej indeks stosu. Powstała w rezultacie tych modyfikacji nowa klasa, nazwana `StackIndex`, w sposób bardziej spójny i klarowny obejmowała ogólne własności stosu i jednocześnie pozwoliła zignorować nie mające istotnego znaczenia szczegóły techniczne reprezentacji. Ogólnie, im prostsze jest pojęcie abstrakcyjne, tym, po prostu, lepiej, dopóki tylko taka abstrakcyjna reprezentacja pozostaje wystarczająca. W przypadku tego programu, prostsza abstrakcja okazała się bardziej bezpieczna, ponieważ to uproszczenie pozwoliło wyeliminować liczne wymuszone konwersje typu. Stała się jednocześnie bardziej efektywna, ponieważ wyeliminowana została z programu tablica zawierająca wskaźniki do danych.

Drugą fundamentalną zmianą w tym programie, która pozwoliła na jego skorygowanie i wyeliminowanie luki w enkapsulacji stosu (wywołanej poprzez zastosowanie publicznego dziedziczenia, podczas gdy klasy pochodne potrzebowały tylko części implementacyjnej po klasie bazowej), było zastąpienie dziedziczenia. Prywatny, obiekt składowy klasy `StackIndex` okazał się dla klas `CharStack` oraz `IntStack` rozwiązaniem prostszym, a umożliwiającym im wykorzystanie funkcjonalności klasy `StackIndex`.

Bibliografia

Model „klient-serwer” został opisany w książce [2], Gorlena i in. oraz [3], Wirfsa-Brocka i in. Termin „klient” jest również stosowany w odniesieniu do programisty piszącego „kod klienta” (tj. kod użytkowy, posługujący się obiektami danych klas). Termin „serwer” (oznaczający generalnie „stronę usługową”, klasę, funkcję, itp.) bywa czasem zastępowany określeniem „usługodawca” (ang. *supplier*).

[1] Ellis M.A. Stroustrup B., *The Annotated C++ Reference Manual*. Reading, MA: Addison--Wesley, 1990.
[2] Gorlen K. E., Orlow S. M., Plexico P. S., *Data Abstraction and Object-Oriented Programming in C++*. Chichester, England: Wiley, 1990.
[3] Wirfs-Brock R., Wilkerson B., Wiener L., *Designing Object-Oriented Software*. Englewoods Cliffs, NJ: Prentice-Hall, 1990.

Ćwiczenie

3.1. Klasa `StackIndex` może być albo prywatną klasą bazową, albo posłużyć do utworzenia prywatnego obiektu składowego wewnątrz klas `IntStack` oraz `CharStack`. Skonstruuj taką sytuację, w której nie można by mieć takiego wyboru. Najpierw utwórz klasę korzystającą z usług innej klasy, która to ***musi*** zostać prywatną klasą bazową. Następnie utwórz taką klasę, w której potrzebna funkcjonalność ***musi*** zostać zaimplementowana w postaci prywatnego obiektu składowego wewnątrz tej klasy.