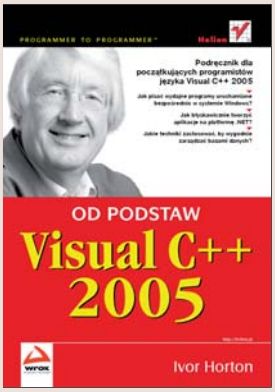
Visual C++ 2005. Od podstaw



Autor: Ivor Horton
Tłumaczenie: Łukasz Piwko (wstęp, rozdz. 1-10),
Marcin Rogóż (rozdz. 11-22, dod. A, B)
ISBN: 978-83-246-0652-8
Tytuł oryginału: Beginning Visual C++ 2005
Format: B5, stron: 1224
oprawa twarda

Podręcznik dla początkujących programistów języka Visual C++ 2005

- Jak pisać wydajne programy uruchamiane bezpośrednio w systemie Windows?
- Jak błyskawicznie tworzyć aplikacje na platformę .NET?
- Jakie techniki zastosować, by wygodnie zarządzać bazami danych?

C++ od czasu swego powstania cieszy się zasłużoną popularnością i nie mogło go zabraknąć wśród języków obsługiwanych przez środowisko Visual Studio, gdzie dostępne są dwie wersje tego języka. C++ ISO/ANSI pozwala pisać bardzo wydajne aplikacje, które można uruchamiać bezpośrednio w systemie Windows, podczas gdy C++/CLI to specyficzna dla Visual Studio odmiana umożliwiająca szybkie tworzenie rozbudowanych programów na platformę .NET.

Książka „Visual C++ 2005. Od podstaw" pozwoli Ci rozpocząć pracę z obiema wersjami języka Visual C++. Poznasz składnię wspólną dla obu odmian tego języka, a także specyficzne funkcje każdej z nich. Nauczysz się korzystać ze zintegrowanego środowiska programistycznego, które ułatwia pisanie, kompilowanie i diagnozowanie kodu. Dowiesz się też, jak obsługiwać i drukować dokumenty, zarządzać bazami danych czy przygotowywać własne biblioteki DLL.

- Praca w zintegrowanym środowisku programistycznym
- Składnia języków C++ ISO/ANSI i C++/CLI
- Programowanie obiektowe w Visual C++
- Diagnozowanie kodu
- Pisanie aplikacji dla platformy .NET
- Tworzenie oprogramowania dla systemu Windows
- Przechowywanie i drukowanie dokumentów
- Tworzenie własnych bibliotek DLL
- Praca z bazami danych

Naucz się korzystać z najnowszych technologii i narzędzi do tworzenia aplikacji dla systemu Windows

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

4

Tablice, łańcuchy znaków i wskaźniki

Poznaliśmy już wszystkie fundamentalne typy danych oraz posiadamy wiedzę na temat wykonywania obliczeń i podejmowania decyzji w programie. Rozdział ten poświęcony został szerszemu omówieniu zastosowań podstawowych technik, które poznaliśmy do tej pory. Zamiast pracować na pojedynczych danych, będziemy operować całymi ich zbiorami. W rozdziale tym dowiesz się:

- Czym są tablice i jak ich używać.
- Jak deklarować i inicjalizować tablice różnego typu.
- Jak deklarować tablice wielowymiarowe i jak ich używać.
- Czym są wskaźniki i jak ich używać.
- Jak deklarować i inicjalizować wskaźniki różnego typu.
- Co łączy tablice i wskaźniki.
- Czym są referencje, jak się je deklaruje, a także poznasz kilka podstawowych informacji na temat ich użycia.
- Jak dynamicznie przydzielać pamięć zmiennym w natywnym C++.
- Jak działa dynamiczne przydzielanie pamięci w programach CLR.
- Czym są uchwyty i odwołania śledzące oraz dlaczego potrzebujemy ich w programach CLR.
- Jak pracować z łańcuchami znaków i tablicami w programach w C++/CLI.
- Czym są wskaźniki wewnętrzne i jak się je tworzy oraz jak się ich używa.

W rozdziale tym będziemy znacznie częściej korzystali z obiektów, mimo że nie wiemy jeszcze, jak się je tworzy, ale jeśli nie wszystko jest całkiem jasne, nie musimy się tym przejmować. Bardziej szczegółowo na temat klas i obiektów będziemy mówić od rozdziału 7.

Obsługa wielu wartości danych tego samego typu

Wiemy już, jak zadeklarować i zainicjalizować różnego typu zmienne, z których każda przechowuje pojedynczy fragment informacji. Fragmenty takie będę nazywał **elementami danych**. Potrafimy stworzyć zmienną typu `char`, przechowującą pojedynczy znak, zmienną typu `short`, `int` i `long`, przechowującą jedną liczbę całkowitą, oraz zmienną typu `float` lub `double`, która przechowuje pojedynczą liczbę zmiennopozycyjną. Oczywistym krokiem naprzód jest nauczenie się operowania kilkoma elementami danych określonego typu za pomocą jednej nazwy zmiennej. Umiejętność taka umożliwiłaby nam znaczne rozszerzenie naszych możliwości.

Oto przykład prezentujący, w jakiej sytuacji moglibyśmy tego potrzebować. Przypuśćmy, że mamy napisać program obsługujący listę płac. Tworzenie zmiennej o nowej nazwie dla pensji czy informacji podatkowych każdego pracownika byłoby zadaniem co najmniej żmudnym. O wiele wygodniej byłoby móc odnosić się do tego pracownika za pomocą jednej nazwy generycznej, takiej jak np. `nazwaPracownika`, oraz mieć inne nazwy generyczne dla różnego rodzaju danych każdego z pracowników, jak pensja czy podatek itd. Oczywiście przydałaby się także możliwość odnalezienia określonego pracownika wśród wielu innych pracowników oraz wydobycia skojarzonych z nim danych generycznych, które są przechowywane w zmiennych. Tego typu potrzeba pojawia się za każdym razem, gdy mamy do czynienia ze zbiorem podmiotów, którymi chcemy zarządzać w programie — mogą to być zarówno piłkarze, jak i okręty wojenne. W C++ oczywiście dostępne są takie mechanizmy.

Tablice

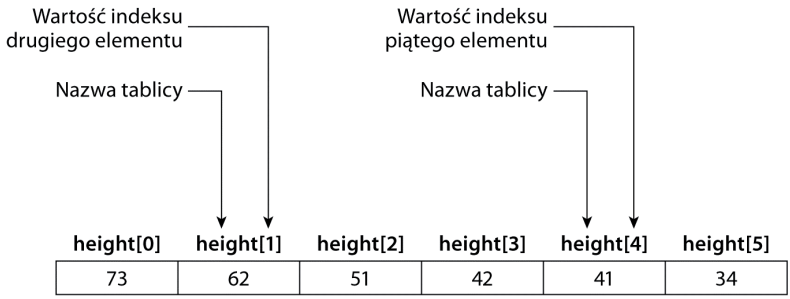
Podstawą rozwiązania wymienionych problemów w C++ ISO/ANSI są **tablice**. Tablica to po prostu zbiór miejsc w pamięci, zwanych **elementami tablicy** lub prościej **elementami**, z których każdy może przechowywać dane tego samego typu i do których odwołujemy się za pomocą tej samej nazwy zmiennej. Nazwy pracowników z listy płac można by było przechowywać w jednej tablicy, ich płace w drugiej, a należny podatek w trzeciej.

Poszczególne elementy w tablicy są określane przez wartość indeksową, która jest po prostu liczbą całkowitą i reprezentuje wszystkie elementy tablicy za pomocą kolejnych liczb, zaczynając od zera. Wartość indeksową elementu tablicy można sobie także wyobrazić jako wartość jego przesunięcia od pierwszego elementu tablicy. Pierwszy element ma wartość przesunięcia równą zero, a zatem jego indeks to 0. Wartość indeksowa 3 odnosi się zatem do czwartego elementu w tablicy. W przypadku listy płac można by było tak zaprojektować tablice, żeby wartości indeksowe odnoszące się do nazwy pracownika, jego płac oraz informacji podatkowych miały ten sam indeks w poszczególnych tablicach.

Podstawowa struktura tablicy została przedstawiona na rysunku 4.1.

Rysunek 4.1 przedstawia tablicę. Tablica o nazwie `height` zawiera sześć elementów, z których każdy przechowuje inną wartość. Mogą się one odnosić na przykład do wzrostu członków rodziny w porządku malejącym lub rosnącym. Jako że elementów jest sześć, to indeksy mają wartości od `0` do `5`. W celu odniesienia się do określonego elementu piszemy nazwę tablicy,

Rysunek 4.1



Tablica **height** zawiera 6 elementów

po której następuje wartość indeksowa wybranego elementu otoczona nawiasami kwadratowymi. Do trzeciego elementu tablicy odwołalibyśmy się za pomocą notacji `height[2]`. Jeżeli przyjmiemy, że indeks stanowi wartość przesunięcia względem pierwszego elementu, to z łatwością zauważymy, że indeks na przykład czwartego elementu wynosi `3`.

Ilość pamięci potrzebnej do przechowywania każdego elementu uzależniona jest od jego typu. Wszystkie elementy tablicy przechowywane są w sąsiadujących blokach pamięci.

Deklarowanie tablic

Zasadniczo tablice deklaruje się tak samo jak zmienne. Jedyna różnica polega na tym, że bezpośrednio po nazwie tablicy w nawiasach kwadratowych znajduje się liczba jej elementów. Można na przykład zadeklarować tablicę liczb całkowitych o nazwie `height`, którą widzieliśmy na poprzednim rysunku, za pomocą następującej instrukcji:

```
long height[6];
```

Jako że wartości typu `long` wymagają czterech bajtów pamięci, cała tablica zajmie ich 24. Tablice mogą być dowolnego rozmiaru, oczywiście w granicach określonych przez ilość pamięci dostępnej na danej platformie sprzętowej.

Tablice mogą być dowolnego typu. Aby na przykład utworzyć tablice do przechowywania informacji o pojemności i mocy silników, można napisać następujące instrukcje:

```
double cubic_inches[10]; // Pojemność silnika.
double horsepower[10]; // Moc silnika.
```

Jeżeli interesujesz się samochodami, to w tych tablicach możesz przechowywać informacje o pojemności i mocy do dziesięciu silników, do których można się odnosić za pomocą wartości indeksowych od 0 do 9. Podobnie jak w przypadku innych zmiennych, w jednej instrukcji można zadeklarować kilka tablic określonego typu, ale w praktyce prawie zawsze lepiej jest każdą deklarację umieszczać w oddzielnym wierszu.

spróbuj sam Używanie tablic

Przed przejściem do analizy kodu wyobraź sobie, że przy każdym tankowaniu zapisujesz ilość zakupionej benzyny do samochodu oraz liczbę przejechanych kilometrów. Możesz teraz napisać program analizujący te dane w celu sprawdzenia, jak wygląda zużycie paliwa za każdym razem, gdy je kupujesz:

```
// Cw4_01.cpp
// Obliczanie liczby kilometrów przejechanych na jednym baku.
#include <iostream>
#include <iomanip>

using std::cin;
using std::cout;
using std::endl;
using std::setw;

int main()
{
   const int MAX = 20;                    // Maksymalna liczba wartości.
   double gas[ MAX ];                     // Ilość benzyny w litrach.
   long miles[ MAX ];                     // Wskazania licznika przebiegu.
   int count = 0;                         // Licznik pętli.
   char indicator = 't';                  // Wskaźnik wprowadzania danych.

   while( (indicator == 't' || indicator == 'T') && count < MAX )
   {
      cout << endl
           << "Podaj ilość paliwa: ";
      cin >> gas[count];                  // Wczytaj ilość paliwa.
      cout << "Podaj dane z licznika przebiegu: ";
      cin >> miles[count];                // Wczytaj wartość z licznika przebiegu.

      ++count;
      cout << "Czy chcesz podać następne dane(t lub n)? ";
      cin >> indicator;
   }

   if(count <= 1)                         // count = 1 po ukończeniu wprowadzania
                                          // pierwszych danych.
   {                                      // ...musimy podać jeszcze jeden zestaw danych.
      cout << endl
           << "Potrzebne są co najmniej dwa zestawy danych. ";
      return 0;
   }

   // Wysyłanie na wyjście wyników od drugiego do ostatniego wpisu.
   for(int i = 1; i < count; i++)
   cout << endl
        << setw(2) << i << "."            // Numer sekwencji wysyłającej.
        << "Zakupiono = " << gas[i] << " litrów paliwa," // Wysyłanie informacji o paliwie.
        << "których wydajność to "                  // Informacja o kilometrach
                                                    // przejechanych na litrze.
        << (miles[i] - miles[i - 1])/gas[i] << " kilometrów na litr.";
```

```
    cout << endl;
    return 0;
}
```

Program zakłada, że za każdym razem napełniamy bak, a więc ilość zakupionego paliwa jest ilością zużytą na przejechanie podanego dystansu. Poniżej znajduje się przykładowy wynik działania tego programu:

```
Podaj ilość paliwa: 12.8
Podaj dane z licznika przebiegu: 25832
Czy chcesz podać następne dane (t lub n)? t

Podaj ilość paliwa: 14.9
Podaj dane z licznika przebiegu: 26337
Czy chcesz podać następne dane (t lub n)? t

Podaj ilość paliwa: 11.8
Podaj dane z licznika przebiegu: 26598
Czy chcesz podać następne dane (t lub n)? n

1. Zakupiono 14,9 litrów paliwa, których wydajność to 33,8926 kilometrów na litr.
2. Zakupiono 11,8 litrów paliwa, których wydajność to 22,1186 kilometrów na litr.
```

Jak to działa

Ponieważ w celu obliczenia liczby przejechanych kilometrów na zakupionym paliwie musimy obliczyć różnicę pomiędzy dwoma wskazaniami licznika przebiegu, to z pierwszej pary wprowadzonych danych pobieramy tylko wskazania tego licznika. Odrzucamy natomiast ilość zakupionego paliwa w pierwszym przypadku, ponieważ zostało ono zużyte wcześniej.

Podczas drugiego okresu pokazanego w wynikach musiały być straszne korki lub jeździliśmy, mając zaciągnięty hamulec ręczny. Rozmiary dwóch tablic `gas` oraz `miles`, użytych do przechowywania wprowadzonych danych, określa stała o nazwie `MAX`. Modyfikując wartość zmiennej `MAX`, zmieniamy maksymalną liczbę wprowadzanych danych. Technika ta jest często wykorzystywana do nadawania programowi elastyczności związanej z ilością obsługiwanych przez niego danych. Oczywiście cały kod programu musi zostać napisany z myślą o rozmiarach tablicy lub innych parametrów, których rozmiary są określane za pomocą zmiennych typu `const`. Jest to zadanie bardzo proste i nie ma powodu unikać takiego rozwiązania. Później dowiemy się jeszcze, jak przydzielać pamięć do przechowywania danych podczas działania programu, dzięki czemu nie ma potrzeby ustalania z góry ilości pamięci przydzielonej dla przechowywanych danych.

Wprowadzanie danych

Dane wczytywane są za pomocą pętli `while`. Jako że zmienna pętlowa `count` może działać od `0` do wartości `MAX-1`, użytkownik nie będzie mógł podać więcej wartości niż tablica może pomieścić. Zmienne `count` i `indicator` inicjalizujemy odpowiednio wartościami `0` i `t`, dzięki czemu pętla `while` wykonana zostanie co najmniej raz. Program prosi o podanie wymaganych danych, a podane wartości są umieszczane w odpowiednich elementach tablicy. Element użyty do przechowywania danej wartości jest określony przez zmienną `count`, której wartość za

pierwszym razem wynosi `0`. Element tablicy jest określony w instrukcji wejściowej `cin` przy użyciu zmiennej `count` jako indeksu. Następnie zmienna `count` zostaje zwiększona o jeden i jest gotowa do przyjęcia następnej wartości.

Po wprowadzeniu każdej wartości program żąda potwierdzenia zamiaru wprowadzenia następnej. Podany znak zostaje zapisany jako wartość zmiennej `indicator`, a następnie sprawdzany w warunku pętli. Działanie pętli zakończy się, jeżeli podany znak nie jest literą „t" lub „T", lub wartość zmiennej `count` jest większa niż wartość zmiennej `MAX`.

Po zakończeniu pętli wczytującej dane (bez względu na sposób) wartość zmiennej `count` jest o jeden większa niż wartość indeksowa ostatniego wprowadzonego do każdej tablicy elementu (pamiętamy, że zwiększamy ją po wprowadzeniu każdego nowego elementu). Sprawdzanie to dokonywane jest w celu upewnienia się, że podane zostały co najmniej dwie pary wartości. Jeżeli nie, program kończy się wyświetleniem odpowiedniej informacji, ponieważ do obliczenia wartości przejechanych kilometrów potrzebne są dwa wskazania licznika przebiegu.

Tworzenie wyników

Wyniki generowane są w pętli `for`. Zmienna kontrolna `i` przyjmuje wartości od `1` do `count - 1`, pozwalając na obliczanie przebiegu jako różnicy pomiędzy bieżącym elementem `miles[i]` oraz elementem poprzednim `miles[i-1]`. Zauważmy, że wartością indeksu może być wyrażenie, które w wyniku daje liczbę całkowitą, mogącą być indeksem w omawianej tablicy, czyli od zera do pomniejszonej o jeden liczby elementów w tablicy.

Jeżeli wartość wyrażenia indeksowego nie należy do zbioru prawidłowych indeksów dla elementów tablicy, to odniesiemy się do nieprawidłowej lokalizacji danych, która może zawierać inne, niepotrzebne dane lub nawet kod programu. Jeżeli odniesienie do takiego elementu pojawi się w wyrażeniu, to w obliczeniach użyjemy przypadkowych danych, przez co uzyskamy wynik, którego z pewnością się nie spodziewaliśmy. Jeśli przechowujemy wynik w elemencie tablicy korzystającym z niedozwolonej wartości indeksu, to nadpiszemy to, co znajduje się w tej lokalizacji. Jeżeli będzie to część naszego programu, to skutek będzie katastrofalny. Nieprawidłowe wartości indeksów nie są bowiem zgłaszane ani przez kompilator, ani podczas działania programu. Jedynym sposobem obrony przed nimi jest takie zaprojektowanie programu, aby zapobiegał takim sytuacjom.

Dane wyjściowe generowane są przez pojedynczą instrukcję wyjściową dla wszystkich wprowadzonych wartości poza pierwszą. Dla każdego wiersza wyników generowany jest także jego numer za pomocą zmiennej kontrolnej pętli `i`. Liczba kilometrów na litr obliczana jest bezpośrednio w wyrażeniu wyjściowym. Elementów tablic w wyrażeniach można używać dokładnie tak samo jak innych zmiennych.

Inicjalizacja tablic

Aby zainicjalizować tablicę w miejscu jej deklaracji, należy po nazwie tej tablicy postawić znak równości, za którym podajemy rozdzielaną przecinkami listę wartości początkowych otoczonych nawiasami klamrowymi. Poniżej znajduje się przykład jednoczesnej deklaracji i inicjalizacji tablicy:

```
int cubic_inches[5] = { 200, 250, 300, 350, 400 };
```

Tablica nazywa się `cubic_inches` i ma pięć elementów przechowujących liczby całkowite. Wartości podane podczas inicjalizacji odpowiadają następującym po sobie wartościom indeksów tablicy. W tym przypadku element `cubic_inches[0]` ma wartość `200`, `cubic_inches[1]` — `250`, `cubic_inches[2]` — `300` i tak dalej.

Nie można podać więcej wartości, niż tablica może pomieścić, ale można podać mniej. Jeżeli jest ich mniej, to wartości indeksów są przydzielane na zwykłych zasadach — pierwszy element ma indeks `0`, a następne kolejne indeksy. Elementy, dla których nie podaliśmy wartości początkowej, inicjalizowane są wartością zero. Nie jest to jednak równoznaczne z niepodaniem żadnej listy. Bez listy wartości inicjalizujących elementy tablicy zawierałyby przypadkowe wartości. Jeżeli podajemy listę wartości początkowych, to musimy podać w niej co najmniej jedną wartość. W przeciwnym przypadku kompilator zgłosi błąd. Przedstawię to na poniższym raczej ograniczonym przykładzie.

spróbuj sam Inicjalizowanie tablicy

```
// Cw4_02.cpp
// Inicjalizowanie tablicy.
#include <iostream>
#include <iomanip>

using std::cout;
using std::endl;
using std::setw;

int main()
{
   int value[5] = { 1, 2, 3 };
   int junk [5];

   cout << endl;
   for(int i = 0; i < 5; i++)
      cout << setw(12) << value[i];

   cout << endl;
   for(int i = 0; i < 5; i++)
      cout << setw(12) << junk[i];

   cout << endl;
   return 0;
}
```

W przykładzie tym deklarujemy dwie tablice. Pierwszą z nich — `value` — inicjalizujemy tylko częściowo, a drugiej — `junk` — nie inicjalizujemy w ogóle. Program generuje dwa wiersze, które na moim komputerze wyglądają następująco:

```
           1           2           3           0           0
-858993460 -858993460 -858993460 -858993460 -858993460
```

Drugi wiersz (odpowiadający wartościom od `junk[0]` do `junk[4]`) na każdym komputerze może wyglądać inaczej.

Jak to działa

Pierwsze trzy wartości tablicy `value` są wartościami inicjalizującymi, a pozostałe dwie mają wartości domyślne, czyli zero. W tablicy `junk` wszystkie wartości są przypadkowe, ponieważ nie podaliśmy ani jednej wartości początkowej. Elementy tej tablicy zawierają to, co zostawił program, który używał zajmowanych przez nie obszarów pamięci.

Aby w wygodny sposób zainicjalizować tablicę wartościami zerowymi, należy podczas inicjalizacji podać jedną wartość równą zero. Na przykład:

```
long data[100] = {0}; // Nadaj wszystkim elementom wartość początkową zero.
```

Powyższa instrukcja deklaruje tablicę o nazwie `data`, w której wszystkie 100 elementów będzie miało wartość zero. Pierwszy element został zainicjalizowany wartością w nawiasach kwadratowych, a pozostałe są również zainicjalizowane wartością zero, gdyż ich wartości nie zostały podane.

Można także pominąć rozmiar tablicy typu numerycznego, pod warunkiem że podamy wartości początkowe. Liczba elementów tablicy określona jest przez liczbę podanych wartości początkowych. Na przykład deklaracja tablicy:

```
int value[] = { 2, 3, 4 };
```

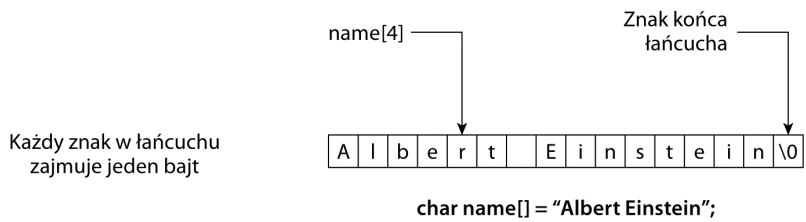
definiuje trzyelementową tablicę o wartościach początkowych `2`, `3` i `4`.

Tablice znakowe oraz obsługa łańcuchów

Tablica typu `char` zwana jest **tablicą znakową** i służy przede wszystkim do przechowywania łańcuchów znaków. Łańcuch znaków to zbiór znaków zakończony specjalnym znakiem określającym jego koniec. Sekwencja znaków oznaczająca koniec łańcucha zdefiniowana jest za pomocą kodu `\0` i czasami nazywa się ją **znakiem zerowym**, który jest bajtem ze wszystkimi bitami zerowymi. Taka forma znaku zerowego nazywana jest często łańcuchem w stylu C, ponieważ ten sposób definiowania łańcuchów został po raz pierwszy wprowadzony w języku C, od którego pochodzi C++, stworzony przez programistę Bjarne'a Stroustrupa. Nie jest to jedyna możliwa reprezentacja łańcucha — inne poznamy trochę później. Innej reprezentacji w szczególności używają programy w C++/CLI, a w bibliotece MFC zdefiniowana jest klasa `CString` służąca do reprezentowania łańcuchów.

Reprezentacja łańcucha w pamięci w stylu C widoczna jest na rysunku 4.2.

Rysunek 4.2



Rysunek 4.2 przedstawia wygląd łańcucha w pamięci oraz pokazuje sposób deklarowania łańcuchów, o którym za chwilę będziemy mówić.

Każdy znak w łańcuchu zajmuje jeden bajt, a więc każdy łańcuch zajmuje tyle pamięci, ile ma znaków plus jeden dla znaku zerowego.

Tablicę znakową można zadeklarować i zainicjalizować za pomocą literału łańcuchowego. Na przykład:

```
char movie_star[15] = "Marilyn Monroe";
```

Należy nadmienić, że kończący znak zerowy wstawiany jest automatycznie przez kompilator. Jeżeli dołączymy taki znak samodzielnie, to po kompilacji będziemy mieli dwa takie znaki. Musimy pozwolić na wstawianie znaku zerowego w liczbie elementów, które wprowadzamy do tablicy.

Można pozwolić kompilatorowi sprawdzić długość zainicjalizowanej tablicy za nas, jak widać na rysunku 4.1. Poniżej znajduje się jeszcze jeden przykład:

```
char president[] = "Ulysses Grant";
```

Jako że rozmiar tablicy nie został określony, kompilator przydziela wystarczająco pamięci, aby elementy mogły pomieścić inicjalizujący łańcuch wraz z kończącym znakiem zerowym. W tym przypadku pamięć zostaje przydzielona dla 14 elementów tablicy `president`. Oczywiście, jeżeli zechcemy później użyć tej tablicy do przechowywania innego łańcucha, to nie może on być dłuższy niż 14 bajtów (łącznie ze znakiem kończącym). Obowiązkiem programisty jest zapewnić taki rozmiar tablicy, aby mogła ona pomieścić każdy łańcuch, który chcemy w niej przechowywać.

Wprowadzanie łańcuchów znaków do programu

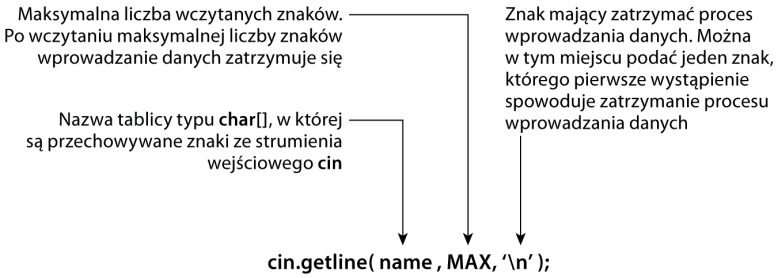
Plik nagłówkowy `<iostream>` zawiera definicje kilku funkcji służących do wczytywania znaków z klawiatury. Jedną z nich, o której będziemy teraz mówić, jest funkcja `getLine()`, przyjmująca sekwencje znaków wprowadzonych za pomocą klawiatury i przechowująca je w tablicy znakowej jako łańcuch zakończony znakiem `\0`. Typowe instrukcje przy użyciu tej funkcji wyglądają następująco:

```
const int MAX = 80;              // Maksymalna długość łańcucha włącznie z \0.
char name[MAX];                  // Tablica do przechowywania łańcucha.
cin.getline(name, MAX, '\n'); // Wczytanie danych wejściowych jako łańcucha.
```

Powyższe instrukcje najpierw deklarują tablicę znakową o nazwie `name`, która może pomieścić `MAX` liczbę elementów, a następnie za pomocą funkcji `getLine()` przyjmują znaki ze strumienia wejściowego. Jak widać, źródło danych (`cin`) zostało zapisane z dwukropkiem oddzielającym je od nazwy funkcji. Znaczenie argumentów funkcji `getLine` pokazano na rysunku 4.3.

Jako że ostatnim argumentem funkcji `getLine()` jest znak `\n` (znak nowego wiersza lub końca wiersza), a drugim argumentem jest `MAX`, znaki są wczytywane aż do napotkania znaku `\n` lub wczytania liczby znaków równej `MAX-1`, w zależności od tego, co pierwsze nastąpi. Maksymalna liczba wczytanych znaków wynosi `MAX-1`, a nie `MAX`, aby pozostawić miejsce dla znaku

Rysunek 4.3



zerowego, który zostanie dołączony do sekwencji znaków przechowywanych w tablicy. Znak `\n` generowany jest w momencie naciśnięcia klawisza *Enter* i jest zazwyczaj najwygodniejszym sposobem zakończenia wprowadzanych danych. Zmieniając ostatni argument, można jednak określić coś jeszcze. Znak `\n` nie jest przechowywany w tablicy `name`, ale — jak już mówiłem — `\0` dodawany jest na końcu łańcucha wejściowego w tej tablicy.

Więcej na temat tej składni dowiemy się przy okazji omawiania klas. Na razie musisz przyjąć do wiadomości, że takie coś istnieje, bowiem użyjemy tego w przykładzie.

spróbuj sam Programowanie z zastosowaniem łańcuchów

Mamy już teraz odpowiednio dużo wiedzy, abyśmy mogli napisać prosty program pobierający łańcuch znaków i liczący, z ilu znaków się on składa.

```
// Cw4_03.cpp
// Liczenie znaków w łańcuchu.
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main()
{
   const int MAX = 80;                  // Maksymalny rozmiar tablicy.
   char buffer[MAX];                    // Bufor danych wejściowych.
   int count = 0;                       // Licznik znaków.

   cout << "Wprowadź łańcuch znaków składający się z mniej niż 80 znaków:\n";
   cin.getline(buffer, MAX, '\n');      // Wczytuj łańcuch aż do napotkania \n.

   while(buffer[count] != '\0')         // Zwiększaj licznik, dopóki
      count++;                          // bieżący znak nie jest zerowy.

   cout << endl
        << "Łańcuch \"" << buffer
        << "\" zawiera " << count << " znaków.";

   cout << endl;
   return 0;
}
```

Przykładowy wynik działania tego programu może być następujący:

```
Wprowadź łańcuch składający się z mniej niż 80 znaków:
Promieniowanie zabija geny

Łańcuch "Promieniowanie zabija geny" zawiera 26 znaków.
```

Jak to działa

Program ten deklaruje tablicę znakową o nazwie `buffer` i przyjmuje łańcuch znaków z klawiatury do tablicy po uprzednim wyświetleniu prośby o podanie łańcucha. Wczytywanie z klawiatury kończy się w momencie naciśnięcia przycisku *Enter* lub gdy liczba wczytanych znaków będzie równa `MAX - 1`.

Pętla `while` służy do liczenia liczby wprowadzanych znaków. Jej działanie kontynuowane jest, dopóki bieżący znak, do którego odwołaniem jest `buffer[count]`, nie jest znakiem `\0`. Ten rodzaj sprawdzania bieżącego znaku podczas przechodzenia przez tablicę jest często spotykaną techniką w C++. Jedyną czynnością wykonywaną w pętli jest zwiększanie zmiennej `count` za każdym razem, gdy znak nie jest znakiem zerowym.

Istnieje też funkcja biblioteczna `strlen()`, która może zaoszczędzić nam fatygi samodzielnego pisania kodu. Jeśli chcemy jej użyć, musimy dołączyć plik nagłówkowy `<cstring>` do programu za pomocą dyrektywy `#include`, jak widać poniżej:

```
#include <cstring>
```

Litera `c` w nazwie pliku nagłówkowego oznacza, że zawiera on definicje należące do biblioteki języka C, który częściowo tworzy standardową bibliotekę C++. Nagłówek ten zawiera również funkcję `wcsnlen()`, która zwraca długość szerokiego łańcucha znaków.

Za pomocą funkcji `strlen()`moglibyśmy zastąpić pętlę `while` następującą instrukcją:

```
count = std::strlen(buffer);
```

Jako argument podaliśmy nazwę tablicy zawierającej łańcuch. Funkcja `strlen()` zwraca długość takiego łańcucha w postaci liczby całkowitej bez znaku typu `siez_t`. Wiele funkcji biblioteki standardowej zwraca wartości typu `size_t`, który jest zdefiniowany w bibliotece standardowej przy użyciu instrukcji `typedef` jako ekwiwalent jednego z typów fundamentalnych — najczęściej `int` bez znaku. Powodem używania typu `size_t` zamiast bezpośrednio jednego z typów fundamentalnych jest fakt, że pozwala on na pewną elastyczność co do tego, jaki jest rzeczywisty typ w różnych implementacjach C++. Standard C++ pozwala na urozmaicenie zestawu wartości należących do typów fundamentalnych w celu maksymalnego wykorzystania architektury sprzętu, a typ `size_t` może być zdefiniowany jako ekwiwalent najodpowiedniejszego typu fundamentalnego w bieżącym środowisku sprzętowym.

Na końcu przykładowego kodu za pomocą jednej instrukcji wyjściowej zostają wyświetlone łańcuch oraz liczba znaków. Zwróć uwagę na kod znaku specjalnego, który został użyty w celu wyświetlenia podwójnego cudzysłowu.

Tablice wielowymiarowe

Tablice z jednym indeksem, które definiowaliśmy do tej pory, zwane są tablicami **jednowymiarowymi**. Tablica może jednak posiadać więcej indeksów niż jeden i w takim przypadku nazywa się ona tablicą **wielowymiarową**. Przypuśćmy, że mamy pole, na którym hodujemy fasolę w rządkach po 10 krzaków, i że pole zawiera 12 takich rządków (tak więc w sumie mamy 120 krzaków). Moglibyśmy zadeklarować tablicę, do której zapisywalibyśmy wagę fasoli zebranej z każdego krzaka za pomocą następującej instrukcji:

```
double beans[12][10];
```

Powyższa instrukcja deklaruje dwuwymiarową tablicę o nazwie `beans`. Pierwszy jej indeks odpowiada numerom rządków, a drugi numerom krzaków w tych rządkach. Aby odnieść się do określonego elementu tej tablicy, potrzebujemy dwóch indeksów. Aby na przykład ustawić wartość elementu odpowiadającego piątemu krzakowi w trzecim rządku, posłużylibyśmy się następującą instrukcją:

```
beans[2][4] = 10.7;
```

Pamiętajmy, że wartości indeksów rozpoczynają się od zera, dlatego indeks rządka to `2`, a indeks piątego krzaka to `4`.

Jako że jesteśmy bogatymi rolnikami, mamy jeszcze kilka takich pól fasoli. Zakładając, że mamy ich osiem, moglibyśmy utworzyć tablicę trójwymiarową do zapisywania danych:

```
double beans[8][12][10];
```

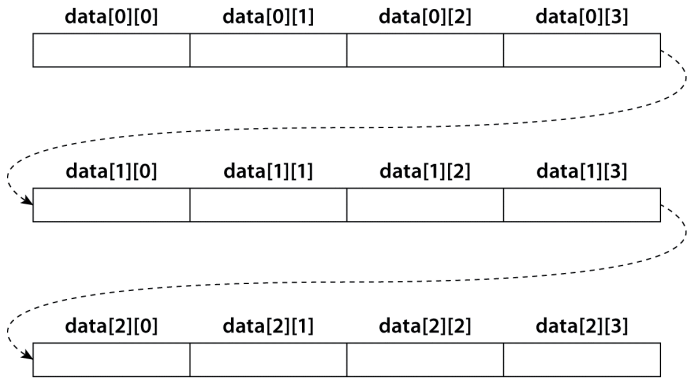
Powyższa tablica przechowuje dane dotyczące wszystkich krzaków fasoli na wszystkich polach. Pierwszy indeks z lewej odnosi się do określonego pola. Jeżeli uda nam się rozwinąć hodowlę do rozmiarów międzynarodowych, możemy użyć tabeli czterowymiarowej. W czwartej tabeli przechowywalibyśmy nazwy poszczególnych państw. Zakładając, że jesteśmy tak samo dobrymi sprzedawcami, jak rolnikami, produkowanie tak dużych ilości fasoli w celu nadążenia za popytem może mieć niekorzystny wpływ na dziurę ozonową.

Tablice przechowywane są w pamięci w taki sposób, że wartość indeksu znajdująca się najdalej po prawej stronie jest zmieniana najwcześniej. A zatem tablica `data[3][4]` składa się z trzech jednowymiarowych tablic, z których każda zawiera cztery elementy. Schemat tej tablicy widoczny jest na rysunku 4.4.

Elementy tej tablicy przechowywane są w przylegających blokach pamięci, jak wskazują strzałki na rysunku 4.4. Pierwszy indeks odnosi się do określonego wiersza w tablicy, a drugi do określonego elementu w tym wierszu.

Zauważmy, że tablica dwuwymiarowa w natywnym C++ jest w rzeczywistości tablicą jednowymiarową zawierającą tablice jednowymiarowe. Tablica trójwymiarowa w natywnym C++ jest w rzeczywistości jednowymiarową tablicą elementów, które stanowią jednowymiarowe tablice tablic jednowymiarowych. W większości przypadków nie trzeba się tym wszystkim przejmować, ale jak później zobaczymy, tablice w C++/CLI nie są takie same jak w natywnym C++. Z powyższego wynika również, że wyrażenia `data[0]`, `data[1]` oraz `data[2]` reprezentują tablice jednowymiarowe.

Rysunek 4.4



Elementy tablicy przechowywane są
w przylegających do siebie sektorach pamięci

Inicjalizowanie tablic wielowymiarowych

Do inicjalizacji tablicy wielowymiarowej używa się rozszerzonej wersji metody użytej do inicjalizacji tablic jednowymiarowych. Aby na przykład zainicjalizować dwuwymiarową tablicę o nazwie `data`, możemy posłużyć się następującą deklaracją:

```
long data[2][4] = {
                   { 1, 2, 3, 5 },
                   { 7, 11, 13, 17 }
                  };
```

Jak widać, wartości początkowe każdego wiersza tablicy znajdują się pomiędzy parą własnych nawiasów klamrowych. Jako że w każdym wierszu są cztery elementy, w każdej grupie znajdują się cztery wartości początkowe, a ponieważ mamy dwa wiersze, to mamy też dwie grupy wartości w nawiasach klamrowych. Każda grupa wartości początkowych oddzielona jest od następnej przecinkiem.

Wartości początkowe w każdym wierszu można pominąć. W takim przypadku zostaną im przypisane wartości zerowe. Na przykład:

```
long data[2][4] = {
                     { 1, 2, 3 },
                     { 7, 11 }
                   };
```

Zastosowałem dodatkowe spacje pomiędzy wartościami początkowymi, aby pokazać, gdzie wartości zostały pominięte. Elementy `data[0][3]`, `data[1][2]` oraz `data[1][3]` nie mają wartości początkowych, a więc zostają ustawione na zero.

Gdybyśmy chcieli wartości wszystkich elementów tablicy ustawić na zero, to moglibyśmy napisać:

```
long data[2][4] = {0};
```

Inicjalizując tablice o większej liczbie wymiarów, należy pamiętać, że będzie potrzeba tyle zagnieżdżonych grup wartości początkowych w nawiasach klamrowych, ile jest wymiarów w tablicy.

spróbuj sam Przechowywanie wielu łańcuchów

Do przechowywania kilku łańcuchów w stylu C możemy użyć jednej dwuwymiarowej tablicy. Jak tego dokonać, prześledzimy na poniższym przykładzie:

```
// Cw4_04.cpp
// Przechowywanie łańcuchów w tablicy.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

int main()
{
   char stars[6][80] = { "Robert Redford",
                         "Hopalong Cassidy",
                         "Lassie",
                         "Slim Pickens",
                         "Boris Karloff",
                         "Oliver Hardy"
                       };
   int dice = 0;

   cout << endl
        << " Wybierz szczęśliwą gwiazdę!"
        << " Podaj cyfrę od 1 do 6: ";
   cin >> dice;

   if(dice >= 1 && dice <= 6)                  // Sprawdź poprawność wprowadzanych danych.
      cout << endl                             // Wyświetl dane gwiazdy.
           << "Twoja szczęśliwa gwiazda to " << stars[dice - 1];
   else
      cout << endl                             // Wprowadzono nieprawidłowe dane.
           << "Przykro mi, ale nie masz swojej szczęśliwej gwiazdy.";

   cout << endl;
   return 0;
}
```

Jak to działa

Poza ogromną wartością rozrywkową, najbardziej interesującą częścią tego kodu jest deklaracja tablicy `stars`. Jest to dwuwymiarowa tablica elementów typu `char`, która może przechowywać do sześciu łańcuchów składających się maksymalnie z 80 znaków każdy (włącznie z kończącym znakiem zerowym dodawanym automatycznie przez kompilator). Łańcuchy inicjalizujące zostały umieszczone w nawiasach klamrowych i oddzielone od siebie przecinkami.

Jedną z wad korzystania w takich przypadkach z tablic jest fakt, że zarezerwowana pamięć jest prawie nieużywana. Wszystkie nasze łańcuchy znaków mają mniej niż 80 znaków, przez co nadwyżkowe elementy w każdym wierszu są marnowane.

Sprawdzenie liczby łańcuchów możemy również pozostawić kompilatorowi, pomijając pierwszy wymiar tablicy i pisząc następującą deklarację:

```
char stars[][80] = { "Robert Redford",
                     "Hopalong Cassidy",
                     "Lassie",
                     "Slim Pickens",
                     "Boris Karloff",
                     "Oliver Hardy"
};
```

Taki zapis zmusi kompilator do zdefiniowania pierwszego wymiaru w taki sposób, aby mógł on pomieścić liczbę łańcuchów inicjalizujących, które określiliśmy. Jako że mamy sześć łańcuchów, to wynik będzie dokładnie taki sam, ale unikamy ryzyka wystąpienia błędu. Nie możemy jednak ominąć obu wymiarów tablicy. W przypadku tablic dwu- lub wielowymiarowych ostatnia tablica po prawej zawsze musi być zdefiniowana.

Zwróć uwagę na średnik znajdujący się na samym końcu deklaracji. Łatwo o nim zapomnieć, kiedy podaje się wartości inicjalizujące tablicy.

Chcąc uzyskać dostęp do łańcucha w poniższej instrukcji w celu wysłania go na wyjście, musimy podać tylko pierwszy indeks:

```
cout << endl                                          // Wyślij dane gwiazdy.
     << "Twoja szczęśliwa gwiazda to " << stars[dice - 1];
```

Pojedynczy indeks wybiera określoną 80-elementową podtablicę, a instrukcja wyjściowa wysyła jej zawartość do momentu napotkania znaku kończącego. Indeks został określony jako `dice - 1`, ponieważ wartości `dice` zawierają się w zbiorze 1 – 6, a wartości indeksów muszą należeć do zbioru 1 – 5.

Pośredni dostęp do danych

Zmienne, o których była mowa do tej pory, stanowią nazwany fragment pamięci, gdzie można przechowywać dane określonego typu. Ich zawartość może pochodzić ze źródła zewnętrznego, takiego jak np. klawiatura, oraz wewnętrznego, czyli wyników obliczeń dokonanych z udziałem innych wprowadzonych wartości. W C++ istnieje jeszcze jeden typ zmiennych, które nie przechowują danych wprowadzonych lub obliczonych w normalny sposób, ale znacznie zwiększają elastyczność programów i nadają im większą moc. Ten typ zmiennej zwany jest **wskaźnikiem**.

Czym jest wskaźnik

Każda lokalizacja w pamięci, w której przechowywane są jakieś dane, ma swój adres. Za pomocą adresów urządzenia komputerowe odnoszą się do poszczególnych elementów danych. Wskaźnik to zmienna przechowująca adres innej zmiennej określonego typu. Ma on — podobnie jak wszystkie inne zmienne — swoją nazwę, a także typ określający, do jakiego rodzaju

danych się odnosi. Należy pamiętać, że typ zmiennej wskaźnikowej zawiera informację, że jest to wskaźnik. Zmienna, która jest wskaźnikiem mogącym zawierać adresy lokalizacji w pamięci, przechowujących wartości typu `int`, jest **wskaźnikiem do** `int`.

Deklarowanie wskaźników

Deklaracja wskaźnika wygląda podobnie do deklaracji zwykłej zmiennej, ale przed nazwą wskaźnika znajduje się jeszcze gwiazdka informująca, że ta zmienna jest właśnie wskaźnikiem. Aby na przykład zadeklarować wskaźnik typu `long` o nazwie `pnumber`, można posłużyć się następującą instrukcją:

```
long* pnumber;
```

Powyższa deklaracja została zapisana z gwiazdką przy nazwie typu. Można ją także napisać jak poniżej:

```
long *pnumber;
```

Dla kompilatora nie stanowi to żadnej różnicy, ale typ zmiennej `pnumber` to wskaźnik do typu `long`, co jest często oznaczane poprzez wstawienie gwiazdki bezpośrednio po nazwie typu. Bez względu na wybrany sposób zapisywania typu wskaźnika należy trzymać się konsekwentnie jednego.

Deklaracje zwykłych zmiennych i wskaźników można mieszać w jednej instrukcji. Na przykład:

```
long* pnumber, number = 99;
```

Powyższa instrukcja — tak jak poprzednio — deklaruje wskaźnik o nazwie `pnumber`, który jest wskaźnikiem do typu `long`, a także zmienną `number` typu `long`. Wskaźniki lepiej deklarować jednak oddzielnie od innych zmiennych, ponieważ można się łatwo pomylić co do typu zadeklarowanych zmiennych, w szczególności gdy lubimy stawiać * przy nazwie typu. Poniższe instrukcje z pewnością są bardziej przejrzyste, a dodatkowo — dzięki umieszczeniu ich w oddzielnych wierszach — można do każdego z nich dodać oddzielny komentarz, co z kolei powoduje, że kod programu jest łatwiejszy do czytania.

```
long number = 99; // Deklaracja i inicjalizacja zmiennej typu long.
long* pnumber;    // Deklaracja wskaźnika typu long.
```

Rozpoczynanie nazw wskaźników od litery `p` (ang. *pointer* — wskaźnik) w C++ jest często stosowaną konwencją. Dzięki temu łatwiej się zorientować, które zmienne w programie są wskaźnikami, co znacznie ułatwia rozszyfrowywanie, co robi dany kod.

Prześledźmy, jak to działa, na przykładzie, nie zastanawiając się, do czego on służy. Do sposobów użycia wskaźników przejdziemy niebawem. Przypuśćmy, że mamy zmienną typu `long` o nazwie `number`, którą zadeklarowaliśmy powyżej, inicjalizując wartością 99. Mamy także wskaźnik `pnumber` (wskaźnik do typu `long`), którego możemy używać do przechowywania adresu zmiennej `number`. Ale w jaki sposób uzyskać adres zmiennej?

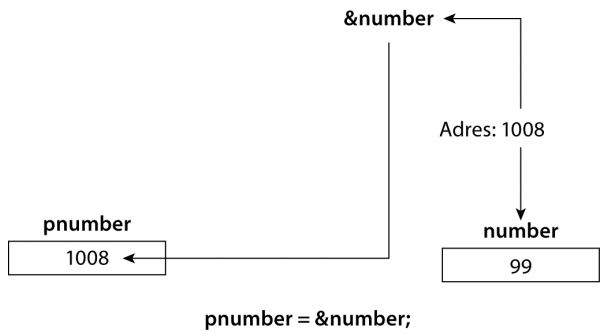
Operator adresowania

To, czego potrzebujemy, to **operator adresowania** — &. Jest to operator jednoargumentowy, który sprawdza adres zmiennej. Jest on także zwany operatorem referencji, ale o tym będziemy mówić trochę później. Aby ustawić wskaźnik, o którym przed chwilą mówiłem, możemy posłużyć się następującą instrukcją:

```
pnumber = &number; // Zapisz adres zmiennej number do pnumber.
```

Rezultat tej operacji został przedstawiony na rysunku 4.5.

Rysunek 4.5



Operatora & można użyć do sprawdzenia adresu dowolnej zmiennej, ale do jego przechowywania potrzebujemy wskaźnika odpowiedniego typu. Jeśli na przykład chcemy przechować adres zmiennej typu `double`, to wskaźnik musi być zadeklarowany jako typ `double*`, czyli wskaźnik do typu `double`.

Używanie wskaźników

Sprawdzenie adresu zmiennej i zapisanie go do wskaźnika jest zadaniem ciekawym, ale najbardziej interesujące jest to, do czego możemy go użyć. Podstawą używania wskaźników jest chęć uzyskania dostępu do danych w zmiennej, na którą one wskazują. Dokonuje się tego za pomocą **operatora pośredniości** *.

Operator pośredniości

Operatora pośredniości * używamy ze wskaźnikiem do uzyskania dostępu do zawartości zmiennej, na którą on wskazuje. Nazwa operator pośredniości pochodzi od tego, że dostęp do danych odbywa się w sposób pośredni. Inna nazwa tego operatora to operator wyłuskania, a proces uzyskiwania dostępu do danych w zmiennej za pomocą wskaźnika nazywa się operacją **wyłuskania**.

Jednym z mylących aspektów tego operatora może być fakt, że mamy już kilka różnych zastosowań tego samego symbolu *. Służy on jako operator mnożenia, operator pośredniości, a także używa się go w deklaracjach wskaźników. Kompilator potrafi wywnioskować z kontekstu, w jakiej funkcji występuje symbol *. Jeżeli na przykład mnożymy dwie zmienne `A*B`, to nie ma możliwości zinterpretowania tego wyrażenia w jakikolwiek inny sposób.

Do czego służą wskaźniki

Jednym z pytań, które najczęściej przychodzą do głowy w takich chwilach, jest: „Do czego one mogą się przydać?”. Bo przecież sprawdzanie adresu zmiennej, którą znamy, i zapisywanie go do wskaźnika w celu późniejszego jego wyłuskania może wydawać się czynnością niepotrzebną. Jest jednak kilka powodów, dla których wskaźniki są ważne.

Jak się niebawem przekonamy, notacji wskaźnikowej można używać do operowania na danych przechowywanych w tablicach — jest ona zazwyczaj szybsza niż tradycyjna notacja tablicowa. Kiedy dojdziemy do definiowania własnych funkcji, dowiemy się, że wskaźniki bardzo często używane są do uzyskiwania dostępu z wnętrza funkcji do dużych bloków danych (takich jak tablice) zdefiniowanych poza nimi. I najważniejsze — później dowiemy się także, że pamięć zmiennym można przydzielać dynamicznie podczas wykonywania programu. Ta możliwość pozwala programowi na dostosowanie poziomu zużycia pamięci w zależności od ilości wprowadzanych do niego danych. Ponieważ nie wiemy z góry, ile zmiennych będziemy tworzyli dynamicznie, najlepszym sposobem na zrobienie tego są wskaźniki, a więc warto zapoznać się z nimi trochę lepiej.

spróbuj sam Używanie wskaźników

Różne aspekty operacji z udziałem wskaźników prześledzimy na poniższym przykładzie:

```
// Cw4_05.cpp
// Ćwiczenie użycia wskaźników.
#include <iostream>
using std::cout;
using std::endl;
using std::hex;
using std::dec;

int main()
{
   long* pnumber = NULL;           // Deklaracja i inicjalizacja wskaźnika.
   long number1 = 55, number2 = 99;

   pnumber = &number1;             // Zapisywanie adresu do wskaźnika.
   *pnumber += 11;                 // Zwiększenie zmiennej number1 o 11.
   cout << endl
        << "number1 = " << number1
        << "   &number1 = " << hex << pnumber;

   pnumber = &number2;             // Zmiana wskaźnika na adres zmiennej number2.
   number1 = *pnumber*10;          // Pomnożenie zmiennej number2 przez 10.

   cout << endl
        << "number1 = " << dec << number1
        << "   pnumber = " << hex << pnumber
        << "   *pnumber = " << dec << *pnumber;

   cout << endl;
   return 0;
}
```

Na moim komputerze powyższy program generuje następujący wynik:

```
number1 = 66            &number1 = 0012FEC8
number1 = 990           pnumber = 0012FEBC          *pnumber = 99
```

Jak to działa

W tym przykładzie nie wprowadzamy żadnych danych. Wszystkie operacje odbywają się na wartościach początkowych zmiennych. Po zapisaniu adresu zmiennej `number1` do wskaźnika `pnumber` jej wartość zostaje zwiększona pośrednio poprzez wskaźnik w poniższej instrukcji:

```
*pnumber += 11;                  // Zwiększenie zmiennej number1 o 11.
```

Zauważ, że podczas pierwszej deklaracji wskaźnika `pnumber` zainicjalizowaliśmy go wartością `NULL`. Inicjalizowanie wskaźników będziemy niebawem omawiali.

Operator pośredniości określa, że do wartości zmiennej `number1`, wskazywanej przez wskaźnik `pnumber`, dodajemy `11`. Gdybyśmy zapomnieli w tej instrukcji wstawić operator *, to oznaczałoby to, że chcemy dodać `11` do adresu przechowywanego we wskaźniku.

Wartość zmiennej `number1` oraz jej adres przechowywany we wskaźniku `pnumber` zostały wyświetlone. Manipulatora `hex` użyliśmy w celu wysłania na wyjście adresu w notacji szesnastkowej. Jeżeli chcemy, aby następne dane były wysłane znowu w notacji dziesiętnej, to w następnej instrukcji wyjściowej musimy skorzystać z manipulatora `dec` w celu przestawienia trybu wysyłania z powrotem na dziesiętny.

Po pierwszym wierszu danych wyjściowych zawartość wskaźnika `pnumber` jest ustawiana na adres zmiennej `number2`. Następnie wartość zmiennej `number1` zostaje pomnożona przez 10:

```
number1 = *pnumber*10;           // Pomnożenie zmiennej number2 przez 10.
```

Obliczenie to zostaje wykonane za pomocą uzyskania dostępu do zawartości zmiennej `number2` pośrednio poprzez wskaźnik. Rezultat tych obliczeń widoczny jest w drugim wierszu danych wyjściowych.

Adresy, które widzimy w wysłanych na wyjście danych, mogą być za każdym razem inne, gdyż są one zależne od obszaru pamięci, w którym został załadowany program, a to z kolei uzależnione jest od konfiguracji systemu operacyjnego. Łańcuch `0x` przed adresami oznacza, że są to liczby szesnastkowe. Zauważmy, że adresy `&number1` oraz `pnumber` (zawierający `&number2`) różnią się o cztery bajty. Pokazuje to, że zmienne `number1` i `number2` zajmują w pamięci miejsca obok siebie, jako że każda zmienna typu `long` zajmuje cztery bajty. Z wysłanych danych wynika, że wszystko działa tak, jak można się było tego spodziewać.

Inicjalizowanie wskaźników

Używanie niezainicjalizowanych wskaźników jest bardzo ryzykowne, ponieważ możemy za ich pomocą łatwo nadpisać losowe lokalizacje pamięci. Rozmiar szkód uzależniony jest od tego, jak dużego mieliśmy pecha, a zatem inicjalizowanie wskaźników jest czymś więcej niż tylko dobrym pomysłem. Wskaźnik można łatwo zainicjalizować wartością będącą adresem

już istniejącej zmiennej. W poniższym przykładzie zainicjalizowałem wskaźnik pnumber adresem zmiennej number za pomocą operatora & z nazwą zmiennej:

```
int number = 0;                    // Zainicjalizowana zmienna typu całkowitego.
int* pnumber = &number;            // Zainicjalizowany wskaźnik.
```

Inicjalizując wskaźnik adresem zmiennej, należy pamiętać, że zmienna ta musi być zadeklarowana przed deklaracją wskaźnika.

Oczywiście możemy wcale nie chcieć inicjalizować wskaźnika adresem określonej zmiennej podczas jego deklaracji. W takim przypadku możemy go zainicjalizować wartością zerową. W Visual C++ do tego celu służy symbol NULL, który został zdefiniowany jako 0. A zatem wskaźnik można zadeklarować i zainicjalizować za pomocą poniższej instrukcji, zamiast używać sposobu z ostatniego przykładu:

```
int* pnumber = NULL;               // Wskaźnik, który na nic nie wskazuje.
```

Dzięki temu mamy pewność, że wskaźnik nie będzie zawierał adresu, który może zostać uznany za prawidłowy, a wartość tego wskaźnika można sprawdzić za pomocą następującej instrukcji if:

```
if(pnumber == NULL)
   cout << endl << "Wskaźnik pnumber ma wartość zerową.";
```

Wskaźnik można oczywiście zainicjalizować jawnie wartością 0, co również daje nam pewność, że nie zostanie mu przypisana żadna przypadkowa wartość. Żaden obiekt nie może mieć adresu 0, a więc użycie tej wartości jako wartości wskaźnika oznacza, że na nic on nie wskazuje. Poza faktem, że kod może stracić nieco na czytelności, jeśli chcemy go kompilować także w innych kompilatorach, to lepiej jest stosować wartość 0 do inicjalizacji wskaźnika, który ma na nic nie wskazywać.

Podejście to jest także zgodne z dobrym stylem programowania w C++ ISO/ANSI. Argument za nim przemawiający to stanowisko, że każdy nazwany obiekt w C++ powinien posiadać swój typ, a NULL nie ma określonego typu — jest aliasem zera. Jak przekonamy się w dalszej części rozdziału, w C++/CLI rzecz ta przedstawia się trochę inaczej.

Aby zainicjalizować wskaźnik wartością 0, wystarczy napisać:

```
int* pnumber = 0;                  // Wskaźnik, który na nic nie wskazuje.
```

W celu sprawdzenia, czy wskaźnik zawiera prawidłowy adres, można posłużyć się następującą instrukcją:

```
if(pnumber == 0)
   cout << endl << "Wskaźnik pnumber ma wartość zerową. ";
```

Równie dobrze moglibyśmy posłużyć się instrukcją:

```
if(!pnumber)
    cout << endl << "Wskaźnik pnumber ma wartość zerową.";
```

Powyższa instrukcja robi dokładnie to samo co poprzednia.

Możemy oczywiście zastosować następującą formę:

```
if(pnumber != 0)
   // Wskaźnik ma prawidłową wartość, a więc użyjmy go.
```

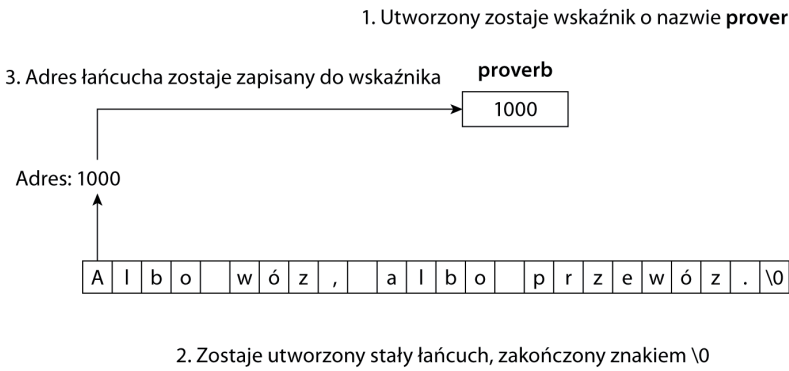
Wskaźniki do typu char

Wskaźnik do typu `char` ma ciekawą właściwość — może być inicjalizowany za pomocą literału znakowego. Wskaźnik taki możemy na przykład zadeklarować i zainicjalizować za pomocą następującej instrukcji:

```
char* proverb = "Albo wóz, albo przewóz.";
```

Powyższa instrukcja wygląda podobnie do inicjalizacji tablicy znakowej, ale nie jest identyczna. Tworzy literał łańcuchowy (w rzeczywistości jest to tablica typu `const char`) z łańcucha znaków znajdującego się w cudzysłowach i zakończonego znakiem `/0` oraz przechowuje adres tego literału we wskaźniku `proverb`. Adresem literału będzie adres jego pierwszego znaku. Przedstawiono to na rysunku 4.6.

Rysunek 4.6



spróbuj sam Szczęśliwe gwiazdy ze wskaźnikami

Aby zobaczyć, jak działają wskaźniki, przepiszemy przykład ze szczęśliwymi gwiazdami, w którym używaliśmy tablic, korzystając tym razem ze wskaźników:

```
// Cw4_06.cpp
// Inicjalizowanie wskaźników za pomocą łańcuchów znaków.
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main()
{
   char* pstr1 = "Robert Redford";
   char* pstr2 = "Hopalong Cassidy";
   char* pstr3 = "Lassie";
   char* pstr4 = "Slim Pickens";
```

```
    char* pstr5 = "Boris Karloff";
    char* pstr6 = "Oliver Hardy";
    char* pstr  = "Twoja szczęśliwa gwiazda to ";

    int dice = 0;

    cout << endl
         << " Wybierz szczęśliwą gwiazdę!"
         << " Podaj cyfrę od 1 do 6: ";
    cin >> dice;

    cout << endl;
    switch(dice)
    {
       case 1: cout << pstr << pstr1;
               break;

       case 2: cout << pstr << pstr2;
               break;

       case 3: cout << pstr << pstr3;
               break;

       case 4: cout << pstr << pstr4;
               break;

       case 5: cout << pstr << pstr5;
               break;

       case 6: cout << pstr << pstr6;
               break;

       default: cout << "Nie masz swojej szczęśliwej gwiazdy.";
    }

    cout << endl;
    return 0;
}
```

Jak to działa

Tablica z przykładu *Cw4_04.cpp* została zastąpiona sześcioma wskaźnikami od `pstr1` do `pstr6`. Każdy z nich został zainicjalizowany jakimś imieniem i nazwiskiem. Zadeklarowany został również jeszcze jeden dodatkowy wskaźnik — `pstr`. Został on zainicjalizowany zdaniem, które chcemy wysłać na początku normalnego wiersza danych wyjściowych. Jako że dysponujemy oddzielnymi wskaźnikami, łatwiej jest wybrać odpowiednią wiadomość wyjściową za pomocą `switch` niż za pomocą instrukcji `if`, z której skorzystaliśmy w pierwszej wersji przykładu. Wszystkimi nieprawidłowymi danymi, które zostały wprowadzone do programu, zajmuje się opcja instrukcji `switch default`.

Wysłanie na wyjście łańcucha wskazywanego przez wskaźnik nie mogłoby już być łatwiejsze. Jak widać, wystarczy tylko podać nazwę wskaźnika. W tej chwili może zrodzić się pytanie, dlaczego w przykładzie *Cw4_05.cpp*, kiedy napisaliśmy nazwę wskaźnika w instrukcji wyjściowej, wyświetlony został adres, na który wskazywał ten wskaźnik. Dlaczego tutaj jest ina-

czej? Odpowiedź leży w sposobie traktowania wskaźników do typu `char` przez instrukcję wyjściową. Wskaźniki tego typu są przez nią traktowane w specjalny sposób — jako łańcuchy (które są tablicami znaków), a więc instrukcja wysyła na wyjście sam łańcuch, a nie jego adres.

Zastosowanie wskaźników w powyższym przykładzie wyeliminowało problem z marnowaniem pamięci, który wystąpił przy użyciu tablic. Jednak program wydaje się teraz trochę rozwlekły — musi być na to jakiś lepszy sposób, i rzeczywiście jest — użycie tablicy wskaźników.

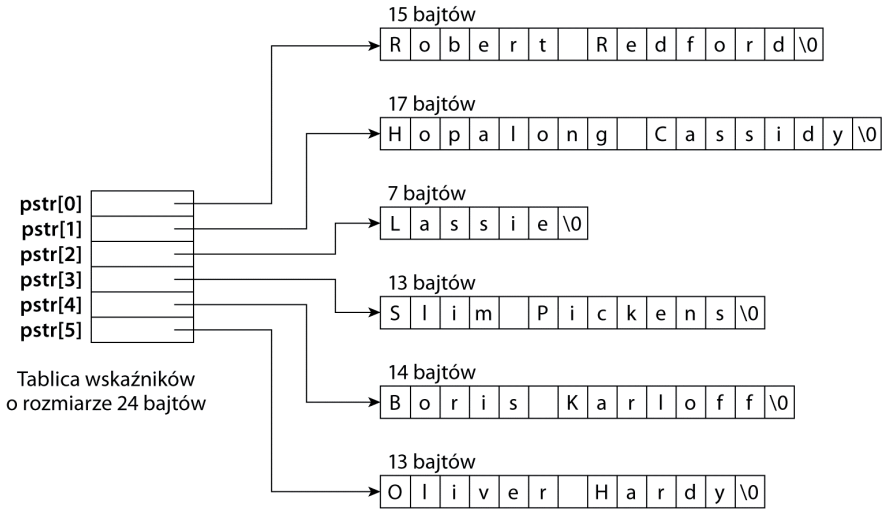
spróbuj sam Tablice wskaźników

W tablicy wskaźników do typu `char` każdy element może wskazywać na niezależny łańcuch znaków, a długość każdego z tych łańcuchów może być inna. Tablicę wskaźników deklaruje się w podobny sposób jak zwykłą tablicę. Przejdźmy teraz do przepisania poprzedniego przykładu przy użyciu tablicy wskaźników:

```
// Cw4_07.cpp
// Inicjalizowanie wskaźników łańcuchami znaków.
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main()
{
   char* pstr[] =  { "Robert Redford",       // Inicjalizowanie tablicy wskaźników.
                     "Hopalong Cassidy",
                     "Lassie",
                     "Slim Pickens",
                     "Boris Karloff",
                     "Oliver Hardy"
                   };
   char* pstart = "Twoja szczęśliwa gwiazda to ";

   int dice = 0;

   cout << endl
        << " Wybierz swoją szczęśliwą gwiazdę!"
        << " Podaj cyfrę od 1 do 6: ";
   cin >> dice;

   cout << endl;
   if(dice >= 1 && dice <= 6)                    // Sprawdzanie poprawności
                                                 // wprowadzanych danych.
      cout << pstart << pstr[dice - 1];          // Wyświetl dane gwiazdy.

   else
      cout << "Przykro mi, ale nie masz swojej szczęśliwej gwiazdy."; // Wprowadzono
                                                                      // nieprawidłowe dane.

   cout << endl;
   return 0;
}
```

Jak to działa

W tym przypadku stosujemy najbardziej efektywne podejście. Mamy jednowymiarową tablicę wskaźników do typu `char` zadeklarowaną w taki sposób, że kompilator sam ustala jej rozmiar na podstawie liczby łańcuchów inicjalizujących. Wykorzystanie pamięci w tym przypadku pokazano na rysunku 4.7.



Rysunek 4.7

W porównaniu z normalną tablicą tablica wskaźników zużywa mniej pamięci. W zwykłej tablicy wszystkie wiersze musiałyby być takiej samej długości jak najdłuższy łańcuch. Sześć wierszy zajmujących po 17 bajtów zajmuje ich w sumie 102. Dzięki użyciu tablicy wskaźników zaoszczędziliśmy cały 1 bajt. Co poszło nie tak? Odpowiedź jest prosta — przy niewielkiej liczbie względnie krótkich łańcuchów znaków rozmiar dodatkowej tablicy wskaźników odgrywa duże znaczenie. Oszczędności zaczęłyby się, gdybyśmy używali więcej dłuższych łańcuchów i mieli bardziej zróżnicowaną długość zmiennych.

Oszczędność pamięci to nie jedyna korzyść ze stosowania wskaźników. W wielu przypadkach oszczędzamy również czas. Pomyślmy, co by było, gdybyśmy chcieli przesunąć łańcuch `"Oliver Hardy"` na pierwszą pozycję, a łańcuch `"Robert Redford"` na ostatnią. Mając tablicę wskaźników, wystarczy tylko zmienić miejsca występowania wskaźników — łańcuchy pozostają tam, gdzie były. W przypadku zwykłej tablicy, której użyliśmy w przykładzie *Cw4_04.cpp*, musielibyśmy dużo kopiować — łańcuch `"Robert Redford"` trzeba by przenieść tymczasowo w jakieś inne miejsce, następnie skopiować łańcuch `"Oliver Hardy"` na jego miejsce i na zakończenie przenieść łańcuch `"Robert Redford"` na ostatnią pozycję. Wykonanie tych wszystkich operacji zajmuje znacznie więcej czasu.

Ze względu na fakt, że nasza tablica ma nazwę `pstr`, nazwa zmiennej przechowującej początek naszego komunikatu musi być inna — `pstart`. Łańcuch, który ma zostać wyświetlony, wybieramy za pomocą bardzo prostej instrukcji warunkowej `if`, podobnej do tej, której użyli-

śmy w pierwszej wersji programu. Jeżeli użytkownik podał właściwą cyfrę, wyświetlana jest odpowiednia gwiazda, zaś w przeciwnym przypadku — odpowiedni komunikat.

Program ten ma tylko jedną wadę — kod zakłada, że istnieje sześć opcji, mimo że kompilator przydziela miejsce dla tablicy wskaźników na podstawie liczby podanych łańcuchów inicjalizujących. W związku z tym, dodając łańcuch do listy, musimy wprowadzić modyfikacje do odpowiednich części programu. Dobrze by było, gdybyśmy mogli dodawać łańcuchy, a program automatycznie przystosowywałby się do ich liczby.

Operator sizeof

Tutaj z pomocą może nam przyjść nowy operator — `sizeof`. Operator ten sprawdza liczbę bajtów zajmowanych przez jego operand i zwraca tę wartość jako wartość całkowitą typu `size_t`. Jak pamiętamy z wcześniejszych wyjaśnień, `size_t` jest typem zdefiniowanym w standardowej bibliotece i zazwyczaj odpowiada typowi `unsigned int`.

Spójrzmy na poniższą instrukcję, która odnosi się do zmiennej `dice` z poprzedniego przykładu:

```
cout << sizeof dice;
```

Wartością wyrażenia `sizeof dice` jest `4`, ponieważ zmienna `dice` została zadeklarowana jako typ `int`, a zatem zajmuje cztery bajty pamięci. Dlatego też powyższe wyrażenie zwróci wartość `4`.

Operator `sizeof` może być zastosowany do elementu w tablicy lub do całej tablicy. Jeżeli zostanie zastosowany do tablicy za pomocą jej nazwy, to operator ten zwróci liczbę bajtów zajmowanych przez wszystkie jej elementy. Gdy natomiast zastosujemy go do pojedynczego elementu za pomocą odpowiedniego indeksu lub indeksów, zwrócona zostanie liczba bajtów zajmowanych przez ten element. W związku z tym liczbę elementów w tablicy `pstr` mogliśmy wyświetlić za pomocą następującej instrukcji:

```
cout << (sizeof pstr)/(sizeof pstr[0]);
```

Wyrażenie `(sizeof pstr)/(sizeof pstr[0])` dzieli liczbę bajtów zajmowanych przez całą tablicę przez liczbę bajtów zajmowanych przez pierwszy element tablicy. Jako że każdy element tablicy zajmuje tyle samo pamięci, otrzymany wynik będzie liczbą elementów znajdujących się w tablicy.

Pamiętajmy, że `pstr` jest tablicą wskaźników — użycie operatora `sizeof` dla tej tablicy lub poszczególnych jej elementów nie umożliwi nam uzyskania informacji na temat pamięci zajmowanej przez łańcuchy znaków.

Operator `sizeof` może być zastosowany do nazwy typu, a nie zmiennej, w którym to przypadku zwrócona wartość jest liczbą bajtów zajmowanych przez zmienną tego typu. Nazwa typu powinna się wówczas znajdować pomiędzy nawiasami okrągłymi. Na przykład po wykonaniu poniższej instrukcji

```
size_t size = sizeof(long);
```

zmienna `long_size` będzie miała wartość `4`. Zmienna `size` została zadeklarowana jako typ `size_t`, aby pasowała do typu wartości zwróconej przez operator `sizeof`. Użycie innego typu całkowitego dla zmiennej `size` może spowodować wygenerowanie przez kompilator ostrzeżenia.

spróbuj sam Używanie operatora sizeof

Kod z ostatniego listingu możemy ulepszyć, stosując operator `sizeof` w taki sposób, że będzie się on automatycznie przystosowywał do dowolnej liczby łańcuchów:

```
// Cw4_08.cpp
// Elastyczne zarządzanie tablicą za pomocą operatora sizeof.
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main()
{
   char* pstr[] = { "Robert Redford",        // Inicjalizowanie tablicy wskaźników.
                    "Hopalong Cassidy",
                    "Lassie",
                    "Slim Pickens",
                    "Boris Karloff",
                    "Oliver Hardy"
                  };
   char* pstart = "Twoja szczęśliwa gwiazda to ";
   int count = (sizeof pstr)/(sizeof pstr[0]);  // Liczba elementów tablicy.

   int dice = 0;

   cout << endl
        << " Wybierz swoją szczęśliwą gwiazdę!"
        << " Podaj cyfrę od 1 do " << count << ": ";
   cin >> dice;

   cout << endl;
   if(dice >= 1 && dice <= count)                // Sprawdź poprawność
                                                 // wprowadzanych danych.
      cout << pstart << pstr[dice - 1];          // Wyświetl dane gwiazdy.
   else
      cout << "Przykro mi, ale nie masz swojej szczęśliwej gwiazdy."; // Wprowadzono
                                                                      // nieprawidłowe dane.

   cout << endl;
   return 0;
}
```

Jak to działa

Jak widać, zmiany, które trzeba było wprowadzić, są bardzo proste. Obliczamy tylko liczbę elementów w tablicy wskaźników `pstr` i zapisujemy ją do zmiennej `count`. Następnie wszędzie tam, gdzie znajdują się odniesienia do ogólnej liczby elementów w tablicy wynoszącej 6, używamy zmiennej `count`. Możemy teraz dodać kilka nazwisk do listy szczęśliwych gwiazd, a program sam się dostosuje do nowej liczby łańcuchów.

Stałe wskaźniki oraz wskaźniki do stałych

Zarówno tablica `pstr` w ostatnim przykładzie, jak i łańcuchy, do których utworzone są wskaźniki, oraz zmienna `count` nie zostały przystosowane do zmieniania wartości w programie. Dobrym pomysłem byłoby wyeliminowanie możliwości przypadkowego ich zmodyfikowania w programie. Zmienną `count` możemy w bardzo łatwy sposób zabezpieczyć przed przypadkową modyfikacją, stosując następującą instrukcję:

```
const int count = (sizeof pstr)/(sizeof pstr[0]);
```

Tablica wskaźników natomiast wymaga poświęcenia więcej uwagi. Tablicę tę zadeklarowaliśmy następująco:

```
char* pstr[] = { "Robert Redford",      // Inicjalizowanie tablicy wskaźników.
"Hopalong Cassidy",
"Lassie",
"Slim Pickens",
"Boris Karloff",
"Oliver Hardy"
};
```

Każdy wskaźnik w tej tablicy jest inicjalizowany adresem literału łańcuchowego `"Robert Redford"`, `"Hopalong Cassidy"` itd. Typem literału łańcuchowego jest tablica `const char`, a więc adres tablicy typu `const` przechowujemy we wskaźniku typu niebędącego `const`. Powodem, dla którego kompilator zezwala nam na użycie literału łańcuchowego do inicjalizacji elementu tablicy typu `char*`, jest wsteczna kompatybilność z istniejącym kodem.

Jeżeli do tablicy znaków wprowadzimy następujące zmiany:

```
*pstr[0] = "Stan Laurel";
```

to programu nie będzie można skompilować.

Gdybyśmy chcieli wartość jednego ze wskaźników w tablicy ustawić, aby wskazywał jakiś znak za pomocą następującej instrukcji:

```
*pstr[0] = 'X';
```

to program się skompiluje, ale ulegnie załamaniu w momencie wykonywania tej instrukcji.

Oczywiście nie chcemy, aby nasz program ulegał awariom podczas działania i możemy temu zapobiec. O wiele lepiej jest zapisać tę deklarację następująco:

```
const char* pstr[] = { "Robert Redford",    // Tablica wskaźników
                       "Hopalong Cassidy",  // do stałych.
                       "Lassie",
                       "Slim Pickens",
                       "Boris Karloff",
                       "Oliver Hardy"
};
```

W tym przypadku pojawia się pewna dwuznaczność, jeżeli chodzi o stałość łańcuchów, na które wskazują wskaźniki będące elementami tablicy. Jeżeli spróbujemy je teraz zmienić, to kompilator w trakcie kompilacji zgłosi błąd.

Nadal jednak moglibyśmy napisać poniższą instrukcję:

```
pstr[0] = pstr[1];
```

Wszyscy szczęściarze, którym należałby się Robert Redford, dostaliby w zamian Hopalong Cassidy, ponieważ oba wskaźniki wskazują na to samo nazwisko. Zauważmy, że nie powoduje to zmiany wartości obiektów, na które wskazuje wskaźnik z tablicy — zmieniana jest wartość wskaźnika przechowywanego w `pstr[0]`. Powinniśmy zatem zapobiegać również zmianom tego typu, gdyż niektórzy mogliby odnieść wrażenie, że stary dobry Hoppy jest mniej seksowny niż Redford. Możemy tego dokonać za pomocą poniższej instrukcji:

```
// Tablica stałych wskaźników do stałych.
const char* const pstr[] = { "Robert Redford",
                             "Hopalong Cassidy",
                             "Lassie",
                             "Slim Pickens",
                             "Boris Karloff",
                             "Oliver Hardy"
};
```

Podsumowując, w odniesieniu do typu `const`, wskaźników oraz obiektów można wyróżnić trzy sytuacje. Możemy się spotkać z:

- wskaźnikiem do stałego obiektu,
- stałym wskaźnikiem do obiektu,
- stałym wskaźnikiem do stałego obiektu.

W pierwszej sytuacji nie można zmieniać obiektu, na który wskazuje wskaźnik, ale możemy ustawić wskaźnik, aby wskazywał na coś innego:

```
const char* pstring = "Jakiś tekst.";
```

W drugiej nie można zmienić adresu przechowywanego przez wskaźnik, ale obiekt, na który on wskazuje, można:

```
char* const pstring = Jakiś takst";
```

W trzeciej sytuacji zarówno obiekt, jak i wskaźnik na niego wskazujący zostały zdefiniowane jako stałe, a zatem żadnego z nich nie można zmienić:

```
const char* const pstring = "Jakiś tekst.";
```

Oczywiście zasady te odnoszą się do wskaźników do wszystkich typów. Wskaźnik do typu `char` został tutaj użyty wyłącznie dla potrzeb przykładu.

Wskaźniki i tablice

Nazwy tablic w niektórych warunkach mogą zachowywać się jak wskaźniki. W większości sytuacji użycie samej nazwy tablicy jednowymiarowej powoduje jej automatyczną konwersję na wskaźnik do pierwszego elementu tablicy. Należy pamiętać, że sytuacja taka nie ma miejsca, gdy nazwa tablicy zostanie użyta jako operand operatora `sizeof`.

Mając poniższe deklaracje:

```
double* pdata;
double data[5];
```

możemy napisać następujące przypisanie:

```
pdata = data; // Inicjalizowanie wskaźnika adresem tablicy.
```

Powyższa instrukcja przypisuje adres pierwszego elementu tablicy `data` wskaźnikowi `pdata`. Użycie nazwy tablicy odnosi się do jej adresu. Jeżeli użyjemy nazwy tablicy łącznie z jakimś indeksem, to będzie się ona odnosiła do zawartości elementu odpowiadającego temu indeksowi, a więc jeżeli chcemy adres tego elementu zapisać do wskaźnika, musimy użyć operatora adresowania:

```
pdata = &data[1];
```

W tym przykładzie wskaźnik `pdata` zawiera adres drugiego elementu tablicy.

Arytmetyka wskaźników

Z udziałem wskaźników można przeprowadzać obliczenia arytmetyczne. Zakres działań arytmetycznych na wskaźnikach jest ograniczony do dodawania i odejmowania, ale możemy także wykonywać operacje porównywania wartości wskaźników w celu otrzymania wyniku w postaci wartości logicznej. Arytmetyka z użyciem wskaźników niejawnie zakłada, że wskaźnik wskazuje tablicę oraz że operacja arytmetyczna dotyczy adresu zawartego we wskaźniku. Aby wskaźnikowi `pdata` przypisać adres trzeciego elementu tablicy `data`, możemy posłużyć się następującą instrukcją:

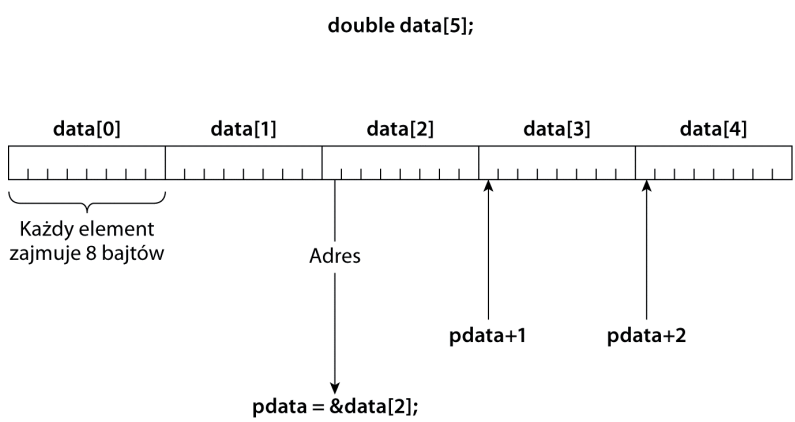
```
pdata = &data[2];
```

W tym przypadku wyrażenie `pdata + 1` odnosiłoby się do adresu elementu `data[3]`, czyli czwartego elementu tablicy `data`. Tak więc aby wskaźnik wskazywał ten element, możemy zastosować następującą instrukcję:

```
pdata += 1;                    // Zwiększ wskaźnik pdata do następnego elementu.
```

Powyższa instrukcja zwiększa adres zawarty we wskaźniku `pdata` o liczbę bajtów zajmowanych przez jeden element tablicy `data`. Ogólnie wyrażenie `pdata + n`, gdzie za `n` możemy podstawić dowolne wyrażenie zwracające liczbę całkowitą, dodaje do adresu zawartego we wskaźniku `pdata` wartość wyrażenia `n*sizeof(double)`, ponieważ został on zadeklarowany jako wskaźnik do typu `double`. Zostało to przedstawione na rysunku 4.8.

Rysunek 4.8



Inaczej mówiąc, zwiększanie lub zmniejszanie wskaźnika działa w kategoriach typu wskazywanego obiektu. Zwiększenie wskaźnika typu `long` o cztery zmienia jego zawartość do następnego adresu typu `long`, a więc zwiększa adres o cztery. Również zwiększenie wskaźnika typu `short` o jeden spowoduje zwiększenie adresu o dwa. Częściej spotykaną notacją zwiększania wskaźnika jest użycie operatora inkrementacji. Na przykład:

```
pdata++; // Zwiększ wskaźnik pdata do wartości następnego elementu.
```

Zapis ten jest równoznaczny (i częściej spotykany) z zapisem `+=`. Mimo tego użyłem formy `+=`, aby było jasne, że — choć wartość zostanie zwiększona o jeden — zwiększenie zazwyczaj nastąpi o liczbę większą niż jeden, z wyjątkiem wskaźnika typu `char`.

Adres powstały w wyniku działań arytmetycznych na wskaźnikach może być wartością z zakresu od wartości adresu pierwszego elementu tablicy do wartości adresu leżącego poza ostatnim jej elementem. Poza tymi granicami zachowanie wskaźnika nie zostało zdefiniowane.

Można oczywiście wyłuskać wskaźnik, na którym wykonaliśmy działania arytmetyczne (inaczej nie byłoby sensu ich wykonywać). Zakładając na przykład, że wskaźnik `pdata` cały czas wskazuje element `data[2]`, poniższa instrukcja:

```
*(pdata + 1) = *(pdata + 2);
```

jest równoznaczna z następującą:

```
data[3] = data[4];
```

Jeżeli chcemy wyłuskać wskaźnik po uprzednim zwiększeniu adresu, który zawiera, to konieczne są nawiasy, ponieważ operator bezpośredniości ma większy priorytet niż operatory arytmetyczne + i -. Jeśli zamiast `*(pdata + 1)` napiszemy `*pdata + 1`, to jeden zostanie dodane do wartości przechowywanej w adresie zawartym w `pdata`, co jest równoznaczne z obliczeniem wartości wyrażenia `data[2] + 1`. Jako że nie jest to `lvalue`, kompilator zgłosi komunikat o błędzie.

Nazwy tablicy można używać w taki sposób, jakby była wskaźnikiem służącym do odwoływania się do elementów tablicy. Jeśli mamy taką samą jednowymiarową tablicę jak poprzednio, zadeklarowaną w następujący sposób:

```
long data[5];
```

to do jej elementów można odwoływać się za pomocą notacji wskaźnikowej, np. do elementu `data[3]` za pomocą `*(data + 3)`. Notację tę można stosować dla wszystkich elementów tablicy, a więc zamiast `data[0]`, `data[1]`, `data[2]` można napisać `*data`, `*(data + 1)`, `*(data+2)` itd.

spróbuj sam Nazwy tablic jako wskaźniki

Zaprezentowany powyżej sposób adresowania tablicowego przećwiczymy na programie obliczającym liczby pierwsze (liczba pierwsza dzieli się bez reszty tylko przez samą siebie i przez jeden).

```
// Cw4_09.cpp
// Obliczanie liczb pierwszych.
#include <iostream>
#include <iomanip>
using std::cout;
using std::endl;
using std::setw;

int main()
{
   const int MAX = 100;            // Liczba wymaganych liczb pierwszych.
   long primes[MAX] = { 2,3,5 };   // Definicja trzech pierwszych liczb pierwszych.
   long trial = 5;                 // Kandydatka na liczbę pierwszą.
   int count = 3;                  // Liczba znalezionych liczb pierwszych.
   int found = 0;                  // Wskaźnik znalezienia liczby pierwszej.

   do
   {
      trial += 2;                        // Następna wartość do sprawdzenia.
      found = 0;                         // Ustawianie wskaźnika znalezienia.
      for(int i = 0; i < count; i++)     // Spróbuj podzielić przez istniejące już liczby pierwsze.
      {
         found = (trial % *(primes + i)) == 0;// True, jeżeli nie ma reszty z dzielenia.
           if(found)                            // Jeżeli nie ma reszty z dzielenia,
              break;                            // to liczba nie jest liczbą pierwszą.
      }
      if (found == 0)                    // Mamy jedną...
         *(primes + count++) = trial;    // ...a więc zapisujemy ją do tablicy liczb pierwszych.
   }while(count < MAX);

   // Wyświetlanie liczb w pięciu kolumnach.
   for(int i = 0; i < MAX; i++)
   {
      if(i % 5 == 0)                  // Znak nowego wiersza po pierwszym i co piątym wierszu.
         cout << endl;
      cout << setw(10) << *(primes + i);
   }
   cout << endl;
   return 0;
}
```

Po skompilowaniu i uruchomieniu tego programu otrzymamy następujący wynik:

```
    2     3     5     7    11
   13    17    19    23    29
   31    37    41    43    47
   53    59    61    67    71
   73    79    83    89    97
  101   103   107   109   113
  127   131   137   139   149
  151   157   163   167   173
  179   181   191   193   197
  199   211   223   227   229
  233   239   241   251   257
  263   269   271   277   281
  283   293   307   311   313
  317   331   337   347   349
  353   359   367   373   379
  383   389   397   401   409
  419   421   431   433   439
  443   449   457   461   463
  467   479   487   491   499
  503   509   521   523   541
```

Jak to działa

Na początku znajdują się te same dyrektywy co zawsze, a więc te dołączające plik nagłówkowy `<iostream>` dla operacji wejścia i wyjścia oraz `<iomanip>`, ponieważ będziemy używać manipulatora strumienia w celu ustawienia szerokości pola dla wyświetlanych liczb. W stałej `MAX` określiliśmy liczbę liczb pierwszych, które mają być obliczone przez nasz program. Trzy pierwsze liczby pierwsze zostały już podane w tablicy `primes`. Cała praca programu wykonywana jest w dwóch pętlach: zewnętrznej pętli `do-while`, która podaje kolejne wartości do sprawdzenia oraz dodaje znalezione liczby pierwsze do tablicy `primes`. Wewnętrzna pętla `for` sprawdza, czy dana liczba jest liczbą pierwszą.

Algorytm zawarty w pętli `for` jest bardzo prosty. Opiera się on na twierdzeniu, że jeżeli liczba nie jest liczbą pierwszą, to musi dać się podzielić przez jedną z pierwszych znalezionych do tej pory mniejszych od niej, ponieważ wszystkie liczby są albo pierwsze albo składają się z liczb pierwszych. W rzeczywistości wystarczyłoby dzielenie przez liczbę mniejszą lub równą pierwiastkowi kwadratowemu sprawdzanej liczby. A więc przykład ten można zmienić tak, aby był jeszcze bardziej efektywny.

```
found = (trial % *(primes + i)) == 0;          // True, jeżeli nie ma reszty.
```

Powyższa instrukcja ustawia zmienną `found` na `1`, jeżeli nie ma reszty z dzielenia wartości zmiennej `trial` przez bieżącą liczbę pierwszą `*(primes + i)` (pamiętamy, że zapis ten jest równoznaczny z `primes[i]`), lub `0` w przeciwnym przypadku. Instrukcja warunkowa `if` powoduje zakończenie działania pętli, jeżeli zmienna `found` ma wartość `1`, ponieważ kandydatka w zmiennej `trial` w tym przypadku nie może być liczbą pierwszą.

Po zakończeniu działania pętli `for` (bez względu na powód) musimy w jakiś sposób zdecydować, czy liczba w zmiennej `trial` była liczbą pierwszą, czy nie. Wskazuje na to zmienna wskaźnikowa `found`.

```
*(primes + count++) = trial;          // ...a więc zapisujemy ją do tablicy liczb pierwszych.
```

Jeżeli zmienna `trial` zawiera liczbę pierwszą, to powyższa instrukcja zapisze ją do tablicy `primes[count]`, a następnie za pomocą przyrostkowego operatora inkrementacji zwiększy zmienną `count`.

Kiedy zostanie już znalezione `MAX` liczb pierwszych, są one wysyłane na wyjście i umieszczane w polach o szerokości dziesięciu znaków, po pięć w każdym wierszu. Za to odpowiedzialna jest poniższa instrukcja:

```
if(i % 5 == 0) // Znak nowego wiersza po pierwszym i co piątym wierszu.
cout << endl;
```

Powyższy kod spowoduje wysłanie znaku nowego wiersza, kiedy `i` ma wartość `0`, `5`, `10` itd.

spróbuj sam Liczenie znaków jeszcze raz

Aby zobaczyć, jak działa notacja wskaźnikowa w operacjach na łańcuchach znaków, zmodyfikujemy wcześniejszy program, który liczył znaki w łańcuchach:

```
// Cw4_10.cpp
// Liczenie znaków w łańcuchu za pomocą wskaźnika.
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main()
{
   const int MAX = 80;                  // Maksymalny rozmiar tablicy.
   char buffer[MAX];                    // Bufor danych wejściowych.
   char* pbuffer = buffer;              // Wskaźnik tablicy buforowej.

   cout << endl                         // Poproś o wprowadzenie danych.
        << "Podaj łańcuch składający się z mniej niż "
        << MAX << " znaków:"
        << endl;

   cin.getline(buffer, MAX, '\n');      // Wczytuj łańcuch aż do napotkania znaku \n.

   while(*pbuffer)                      // Kontynuuj aż do napotkania znaku \0.
      pbuffer++;

   cout << endl
        << "Łańcuch \"" << buffer
        << "\" ma " << pbuffer - buffer << " znaków.";
   cout << endl;
   return 0;
}
```

Poniżej znajduje się przykładowy wynik działania tego programu:

```
Podaj łańcuch składający się z mniej niż 80 znaków:
Nawet najlepsza kobieta ma jeszcze diabelskie żebro w sobie.
Łańcuch "Nawet najlepsza kobieta ma jeszcze diabelskie żebro w sobie." ma 60 znaków.
```

Jak to działa

Powyższy program operuje wskaźnikiem `pbuffer` zamiast nazwą tablicy `buffer`. Nie ma potrzeby stosowania licznika `count` dla zmiennej, gdyż wskaźnik zwiększany jest w pętli `while` aż do napotkania znaku `\0`. Kiedy do tego dojdzie, wskaźnik `pbuffer` będzie zawierał adres tej pozycji w łańcuchu. A zatem liczba znaków we wprowadzonym łańcuchu stanowi różnicę pomiędzy adresem przechowywanym we wskaźniku `pbuffer` a adresem początku tablicy określonym przez `buffer`.

Mogliśmy również zwiększyć wskaźnik w pętli, zapisując ją następująco:

```
while(*pbuffer++); // Kontynuuj aż do napotkania znaku \0.
```

Teraz pętla nie zawiera żadnych instrukcji, tylko warunek sprawdzający. Zapis ten działałby prawie bez zarzutów — wskaźnik byłby zwiększany po napotkaniu znaku `\0`, dzięki czemu adres byłby o jeden większy niż ostatnia pozycja w łańcuchu. W związku z tym licznik liczby znaków musielibyśmy przedstawić jako `pbuffer - buffer -1`.

Zauważmy, że nie możemy tutaj użyć nazwy tablicy w taki sam sposób jak wskaźnika. Wyrażenie `buffer++` jest nieprawidłowe, ponieważ nie można modyfikować adresu wartości, która jest reprezentowana za pomocą nazwy tablicy. Mimo że nazwy tablicy możemy używać tak, jakby była wskaźnikiem, to jednak nim nie jest, ponieważ adres przez nią reprezentowany jest stały.

Używanie wskaźników z tablicami wielowymiarowymi

Używanie wskaźnika do przechowywania adresu tablicy jednowymiarowej jest względnie proste, ale w przypadku tablic wielowymiarowych sprawy się trochę komplikują. Jeżeli nie planujesz się tym zajmować, to możesz pominąć tę część, ponieważ może się ona wydać trochę niejasna. Ale jeżeli masz trochę doświadczenia w języku C, to warto przeczytać także tę część.

Planując korzystanie ze wskaźników z tablicami wielowymiarowymi, musimy pamiętać, aby się nie pogubić. Zilustruję to na przykładzie tablicy `beans` zadeklarowanej w następujący sposób:

```
double beans[3][4];
```

Aby zadeklarować wskaźnik `pbeans` i przypisać mu wartość, możemy posłużyć się następującą instrukcją:

```
double* pbeans;
pbeans = &beans[0][0];
```

W powyższym kodzie ustawiamy wskaźnik na adres pierwszego elementu tablicy, który jest typu `double`. Można również ustawić wskaźnik na adres pierwszego wiersza w tablicy za pomocą następującej instrukcji:

```
pbeans = beans[0];
```

Zapis ten jest równoznaczny z użyciem nazwy tablicy jednowymiarowej, która została zastąpiona adresem. Użyliśmy go już wcześniej. Ponieważ jednak `beans` jest tablicą dwuwymiarową, nie można ustawić wskaźnika na adres za pomocą następującej instrukcji:

```
pbeans = beans; // Ten zapis spowoduje błąd!!
```

Problem leży w typie. Typ zdefiniowanego wskaźnika to `double*`, ale tablica jest typu `double[3][4]`. Wskaźnik przechowujący adres tej tablicy musi być typu `double[4]*`. W języku C++ rozmiary tablicy kojarzone są z jej typem, a więc powyższa instrukcja jest prawidłowa tylko wtedy, kiedy wskaźnik zostanie zadeklarowany z wymaganym wymiarem. Robi się to za pomocą trochę bardziej skomplikowanej niż dotychczas przez nas stosowanej notacji:

```
double (*pbeans)[4];
```

Nawiasy w tym kodzie są niezbędne. Gdybyśmy ich nie zastosowali, zadeklarowalibyśmy tablicę wskaźników. Teraz poprzedni przykład jest prawidłowy, ale wskaźnik ten może być używany wyłącznie do przechowywania adresów z pokazanego wymiaru tablicy.

Notacja wskaźnikowa z tablicami wielowymiarowymi

W celu odniesienia się do elementów tablicy można zastosować notację wskaźnikową. Do każdego elementu zadeklarowanej przez nas wcześniej tablicy `beans` zawierającej trzy wiersze po cztery elementy możemy odnieść się na dwa sposoby:

- używając nazwy tablicy z dwoma indeksami,
- używając nazwy tablicy w notacji wskaźnikowej.

W związku z tym obie poniższe instrukcje są równoznaczne:

```
beans[i][j]
*(*(beans + i) + j)
```

Spójrzmy, jak to działa. W pierwszym wierszu użyto normalnego indeksowania w celu odniesienia się do elementu z przesunięciem `j` w wierszu `i` tablicy.

Znaczenie drugiego wiersza można odgadnąć, przechodząc od wewnątrz do zewnątrz. `beans` odnosi się do adresu pierwszego wiersza tablicy, a więc `beans + i` odnosi się do wiersza `i` tablicy. Wyrażenie `*(beans + i)` stanowi adres pierwszego elementu wiersza `i`, w związku z czym `*(beans + i) + j` jest adresem elementu w wierszu `i` o przesunięciu `j`. Tak więc całe wyrażenie odnosi się do wartości tego elementu.

Jeśli chcemy napisać naprawdę trudny do odczytania kod — choć nie jest to zalecane — poniższe dwie instrukcje, w których pomieszane zostały notacje tablicowa i wskaźnikowa, są również poprawnymi odniesieniami do tego samego elementu tablicy:

```
*(beans[i] + j)
(*(beans + i))[j]
```

Jest jeszcze jeden powód używania wskaźników, tak naprawdę najważniejszy ze wszystkich — możliwość dynamicznego przydzielania pamięci zmiennym. Tym się teraz będziemy zajmować.

Dynamiczne przydzielanie pamięci

Praca z ustaloną liczbą zmiennych w programie może być bardzo ograniczona. Podczas wykonywania programu często zachodzi potrzeba podjęcia decyzji, ile pamięci przydzielić do przechowywania zmiennych różnego typu, w zależności od wprowadzanych danych. W przypadku jednych danych najlepiej byłoby użyć dużej tablicy całkowitej, natomiast w innym przypadku wymagana może być tablica liczb zmiennopozycyjnych. Jako że zmienne, dla których pamięć przydzielana jest dynamicznie, nie mogą być zdefiniowane w czasie kompilacji, nie można im także nadać żadnych nazw w źródle programu. Podczas ich tworzenia identyfikowane są one za pomocą adresu w pamięci, który przechowywany jest we wskaźniku. Dzięki potędze wskaźników oraz narzędziom dynamicznego zarządzania pamięcią w Visual C++ 2005 pisanie programów o takim stopniu elastyczności jest szybkie i łatwe.

Pamięć wolna, czyli sterta

W większości przypadków podczas wykonywania programów w komputerze znajdują się nieużywane zasoby pamięci. Pamięć ta w C++ zwana jest **stertą** lub czasami **pamięcią wolną**. W pamięci tej można zarezerwować obszar dla nowej zmiennej danego typu za pomocą specjalnego operatora, który zwraca adres przydzielonego obszaru pamięci. Operator ten to `new`, a jego odpowiednikiem o odwrotnym działaniu jest operator `delete`, który usuwa przydzieloną przez `new` pamięć.

Pamięć wolną można przydzielić zmiennym w jednej części programu, a następnie ją zwolnić i zwrócić na stertę po zakończeniu pracy z tymi zmiennymi. W ten sposób zwalniamy pamięć do ponownego użycia przez inne zmienne z dynamicznie przydzielaną pamięcią w dalszej części programu.

Pamięci wolnej używamy zawsze, gdy potrzebujemy przydzielić pamięć elementom, które mogą być określone wyłącznie podczas działania programu. Przykładem takiej sytuacji może być przydzielenie pamięci zmiennej do przechowywania łańcucha wprowadzonego przez użytkownika programu. Nie ma sposobu sprawdzenia rozmiaru tego łańcucha z wyprzedzeniem, a więc pamięć przydzielimy mu w czasie działania programu za pomocą operatora `new`. Później zobaczymy przykład dynamicznego przydzielania pamięci tablicom, w którym wymiary tablicy określane będą przez użytkownika w czasie działania programu.

Technika ta daje bardzo duże możliwości. Pozwala na bardzo efektywne używanie pamięci i w wielu przypadkach programy napisane przy jej użyciu potrafią poradzić sobie z o wiele większymi problemami związanymi z wykorzystaniem znacznie większej ilości danych, niż byłoby to możliwe w innym przypadku.

Operatory new i delete

Przypuśćmy, że potrzebujemy miejsca w pamięci dla zmiennej typu `double`. W takim przypadku możemy zdefiniować wskaźnik typu `double`, a następnie zażądać przydzielenia mu pamięci podczas działania programu. Aby tego dokonać, posłużymy się operatorem `new`, jak poniżej:

```
double* pvalue = NULL; // Wskaźnik zainicjalizowany wartością zerową.
pvalue = new double;   // Żądanie pamięci dla zmiennej typu double.
```

Jest to dobry moment, aby przypomnieć, że *wszystkie wskaźniki muszą zostać zainicjalizowane*. Dynamiczne używanie pamięci zazwyczaj pociąga za sobą pewną liczbę oczekujących na użycie wskaźników i ważne jest, żeby nie zawierały one żadnych przypadkowych wartości. Powinniśmy zadbać o to, by każdy wskaźnik nieprzechowujący żadnej prawidłowej wartości zawierał wartość zerową.

Operator `new` w drugim wierszu powyższego kodu powinien zwrócić adres pamięci na stercie przydzielonej zmiennej typu `double`. Adres ten będzie przechowywany we wskaźniku `pvalue`. Następnie możemy odnieść się do tej zmiennej przy użyciu tego wskaźnika za pomocą operatora pośredniości, jak już widzieliśmy. Na przykład:

```
*pvalue = 9999.0;
```

Oczywiście przydzielenie pamięci może nie nastąpić ze względu na fakt, że wolna pamięć została wyczerpana lub pofragmentowana w wyniku wcześniejszego używania, co oznacza, że nie ma wystarczającej liczby występujących obok siebie bajtów, aby pomieścić zmienną, dla której chcemy uzyskać miejsce. Nie musimy się jednak zbyt o to martwić. W standardzie ANSI C++ operator `new` *spowoduje wyjątek*, jeżeli pamięć nie będzie mogła zostać przydzielona z jakiegokolwiek powodu, co z kolei spowoduje zakończenie programu. Wyjątki w C++ są mechanizmem sygnalizowania błędów. Więcej na ich temat będziemy mówili w rozdziale 6.

Zmienną utworzoną za pomocą operatora `new` możemy również zainicjalizować. Biorąc jako przykład zmienną typu `double`, której pamięć została przydzielona za pomocą operatora `new` i której adres przechowywany jest we wskaźniku `pvalue`, jej wartość moglibyśmy podczas tworzenia ustawić na `999.0` za pomocą następującej instrukcji:

```
pvalue = new double(999.0);          // Przydziel pamięć zmiennej double i zainicjalizuj ją.
```

Kiedy nie potrzebujemy już zmiennej, której przydzieliliśmy dynamicznie pamięć, możemy zwolnić zajmowaną przez nią pamięć za pomocą operatora `delete`:

```
delete pvalue;                       // Zwolnij pamięć wskazywaną przez wskaźnik pvalue.
```

Dzięki temu nieużywana pamięć może być użyta przez inną zmienną. Jeżeli nie użyjemy operatora `delete` i do wskaźnika `pvalue` zapiszemy inny adres, to nie będzie możliwości zwolnienia tej pamięci ani też użycia przechowywanej w niej zmiennej, gdyż dostęp do jej adresu zostanie utracony. Sytuacja taka nazywa się wyciekiem pamięci, zwłaszcza jeżeli powtarza się kilkakrotnie.

Dynamiczne przydzielanie pamięci tablicom

Dynamiczne przydzielanie pamięci tablicy jest bardzo proste. Jeżeli chcemy przydzielić pamięć tablicy typu `char`, zakładając, że `pstr` jest wskaźnikiem do `char`, możemy posłużyć się następującą instrukcją:

```
pstr = new char[20];                 // Przydziel pamięć łańcuchowi składającemu się z 20 znaków.
```

Aby usunąć tablicę utworzoną przed chwilą w obszarze pamięci wolnej, musimy użyć operatora `delete`. Instrukcja do tego służąca wygląda następująco:

```
delete [] pstr;                    // Usunięcie tablicy wskazywanej przez wskaźnik pstr.
```

Warto zauważyć, że w powyższym kodzie zastosowaliśmy nawiasy kwadratowe, aby zaznaczyć, że usuwamy tablicę. Usuwając tablicę z wolnej pamięci, należy zawsze wpisywać kwadratowe nawiasy, w innym przypadku wynik operacji będzie bowiem nieprzewidywalny. Należy również zauważyć, że nie określamy tu żadnych wymiarów, po prostu wpisujemy `[]`.

Oczywiście wskaźnik `pstr` zawiera teraz adres obszaru pamięci, który mógł zostać już przydzielony do jakiegoś innego celu, a więc z pewnością nie powinniśmy go używać. Usuwając obiekt z pamięci za pomocą operatora `delete` w celu jej zwolnienia, wartość wskaźnika zawsze powinno się ponownie ustawiać na zero:

```
pstr = 0;                          // Ustaw wskaźnik na zero.
```

spróbuj sam Używanie wolnej pamięci

Sposób działania dynamicznego przydzielania pamięci prześledzimy na zmodyfikowanej wersji programu obliczającego określoną liczbę liczb pierwszych. Tym razem do ich przechowywania użyjemy obszarów wolnej pamięci.

```
// Cw4_11.cpp
// Obliczanie liczb pierwszych przy użyciu dynamicznego przydzielania pamięci.
#include <iostream>
#include <iomanip>
using std::cin;
using std::cout;
using std::endl;
using std::setw;

int main()
{
   long* pprime = 0;               // Wskaźnik tablicy prime.
   long trial = 5;                 // Kandydatka na liczbę pierwszą.
   int count = 3;                  // Licznik znalezionych liczb pierwszych.
   int found = 0;                  // Wskaźnik znalezienia liczby pierwszej.
   int max = 0;                    // Żądana liczba liczb pierwszych.

   cout << endl
        << "Podaj, ile chcesz obliczyć liczb pierwszych (co najmniej 4): ";
   cin >> max;                     // Żądana liczba liczb pierwszych.

   if(max < 4)                     // Sprawdź podaną liczbę, jeżeli jest to mniej niż 4,
      max = 4;                     // to zwiększ ją do 4.

   pprime = new long[max];

   *pprime = 2;                    // Wstaw trzy
   *(pprime + 1) = 3;              // początkowe liczby pierwsze.
   *(pprime + 2) = 5;

   do
```

```
    {
       trial += 2;                                  // Następna wartość do sprawdzenia.
       found = 0;                                   // Ustaw wskaźnik found.
       for(int i = 0; i < count; i++)               // Dzielenie przez istniejące liczby pierwsze.
       {
          found =(trial % *(pprime + i)) == 0;// True, jeżeli dzielenie nie ma reszty.
          if(found)                                 // Jeżeli nie ma reszty z dzielenia,
             break;                                 // to liczba nie jest liczbą pierwszą.
       }
       if (found == 0)                              // Mamy jedną...
          *(pprime + count++) = trial;              // ...a więc zapisujemy ją do tablicy primes.
    } while(count < max);

    // Wyślij pięć liczb pierwszych do wiersza.
    for(int i = 0; i < max; i++)
    {
       if(i % 5 == 0)                               // Nowy wiersz dla pierwszego i co piątego wiersza.
          cout << endl;
       cout << setw(10) << *(pprime + i);
    }
    delete [] pprime;                               // Zwolnij pamięć
    pprime = 0;                                     // i ponownie ustaw wartość wskaźnika.
    cout << endl;
    return 0;
}
```

Poniżej znajduje się przykładowy wynik działania tego programu.

```
Podaj, ile chcesz obliczyć liczb pierwszych (co najmniej 4): 20

 2     3     5     7    11
13    17    19    23    29
31    37    41    43    47
53    59    61    67    71
```

Jak to działa

W rzeczywistości program ten jest podobny do swojej pierwotnej wersji. Po otrzymaniu żądanej liczby liczb pierwszych w zmiennej całkowitej `max` za pomocą operatora `new` przydzielamy pamięć z wolnego obszaru tablicy o odpowiednich rozmiarach. Dodaliśmy także zabezpieczenie na ewentualność podania przez użytkownika liczby mniejszej niż cztery. Jest to potrzebne, ponieważ program wymaga przydzielenia pamięci z wolnego obszaru co najmniej trzem inicjalizującym liczbom pierwszym plus jednej nowej. Rozmiar tablicy określamy, stawiając zmienną `max` pomiędzy kwadratowymi nawiasami znajdującymi się za specyfikacją typu tablicy:

```
pprime = new long[max];
```

Adres obszaru pamięci przydzielonego za pomocą operatora `new` przechowujemy we wskaźniku `pprime`. Gdyby pamięć nie mogła zostać przydzielona, program zostałby w tym momencie zamknięty.

Po pomyślnym przydzieleniu pamięci przechowującej liczby pierwsze trzem pierwszym elementom tablicy zostają nadane wartości będące trzema początkowymi liczbami pierwszymi:

```
*pprime = 2;          // Wstaw trzy
*(pprime + 1) = 3;  // początkowe liczby pierwsze.
*(pprime + 2) = 5;
```

W celu uzyskania dostępu do trzech pierwszych elementów tablicy posłużyliśmy się operatorem wyłuskania. Jak już widzieliśmy wcześniej, nawiasy w drugiej i trzeciej instrukcji zostały zastosowane ze względu na fakt, że priorytet operatora * jest wyższy niż operatora +.

Nie można podać wartości początkowych elementów tablicy, dla której przydzielamy pamięć dynamicznie. Jeżeli chcemy ustawić wartości początkowe elementów tablicy, musimy użyć jawnych instrukcji przypisania.

Obliczanie liczb pierwszych odbywa się identycznie jak poprzednio. Jedyna zmiana polega na tym, że nazwa wskaźnika `pprime` zastąpiła nazwę tablicy `primes`, której używaliśmy w poprzedniej wersji. Proces wysyłania danych na wyjście jest taki sam. Dynamiczne pozyskiwanie pamięci nie sprawia żadnych problemów. Po jej przydzieleniu nie ma ona żadnego wpływu na sposób zapisu obliczeń.

Po zakończeniu używania tablicy usuwamy ją z obszaru wolnej pamięci za pomocą operatora `delete`, pamiętając o dodaniu nawiasów kwadratowych w celu zaznaczenia, że usuwamy tablicę.

```
delete [] pprime;              // Zwolnij pamięć
```

Mimo że w tym przypadku nie jest to konieczne, ustawiamy wskaźnik na zero:

```
pprime = 0;                    // i ponownie ustaw wartość wskaźnika.
```

Cała pamięć przydzielona w programie z wolnego obszaru jest zwalniana w momencie jego zakończenia, ale dobrze jest wyrobić sobie nawyk ponownego ustawiania wskaźników na zero, kiedy nie wskazują one już prawidłowych obszarów pamięci.

Dynamiczne przydzielanie pamięci tablicom wielowymiarowym

Przydzielanie wolnej pamięci tablicy wielowymiarowej wymaga użycia operatora `new` w trochę bardziej skomplikowanej formie niż w przypadku tablic jednowymiarowych. Zakładając, że wskaźnik `pbeans` został już prawidłowo zadeklarowany, aby uzyskać pamięć dla tablicy `beans[3][4]`, której używaliśmy już wcześniej, moglibyśmy napisać następującą instrukcję:

```
pbeans = new double [3][4];          // Przydziel pamięć tablicy 3×4.
```

Wystarczy tylko podać wymiary tablicy w nawiasach kwadratowych po nazwie typu elementów tablicy.

Przydzielanie pamięci tablicy trójwymiarowej wymaga podania dodatkowego wymiaru za operatorem `new`, jak widać na poniższym przykładzie:

```
pBigArray = new double [5][10][10]; // Przydziel pamięć tablicy 5×10×10.
```

Bez względu na liczbę wymiarów w utworzonej tablicy, aby ją zniszczyć i zwolnić zajmowaną przez nią pamięć, piszemy następującą instrukcję:

```
delete [] pBigArray;                    // Zwolnij pamięć zajmowaną przez tablicę.
```

Bez względu na liczbę wymiarów w tablicy niszczymy ją zawsze za pomocą operatora `delete`, po którym umieszczamy jedną parę nawiasów kwadratowych.

Wiemy już, że zmiennej możemy użyć jako określenia wymiaru tablicy jednowymiarowej, dla której pamięć ma zostać przydzielona dynamicznie za pomocą operatora `new`. Taka sama możliwość istnieje w przypadku tablic dwuwymiarowych, ale z tym ograniczeniem, że za pomocą zmiennej może być określony tylko ostatni wymiar po lewej. Wszystkie pozostałe wymiary muszą być stałymi lub wyrażeniami stałymi. W związku z tym możemy napisać:

```
pBigArray = new double[max][10][10];
```

`max` w powyższym kodzie jest zmienną. Podanie zmiennej dla innego wymiaru niż ostatni po lewej spowoduje wygenerowanie przez kompilator komunikatu o błędzie.

Używanie referencji

Referencje pod wieloma względami przypominają wskaźniki (dlatego też mówię o nich dopiero teraz), ale nie są tym samym. Prawdziwe znaczenie referencji staje się oczywiste, kiedy zaczynamy używać ich z funkcjami, a w szczególności w kontekście programowania zorientowanego obiektowo. Na pierwszy rzut oka wydają się bardzo prostą, a nawet trywialną koncepcją, ale to tylko mylące pozory. Jak zobaczymy później, referencje dają nam pewne niezwykłe możliwości, a w niektórych sytuacjach pozwalają na osiągnięcie rezultatów niemożliwych do uzyskania inną drogą.

Czym jest referencja

Referencja jest aliasem zmiennej. Ma ona nazwę, której można użyć zamiast nazwy zmiennej. Jako że jest to alias zmiennej, a nie wskaźnik, to zmienna ta musi być zadeklarowana przed deklaracją referencji. Dodatkowo — w przeciwieństwie do wskaźników — referencji nie można zmieniać, aby reprezentowały inne zmienne.

Deklarowanie i inicjalizowanie referencji

Przypuśćmy, że mamy poniższą deklarację zmiennej:

```
long number = 0;
```

Referencję do tej zmiennej możemy zadeklarować za pomocą następującej instrukcji:

```
long& rnumber = number;          // Deklaracja referencji do zmiennej number.
```

Znak & znajdujący się po nazwie typu long i przed nazwą zmiennej rnumber informuje, że deklarowana jest właśnie referencja i nazwa zmiennej (number), którą reprezentuje, znajdująca się po znaku równości, została określona jako wartość początkowa. A zatem zmienna rnumber jest referencją do long. Możemy teraz użyć naszej referencji zamiast oryginalnej nazwy zmiennej. Na przykład instrukcja:

```
rnumber += 10;
```

spowoduje zwiększenie zmiennej number o 10.

Spójrzmy, jaka jest różnica pomiędzy referencją rnumber a wskaźnikiem pnumber, zadeklarowanymi w poniższej instrukcji:

```
long* pnumber = &number;          // Inicjalizacja wskaźnika adresem.
```

Powyższa instrukcja deklaruje wskaźnik pnumber i inicjalizuje go adresem zmiennej number. Dzięki temu możemy zwiększyć wartość zmiennej number za pomocą poniższej instrukcji:

```
*pnumber += 10;                   // Zwiększ zmienną number poprzez wskaźnik.
```

Pomiędzy wskaźnikiem a referencją jest znaczna różnica. Wskaźnik musi zostać wyłuskany i bez względu na to, jaki adres zawiera, jest używany do uzyskiwania dostępu do zmiennej, która ma być użyta w wyrażeniu. W przypadku referencji nie ma potrzeby wyłuskiwania. Czasami referencja przypomina wskaźnik, który został już wyłuskany, jednak nie można jej zmienić, aby wskazywała inną zmienną. Referencja jest wiernym odpowiednikiem zmiennej, do której się odnosi. Może się wydawać, że referencja jest po prostu alternatywnym sposobem zapisu danej zmiennej i w tym przypadku rzeczywiście tak jest. Jednak przy omawianiu funkcji w C++ przekonamy się, że nie jest to prawda i że dostarcza ona pewnych bardzo pożytecznych możliwości.

Programowanie w C++/CLI

W CLR dynamiczne przydzielanie pamięci działa inaczej. CLR posiada własną stertę pamięci, która jest niezależna od sterty w natywnym C++. CLR automatycznie usuwa pamięć przydzieloną na stercie, kiedy nie jest już potrzebna, a więc pisząc programy CLR, nie potrzebujemy operatora delete. CLR może również co pewien czas kompaktować stertę w celu uniknięcia jej fragmentacji. Zarządzanie stertą i czyszczenie sterty dostarczanej przez CLR nazywa się **usuwaniem nieużytków** (nieużytki to usunięte zmienne i obiekty), a sterta poddana takiemu procesowi to **sterta poddana procesowi usuwania nieużytków** (ang. *garbage-collected heap*). W programach w C++/CLI do przydzielania pamięci zamiast operatora new używamy operatora gcnew. Przedrostek gc oznacza, że przydzielamy pamięć ze sterty oczyszczonej (ang. *garbage-collected heap*), a nie z natywnej sterty C++, gdzie usuwanie nieużytków należy do naszych obowiązków.

Mechanizm usuwania nieużytków CLR potrafi usuwać obiekty i zwalniać zajmowaną przez nie pamięć, kiedy nie są już potrzebne. W tym momencie rodzi się pytanie: „Skąd ten mechanizm wie, kiedy dany obiekt na stercie nie jest już potrzebny?”. Odpowiedź jest bardzo prosta: CLR śledzi każdą zmienną, która wskazuje jeden z obiektów na stercie. Gdy nie ma żadnych

zmiennych zawierających adres danego obiektu, nie można się do niego odwołać, a więc można go usunąć.

Ponieważ proces usuwania nieużytków może być wykonywany razem z kompaktowaniem sterty w celu usunięcia pofragmentowanych, nieużywanych bloków pamięci, adresy elementów danych przechowywanych na stercie mogą ulec zmianie. W związku z tym z taką stertą nie możemy używać zwykłych wskaźników z natywnego C++, ponieważ — jeżeli lokalizacja danych się zmieni — stałyby się one bezużyteczne. W tym przypadku potrzebujemy sposobu uzyskiwania dostępu do obiektów na stercie, który pozwala na uaktualnianie adresu, kiedy mechanizm usuwania nieużytków zmieni lokalizacje danych na stercie. Możemy to osiągnąć na dwa sposoby: za pomocą **uchwytu śledzącego** (zwanego także po prostu **uchwytem** — ang. *tracking handle*), który jest analogiczny do wskaźnika w natywnym C++, oraz za pomocą referencji śledzącej (ang. *tracking reference*), która w CLR jest odpowiednikiem referencji z natywnego C++.

Uchwyty śledzące

Uchwyt śledzący jest pod pewnymi względami podobny do zwykłego uchwytu z natywnego C++, ale są też znaczne różnice. Uchwyt śledzący przechowuje adres, który jest automatycznie aktualizowany przez mechanizm usuwania nieużytków, jeżeli obiekt przez niego wskazywany zostanie przeniesiony podczas kompaktowania sterty. Nie można jednak przy użyciu wskaźnika śledzącego wykonywać operacji arytmetycznych na adresach, tak jak robiliśmy to przy użyciu natywnych wskaźników. Rzutowanie wskaźników śledzących jest również niedozwolone.

Do wszystkich obiektów utworzonych na stercie CLR muszą być utworzone uchwyty śledzące. Wszystkie obiekty należące do klasy, będące odniesieniami do typów klasowych, przechowywane są na stercie, a zatem zmienne tworzone w celu odnoszenia się do tych obiektów muszą być uchwytami śledzącymi. Na przykład typ klasowy `String` jest typem klasy referencji, a więc zmienne odnoszące się do obiektów typu `String` muszą być uchwytami śledzącymi. Pamięć dla typów klas wartości przydzielana jest domyślnie na stosie, ale można wybrać stertę za pomocą operatora `gcnew`. Jest to także dobry moment, aby przypomnieć sobie to, co powiedziałem w drugim rozdziale, że zmienne, którym została przydzielona pamięć na stercie (czyli wszystkie typy referencyjne CLR), nie mogą być deklarowane w zasięgu globalnym.

Deklarowanie uchwytów śledzących

Uchwyt do typu definiujemy, stawiając po jego nazwie znak ^ (potocznie zwany daszkiem). Na przykład poniżej znajduje się deklaracja uchwytu śledzącego o nazwie `proverb`, który może przechowywać adres obiektu typu `String`:

```
String^ proverb;
```

Powyższy kod definiuje zmienną `proverb` jako uchwyt śledzący typu `String^`. Podczas tworzenia uchwytu jest on automatycznie inicjalizowany wartością zerową, a więc nie odnosi się do niczego. Aby jawnie przypisać uchwytowi wartość zerową, należy posłużyć się słowem kluczowym `nullptr`:

```
proverb = nullptr;              // Ustaw wartość uchwytu na zero.
```

Należy zauważyć, że w tym przypadku — w przeciwieństwie do natywnych wskaźników — nie można użyć wartości `0`, która reprezentowałaby wartość `null`. Jeżeli użyjemy tutaj wartości `0`, to zostanie ona przekonwertowana na typ obiektu, do którego odnosi się ten uchwyt, a adres tego nowego obiektu będzie przechowywany w tym uchwycie.

Oczywiście uchwyt można zainicjalizować jawnie podczas deklaracji. Poniżej znajduje się jeszcze jedna przykładowa instrukcja definiująca uchwyt do obiektu `String`:

```
String^ saying = L"Kiedyś myślałem, że jestem niezdecydowany, ale teraz to już
sam nie wiem.";
```

Powyższa instrukcja tworzy na stercie obiekt typu `String` zawierający łańcuch po prawej stronie przypisania. Adres nowego obiektu przechowywany jest w zmiennej `saying`. Warto zauważyć, że typem literału łańcuchowego jest `const wchar_t*`, a nie typ `String`. Sposób, w jaki zdefiniowana jest klasa `String`, umożliwia używanie takich literałów do tworzenia obiektów typu `String`.

Poniżej znajduje się przykład utworzenia uchwytu do typu wartości:

```
int^ value = 99;
```

Powyższa instrukcja tworzy uchwyt `value` typu `int^`, a wartość, którą wskazuje na stosie, została zainicjalizowana wartością `99`. Pamiętajmy, że utworzyliśmy pewien rodzaj wskaźnika, a więc `value` nie może uczestniczyć w działaniach arytmetycznych bez uprzedniego wyłuskania. Do tego celu z kolei używamy tego samego operatora * co w przypadku wskaźników natywnych. Na przykład poniżej znajduje się instrukcja, w której została użyta wartość wskazywana przez uchwyt śledzący w działaniu arytmetycznym:

```
int result = 2*(*value)+15;
```

Wyrażenie `*value` znajdujące się w nawiasach uzyskuje dostęp do liczby całkowitej przechowywanej w adresie umieszczonym w uchwycie śledzącym, dzięki czemu zmienna `result` zostaje ustawiona na wartość 213.

Zauważmy, że jeśli po prawej stronie instrukcji użyjemy uchwytu, to nie ma potrzeby wyłuskiwać go jawnie — kompilator sam się tym zajmie. Na przykład:

```
int^ result = 0;
result = 2*(*value)+15;
```

W powyższym przykładzie najpierw tworzymy uchwyt `result`, który wskazuje na stercie wartość `0`. Warto zauważyć, że w tym momencie kompilator zgłosi ostrzeżenie, ponieważ wygląda to tak, jakbyśmy chcieli zainicjalizować uchwyt wartością zerową, a tak nie należy tego robić. Jako że w następnej instrukcji `result` znajduje się po lewej stronie przypisania, a prawa strona daje wynik, kompilator jest w stanie „domyślić się”, że przed zapisaniem wartości trzeba najpierw wyłuskać `result`. Oczywiście moglibyśmy zapisać to także w jawny sposób:

```
*result = 2*(*value)+15;
```

Zauważmy, że zapis taki działa tylko wtedy, gdy `result` jest już zdefiniowany. Gdyby został on tylko zadeklarowany, to po uruchomieniu programu otrzymalibyśmy błąd wykonywania. Na przykład:

```
int^ result;                    // Deklaracja, ale nie definicja.
*result = 2*(*value)+15;        // Komunikat o błędzie — nieobsłużony wyjątek.
```

Jako że w drugiej instrukcji wyłuskujemy uchwyt `result`, sugerujemy, że obiekt przez niego wskazywany już istnieje. Ale tak nie jest i dlatego spowodowaliśmy błąd wykonania. Pierwsza instrukcja jest deklaracją uchwytu `result`, którego wartość domyślnie zostaje ustawiona na `null`, a wartości tej nie można wyłuskać. Jeżeli nie wyłuskamy jawnie uchwytu `result` w drugiej instrukcji, to wszystko będzie w porządku, gdyż wynik wyrażenia znajdującego się po prawej stronie przypisania jest typem klasy wartości, a jego adres przechowywany jest w uchwycie `result`.

Tablice CLR

Tablice CLR są inne niż tablice w natywnym C++. Pamięć dla tablicy CLR przydzielana jest na oczyszczonej stercie, ale to nie jedyna różnica. Tablice CLR mają wbudowane pewne funkcje (za chwilę będziemy o nich mówić), których nie mają tablice w natywnym C++. Typ zmiennej tablicowej określamy za pomocą słowa kluczowego `array`. W trójkątnych nawiasach, po słowie kluczowym `array`, musimy także podać typ dla elementów tablicy. W związku z tym ogólna forma zmiennej odnoszącej się do tablicy jednowymiarowej to `array<element_type>^`. Jako że tablica CLR tworzona jest na stercie, zmienna tablicowa zawsze jest uchwytem śledzącym. Poniżej znajduje się przykładowa deklaracja zmiennej tablicowej:

```
array<int>^ data;
```

Zmienna tablicowa `data` może przechowywać wskaźnik do dowolnej jednowymiarowej tablicy elementów typu `int`.

Tablicę CLR można utworzyć równocześnie z deklaracją zmiennej tablicowej za pomocą operatora `gcnew`:

```
array<int>^ data = gcnew array<int>(100);     // Utwórz tablicę przechowującą 100 liczb
                                              // całkowitych.
```

Powyższa instrukcja tworzy jednowymiarową tablicę o nazwie `data` (zauważ, że zmienna tablicowa jest uchwytem śledzącym, a więc nie można zapomnieć o znaku ^ po typie elementu pomiędzy nawiasami trójkątnymi). Liczba elementów znajduje się w nawiasach okrągłych po określeniu typu tablicy, a więc tablica ta zawiera 100 elementów, z których każdy może przechowywać wartości typu `int`.

Podobnie jak w przypadku tablic w natywnym C++, elementy w tablicach CLR indeksowane są od zera, a więc wartości elementów tablicy `data` możemy ustawić następująco:

```
for(int i = 0 ; i<100 ; i++)
data[i] = 2*(i+1);
```

Powyższa pętla ustawia wartości elementów na 2, 4, 6 itd. aż do 200. Elementy w tablicach CLR są obiektami, a więc w tej tablicy przechowujemy obiekty typu Int32. Oczywiście w działaniach arytmetycznych zachowują się one jak zwykłe liczby całkowite, tak więc fakt, że są one obiektami, jest w takich przypadkach bez znaczenia.

W powyższej pętli liczba elementów została podana w postaci literału. Lepiej jednak byłoby użyć właściwości Length tablicy, która przechowuje liczbę elementów:

```
for(int i = 0 ; i < data->Length ; i++)
  data[i] = 2*(i+1);
```

Aby uzyskać dostęp do właściwości Length, używamy operatora ->, ponieważ data jest uchwytem i zachowuje się jak wskaźnik. Właściwość Length zapisuje liczbę wartości w postaci 32-bitowej liczby całkowitej. W razie potrzeby rozmiar tablicy możemy rozszerzyć do wartości 64-bitowej za pomocą właściwości LongLength.

Można również przejść przez wszystkie elementy tablicy za pomocą pętli for each:

```
array<int>^ values = { 3, 5, 6, 8, 6};
for each(int item in values)
{
   item = 2*item + 1;
   Console::Write("{0,5}",item);
}
```

Zmienna item znajdująca się wewnątrz pętli odnosi się do wszystkich elementów w tablicy values. Pierwsza instrukcja w ciele pętli zamienia bieżącą wartość elementu na jej podwójną wartość plus jeden. Druga instrukcja wysyła na wyjście nową wartość wyrównaną do prawej w polu o szerokości pięciu znaków, a więc wynik tego fragmentu kodu przedstawia się następująco:

```
7     11     13     17     13
```

Zmienna tablicowa może przechowywać adres dowolnej tablicy o takiej samej liczbie wymiarów i takim samym typie danych. Na przykład:

```
data = gcnew array<int>(45);
```

Powyższa instrukcja tworzy nową jednowymiarową tablicę 45 elementów typu int i zapisuje jej adres do uchwytu data. Oryginalna tablica zostaje usunięta.

Można również utworzyć tablicę, podając zbiór wartości początkowych elementów:

```
array<double>^ samples = { 3.4, 2.3, 6.8, 1.2, 5.5, 4.9, 7.4, 1.6};
```

Rozmiar tablicy określony jest przez liczbę wartości początkowych znajdujących się w nawiasach (w tym przypadku osiem). Wartości te zostają przypisane do elementów w kolejności, w jakiej zostały podane.

Oczywiście elementy w tablicy mogą być dowolnego typu, a więc z łatwością możemy utworzyć tablicę łańcuchów:

```
array<String^>^ names = { "Jack", "Jane", "Joe", "Jessica", "Jim", "Joanna"};
```

Elementy tej tablicy zostały zainicjalizowane za pomocą łańcuchów podanych w nawiasach. Liczba tych łańcuchów określa liczbę elementów tablicy. Obiekty typu `String` tworzone są na stercie CLR, a więc typ elementu jest typem uchwytu śledzącego — `String^`.

Gdy nie zainicjalizujemy zadeklarowanej zmiennej tablicowej, musimy tę tablicę utworzyć jawnie, jeśli chcemy używać listy wartości początkowych. Na przykład:

```
array<String^>^ names;                          // Deklaracja zmiennej tablicowej.
names = gcnew array<String^>{ "Jack", "Jane", "Joe", "Jessica", "Jim", "Joanna"};
```

Druga z powyższych instrukcji tworzy tablicę i inicjalizuje ją łańcuchami w nawiasach. Bez jawnej definicji z użyciem operatora `gcnew`, instrukcji tej nie można by skompilować.

Za pomocą funkcji statycznej `Clear()` zdefiniowanej w klasie `Array` można ustawić na zero dowolną sekwencję elementów zawierających wartości liczbowe. Funkcje statyczne wywołuje się przy użyciu nazwy klasy. Więcej na temat tych funkcji dowiemy się przy okazji szczegółowego omawiania klas. Poniżej znajduje się przykład użycia funkcji `Clear()`:

```
Array::Clear(samples, 0, samples->Length); // Ustaw wszystkie elementy na zero.
```

Pierwszym argumentem funkcji `Clear()` jest tablica, którą chcemy wyczyścić, drugi argument to indeks pierwszego elementu, który chcemy wyczyścić, a trzeci to liczba elementów do wyczyszczenia. A zatem powyższy przykład ustawia wszystkie elementy tablicy `samples` na `0.0`. Jeżeli funkcję `Clear()` zastosujemy z tablicą uchwytów śledzących, jak np. `String^`, to elementy zostaną ustawione na `null`. Zastosowanie tej funkcji dla tablicy wartości logicznych spowoduje ustawienie wartości elementów na `false`.

Czas na wyczyszczenie jakiejś tablicy.

spróbuj sam Używanie tablic CLR

Przykład ten generuje tablicę zawierającą losowe wartości, a następnie odnajduje największą z nich:

```
// Cw4_12.cpp: main project file.
// Używanie tablic CLR.
#include "stdafx.h"

using namespace System;

int main(array<System::String ^> ^args)
{
  array<double>^ samples = gcnew array<double>(50);

  // Generowanie losowych wartości dla elementów.
  Random^ generator = gcnew Random;
  for(int i = 0 ; i< samples->Length ; i++)
     samples[i] = 100.0*generator->NextDouble();

  // Wysyłanie na wyjście próbek.
  Console::WriteLine(L"Tablica zawiera następujące liczby:");
  for(int i = 0 ; i< samples->Length ; i++)
```

```
    {
      Console::Write(L"{0,10:F2}", samples[i]);
      if((i+1)%5 == 0)
         Console::WriteLine();
    }

    // Szukanie największej wartości.
    double max = 0;
    for each(double sample in samples)
       if(max < sample)
          max = sample;

    Console::WriteLine(L"Największa wartość w tablicy to {0:F2}", max);
    return 0;
}
```

Przykładowy wynik działania tego programu widać poniżej:

```
Tablica zawiera następujące liczby:
     63,03     66,07     83,73     12,11     88,30
     80,28     76,99     89,57     83,78     66,96
     69,56     25,02     86,09     56,39     48,04
      0,84     64,79     90,73     11,58     46,37
     11,26     55,80     75,29     75,01     50,44
     88,99     26,72     57,32     95,52     63,91
     77,49     43,02     28,21      6,97     47,51
     24,58     50,23     40,12     72,85     54,45
     79,60     15,13     80,63     86,40     56,83
     66,04     41,69     59,03      5,86      9,94
Największa wartość w tablicy to 95,52
```

Jak to działa

Najpierw tworzymy tablicę 50 elementów typu `double`:

```
array<double>^ samples = gcnew array<double>(50);
```

Zmienna tablicowa `samples` musi być uchwytem śledzącym, gdyż tablice CLR tworzone są na stercie oczyszczonej.

Tablicę wypełniamy losowymi wartościami typu `double` za pomocą następujących instrukcji:

```
Random^ generator = gcnew Random;
for(int i = 0 ; i< samples->Length ; i++)
   samples[i] = 100.0*generator->NextDouble();
```

Pierwsza instrukcja tworzy na stercie CLR obiekt typu `Random`. Obiekt `Random` zawiera funkcje, które generują liczby pseudolosowe. W tym przypadku użyliśmy funkcji `NextDouble()` w pętli, która zwraca losową liczbę typu `double` należącą do zbioru 0,0 – 1,0. Dzięki pomnożeniu tego przez 100 otrzymujemy wartości od 0,0 do 100,0. Pętla `for` zapisuje do każdego elementu tablicy `samples` losową liczbę.

Obiekt `Random` zawiera również funkcję `Next()`, która zwraca losową liczbę nieujemną typu `int`. Jeżeli w wywołaniu funkcji `Next()` podamy argument w postaci liczby całkowitej, to zwróci ona losową, nieujemną wartość mniejszą niż podany argument. Można także

podać dwa argumenty w postaci liczb całkowitych, które będą reprezentowały wartości minimalną i maksymalną zwróconych liczb losowych.

Następna pętla wysyła na wyjście zawartość tablicy, po pięć elementów na wiersz:

```
Console::WriteLine(L"Tablica zawiera następujące liczby:");
for(int i = 0 ; i< samples->Length ; i++)
{
   Console::Write(L"{0,10:F2}", samples[i]);
   if((i+1)%5 == 0)
      Console::WriteLine();
```

Wewnątrz pętli określamy, że wartość każdego elementu ma znajdować się w polu o szerokości 10 i mieć dwa miejsca po przecinku. Dzięki określeniu szerokości pól wartości zostaną wyrównane w kolumnach. Za każdym razem, gdy wynikiem działania `(i+1)%5` jest zero, wysyłamy znak nowego wiersza, co ma miejsce co pięć wartości, dzięki czemu w każdym wierszu mamy pięć wartości.

Na zakończenie sprawdzamy największą wartość:

```
double max = 0;
for each(double sample in samples)
   if(max < sample)
      max = sample;
```

W powyższym przykładzie użyłem pętli `for each`, aby pokazać, że można jej tutaj użyć. Porównuje ona wartość `max` z wartością każdego elementu i za każdym razem, gdy znajdzie wartość większą od niej, wartość `max` jest ustawiana na tę właśnie wartość. W ten sposób otrzymujemy największą wartość.

Gdybyśmy chcieli jeszcze zapisać pozycję indeksu elementu zawierającego największą wartość, to moglibyśmy posłużyć się pętlą `for`. Na przykład:

```
double max = 0;
int index = 0;
for (int i = 0 ; i < sample->Length ; i++)
   if(max < samples[i])
   {
      max = samples[i];
      index = i;
   }
```

Sortowanie tablic jednowymiarowych

Klasa `Array` w przestrzeni nazw `System` definiuje funkcję `Sort()`, która sortuje elementy tablicy jednowymiarowej w porządku rosnącym. Aby posortować zawartość tablicy, wystarczy jako argument funkcji `Sort()` podać uchwyt do niej. Poniżej znajduje się przykład:

```
array<int>^ samples = { 27, 3, 54, 11, 18, 2, 16};
Array::Sort(samples);                          // Sortuj elementy tablicy.
for each(int value in samples)                 // Wyświetl elementy tablicy.
   Console::Write(L"{0, 8}", value);
Console::WriteLine();
```

Wywołanie funkcji `Sort()` spowodowało ustawienie elementów tablicy `samples` w rosnącej kolejności. Wynik wykonania powyższego fragmentu kodu jest następujący:

```
2     3     11     16     18     27     54
```

Podając dwa dodatkowe argumenty do funkcji `Sort()`, można ustawić w kolejności pewien zbiór wartości. Te argumenty to indeks pierwszego elementu ze zbioru do posortowania, a drugi to liczba kolejnych elementów. Na przykład:

```
array<int>^ samples = { 27, 3, 54, 11, 18, 2, 16};
Array::Sort(samples, 2, 3);                    // Sortuj elementy ze zbioru 2 – 4.
```

Powyższa instrukcja sortuje trzy elementy tablicy `samples`, które zaczynają się od pozycji indeksowej 2. Po wykonaniu tych instrukcji elementy w tablicy będą miały następujące wartości:

```
27     3     11     18     54     2     16
```

Funkcja `Sort()` występuje w jeszcze kilku innych wersjach, o których możemy przeczytać w dokumentacji. Jedną z nich, szczególnie przydatną, wprowadzę teraz. Ta wersja funkcji zakłada, że mamy dwie skojarzone tablice, tak że elementy w pierwszej z nich są kluczami odpowiadających im elementów w drugiej. Moglibyśmy na przykład przechowywać w jednej tablicy imiona osób, a w drugiej ich wagę. Funkcja `Sort()` sortuje tablicę `names` (zawierającą imiona) w kolejności rosnącej, a elementy tablicy `weights` (zawierającej wagę) sortuje w taki sposób, aby nadal odpowiadały one kolejności pierwszej tablicy. Spójrzmy na przykład.

spróbuj sam Sortowanie dwóch skojarzonych ze sobą tablic

Poniższy kod tworzy tablicę imion oraz przechowuje wagę każdej osoby w elementach drugiej tablicy, odpowiadających danym osobom. Następnie obie tablice zostają posortowane za pomocą jednej operacji:

```
// Cw4_13.cpp: main project file.
// Sortowanie tablicy kluczy (nazwisk) i tablicy obiektów (wagi).

#include "stdafx.h"

using namespace System;

int main(array<System::String ^> ^args)
{
  array<String^>^ names = { "Jill", "Ted", "Mary", "Eve", "Bill", "Al"};
  array<int>^ weights = { 103, 168, 128, 115, 180, 176};

  Array::Sort( names,weights);                    // Sortuj tablice.
  for each(String^ name in names)                 // Wyświetl imiona.
    Console::Write(L"{0, 10}", name);
  Console::WriteLine();

  for each(int weight in weights)                 // Wyświetl wagi.
    Console::Write(L"{0, 10}", weight);
  Console::WriteLine();
  return 0;
}
```

Wynik działania powyższego programu przedstawia się następująco:

```
 Al      Bill       Eve      Jill      Mary       Ted
176       180       115       103       128       168
```

Jak to działa

Wartości w tablicy `weights` odpowiadają wadze osoby znajdującej się w elemencie o tym samym indeksie w tablicy `names`. Funkcja `Sort()`, którą tutaj wywołujemy, sortuje obie tablice za pomocą użytej jako argument pierwszej tablicy (w tym przypadku `names`) w celu określenia kolejności obu tablic. W wynikach działania programu widać, że każdej osobie nadal przypisana jest odpowiednia waga w drugiej tablicy.

Przeszukiwanie tablicy jednowymiarowej

W klasie `Array` dostępne są również funkcje służące do wyszukiwania elementów w tablicach jednowymiarowych. Funkcja `BinarySearch()` przeszukuje całą tablicę lub określoną jej część w celu odnalezienia indeksu danego elementu za pomocą algorytmu binarnego. Algorytm binarny wymaga, aby elementy, które mają zostać przeszukane, były posortowane, a więc przed użyciem tej funkcji najpierw trzeba tablicę posortować.

Poniższy kod przeszukuje całą tablicę:

```
array<int>^ values = { 23, 45, 68, 94, 123, 127, 150, 203, 299};
int toBeFound = 127;
int position = Array::BinarySearch(values, toBeFound);
if(position<0)
   Console::WriteLine(L"Liczba {0} nie została odnaleziona.", toBeFound);
else
   Console::WriteLine(L"Liczba {0} została odnaleziona w indeksie {1}.",
   toBeFound, position);
```

Wartość, którą chcemy znaleźć, przechowywana jest w zmiennej `toBeFound`. Pierwszym argumentem do funkcji `BinarySearch()` jest uchwyt do tablicy, którą chcemy przeszukać, a drugi argument określa, czego szukamy. Rezultat poszukiwania zwracany przez funkcję `BinarySearch()` jest wartością typu `int`. Jeżeli w tablicy zostanie znaleziony drugi z podanych argumentów, to zwrócony zostanie jego indeks. W przeciwnym przypadku zwrócona zostanie ujemna liczba całkowita. A zatem zwróconą wartość trzeba sprawdzić w celu określenia, czy cel został odnaleziony. Jako że wartości w tablicy `values` są już posortowane rosnąco, nie ma potrzeby sortować tablicy przed jej przeszukaniem. Powyższy fragment kodu da następujący wynik:

```
Liczba 127 została odnaleziona w indeksie 5.
```

Aby przeszukać tylko określony zbiór elementów tablicy, należy użyć wersji funkcji `BinarySearch()`, która przyjmuje cztery argumenty. Pierwszy argument to uchwyt tablicy do przeszukania, drugi to indeks, od którego ma się rozpocząć przeszukiwanie, trzeci określa liczbę elementów, które chcemy przeszukać, a ostatni poszukiwaną wartość. Poniżej znajduje się przykład takiego przeszukiwania:

```
array<int>^ values = { 23, 45, 68, 94, 123, 127, 150, 203, 299};
int toBeFound = 127;
int position = Array::BinarySearch(values, 3, 6, toBeFound);
```

Powyższy kod przeszukuje wartości tablicy od czwartego elementu do ostatniego. Podobnie jak poprzednia wersja, funkcja zwraca znaleziony indeks lub wartość ujemną, jeżeli nic nie znajdzie.

Omówmy przeszukiwanie na przykładzie.

spróbuj sam Przeszukiwanie tablic

Poniżej znajduje się zmodyfikowana wersja kodu z poprzedniego listingu z dodanym przeszukiwaniem:

```
// Cw4_14.cpp: main project file.
// Przeszukiwanie tablicy.

#include "stdafx.h"

using namespace System;

int main(array<System::String ^> ^args)
{
  array<String^>^ names = { "Jill", "Ted", "Mary", "Eve", "Bill",
                            "Al",   "Ned", "Zoe",  "Dan", "Jean"};
  array<int>^ weights = { 60, 112, 70, 80, 120,
                          110, 125, 58,  119, 74 };
  array<String^>^ toBeFound = {"Bill", "Eve", "Al", "Fred"};

  Array::Sort( names, weights);                    // Sortuj tablice.

  int result = 0;                                  // Zapisz wynik szukania.
  for each(String^ name in toBeFound)              // Szukaj wag.
  {
    result = Array::BinarySearch(names, name);     // Przeszukaj tablicę names.

    if(result<0)                                   // Sprawdź wynik.
      Console::WriteLine(L"Waga osoby o imieniu {0} nie została odnaleziona.", name);
    else
      Console::WriteLine(L"{0} waży {1} kg.", name, weights[result]);
  }
  return 0;
}
```

Poniżej widać rezultat powyższego programu:

```
Bill waży 120 kg.
Eve waży 80 kg.
Al waży 110 kg.
Waga osoby o imieniu Fred nie została odnaleziona.
```

Jak to działa

Utworzyliśmy dwie skojarzone ze sobą tablice — tablicę imion oraz tablicę odpowiadających tym osobom wag w kilogramach. Utworzyliśmy także tablicę `toBeFound` do przechowywania imion osób, których wagę chcemy poznać.

Tablice `names` i `weights` sortujemy według tablicy `names`. Następnie w pętli `for each` przeszukujemy tablicę `names` w celu odnalezienia każdego z imion znajdujących się w tablicy `toBeFound`. Zmiennej pętlowej `name` zostaje przypisane po kolei każde imię z tablicy `toBeFound`. Bieżącego imienia wewnątrz pętli poszukujemy za pomocą następującej instrukcji:

```
result = Array::BinarySearch(names, name);          // Przeszukaj tablicę names.
```

Instrukcja ta zwraca indeks elementu z tablicy `names`, który zawiera imię `name` lub ujemną liczbę całkowitą, jeżeli imię nie zostanie odnalezione. Następnie za pomocą instrukcji warunkowej `if` sprawdzamy wynik i wysyłamy odpowiedni komunikat:

```
if(result<0)                                        // Sprawdź wynik.
   Console::WriteLine(L"Osoba o imieniu {0} nie została odnaleziona.", name);
else
   Console::WriteLine(L"{0} waży {1} kg.", name, weights[result]);
```

Jako że kolejność w tablicy `weights` została zmieniona w celu dopasowania do tablicy `names`, zawartością zmiennej `result` możemy zindeksować tablicę `weights`. Indeks w tym przypadku ma wartość taką jak element z tablicy `names`, gdzie zostało znalezione imię.

Jak widać, w tym, co wygenerował program, imię `Fred` nie zostało znalezione.

Kiedy poszukiwanie binarne zakończy się niepowodzeniem, to zwrócona wartość nie jest przypadkową wartością ujemną. Jest ona bitowym odpowiednikiem indeksu pierwszego elementu, który jest większy od poszukiwanego obiektu, lub jest bitowym odpowiednikiem właściwości `Length` tablicy, jeżeli żaden element nie jest większy niż poszukiwany obiekt. Dysponując tą wiedzą, można użyć funkcji `BinarySearch()` do sprawdzenia, gdzie w tablicy powinniśmy wstawić nowy obiekt, nie zaburzając przy tym kolejności elementów. Przypuśćmy, że do tablicy `names` chcemy dodać imię `Fred`. Indeks miejsca, w którym powinniśmy to imię umieścić, możemy znaleźć za pomocą poniższych instrukcji:

```
array<String^>^ names = { "Jill", "Ted", "Mary", "Eve", "Bill",
                          "Al", "Ned", "Zoe", "Dan", "Jean"};
Array::Sort(names);       // Sortuj tablicę.
String^ name = L"Fred";
int position = Array::BinarySearch(names, name);
if(position<0)            // Jeżeli wartość ujemna,
  position = ~position;   // odwróć bity, aby uzyskać indeks miejsca do wstawienia.
```

Jeżeli wynik wyszukiwania jest negatywny, odwrócenie wszystkich bitów daje nam indeks miejsca, w którym powinno zostać wstawione nowe imię. Jeżeli wynik jest pozytywny, imię jest identyczne z imieniem w tym miejscu, a więc możemy tego wyniku użyć bezpośrednio jako nowej pozycji.

Tablicę `names` możemy teraz skopiować do nowej tablicy zawierającej jeden element więcej i użyć wartości pozycji w celu wstawienia imienia w odpowiednim miejscu:

```
array<String^>^ newNames = gcnew array<String^>(names->Length+1);
// Skopiuj elementy z tablicy names do newNames
for(int i = 0 ; i<position ; i++)
   newNames[i] = names[i];

newNames[position] = name;          // Skopiuj nowy element.

if(position<names->Length)          // Jeżeli w tablicy names pozostały jakieś elementy,
   for(int i = position ; i<names->Length ; i++)
      newNames[i+1] = names[i];     // skopiuj je do tablicy newNames.
```

Powyższy kod tworzy tablicę o jeden element większą niż stara tablica. Następnie kopiujemy wszystkie elementy ze starej tablicy do nowej do indeksu `position - 1`. Następnie kopiujemy nowe imię, po którym umieszczone zostają pozostałe elementy starej tablicy. Aby usunąć starą tablicę, piszemy:

```
names = nullptr;
```

Tablice wielowymiarowe

Możemy tworzyć tablice wielowymiarowe. Maksymalnie tablica może być 32-wymiarowa, co powinno w zupełności wystarczyć do większości zastosowań. Liczbę wymiarów tablicy podajemy w trójkątnych nawiasach bezpośrednio po typie elementu, oddzielając ją od niego przecinkiem. Domyślnie tablica ma jeden wymiar (dlatego do tej pory nie musieliśmy podawać liczby wymiarów). Poniżej znajduje się przykład utworzenia dwuwymiarowej tablicy elementów typu całkowitego:

```
array<int, 2>^ values = gcnew array<int, 2>(4, 5);
```

Powyższa instrukcja tworzy dwuwymiarową tablicę z czterema wierszami i pięcioma kolumnami, a więc w sumie złożoną z 20 elementów. Aby uzyskać dostęp do tablicy wielowymiarowej, należy podać odpowiednią liczbę indeksów — po jednym dla każdego wymiaru. Indeksy podaje się w nawiasach kwadratowych, oddzielanych przecinkami i umieszczanych po nazwie tablicy. Poniższy przykładowy kod ustawia wartości elementów dwuwymiarowej tablicy liczb całkowitych:

```
int nrows = 4;
int ncols = 5;
array<int, 2>^ values = gcnew array<int, 2>(nrows, ncols);
for(int i = 0 ; i<nrows ; i++)
   for(int j = 0 ; j<ncols ; j++)
      values[i,j] = (i+1)*(j+1);
```

Zagnieżdżone pętle przechodzą przez wszystkie elementy tablicy. Pętla zewnętrzna iteruje przez wiersze, a wewnętrzna przez wszystkie elementy w bieżącym wierszu. Jak widzimy, każdy element ustawiany jest na wartość wyrażenia `(i+1)*(j+1)`, dzięki czemu elementy w pierwszym wierszu będą miały wartości 1, 2, 3, 4, 5, w drugim — 2, 4, 6, 8, 10, a w trzecim — 4, 6, 12, 16, 20.

Jak nietrudno zauważyć, użyta tutaj notacja dotycząca uzyskiwania dostępu do elementu tablicy dwuwymiarowej jest inna niż notacja użyta w natywnym C++. Nie jest to przypadek. Tablica w C++/CLI nie jest tablicą tablic, tak jak tablica w natywnym C++, ale jest prawdziwą ta-

blicą dwuwymiarową. Do tablic dwuwymiarowych w C++/CLI nie można stosować pojedynczych indeksów, gdyż nie mają one tutaj znaczenia — tablica jest prawdziwą tablicą dwuwymiarową, a nie tablicą tablic. Jak już wcześniej mówiłem, liczba wymiarów tablicy określana jest mianem jej poziomu, a więc poziom tablicy `values` w poprzednim fragmencie kodu to 2. W C++/CLI można oczywiście definiować tablice na poziomach 3. i wyższych, aż do 32. W przeciwieństwie do C++/CLI, tablice w natywnym C++ są zawsze na poziomie 1., ponieważ są one dwiema lub większą liczbą tablic tablic. Jak przekonamy się później w C++/CLI również można definiować tablice tablic.

Użyjmy tablicy wielowymiarowej w przykładzie.

spróbuj sam Używanie tablic wielowymiarowych

Poniższy program CLR tworzy tabliczkę mnożenia 12×12 w dwuwymiarowej tablicy:

```
// Cw4_15.cpp: main project file.
// Używanie tablic dwuwymiarowych.

#include "stdafx.h"

using namespace System;

int main(array<System::String ^> ^args)
{
  const int SIZE = 12;
  array<int, 2>^ products = gcnew array<int, 2>(SIZE,SIZE);

  for (int i = 0 ; i < SIZE ; i++)
    for(int j = 0 ; j < SIZE ; j++)
      products[i,j] = (i+1)*(j+1);

  Console::WriteLine(L"Oto tabliczka mnożenia do {0}:",  SIZE);

  // Rysuj poziomą linię oddzielającą.
  for(int i = 0 ; i <= SIZE ; i++)
    Console::Write(L"____");
  Console::WriteLine();                  // Wyślij znak nowego wiersza.

  // Rysuj górną krawędź tabeli.
  Console::Write(L"    |");
  for(int i = 1 ; i <= SIZE ; i++)
    Console::Write(L"{0,3} |", i);
  Console::WriteLine();                  // Wyślij znak nowego wiersza.

  // Rysuj poziomą linię oddzielającą z kreskami pionowymi.
  for(int i = 0 ; i <= SIZE ; i++)
    Console::Write(L"____|");
  Console::WriteLine();                  // Wyślij znak nowego wiersza.

  // Wyślij pozostałe wiersze.
  for(int i = 0 ; i<SIZE ; i++)
  {
    Console::Write(L"{0,3} |", i+1);
    for(int j = 0 ; j<SIZE ; j++)
```

```
        Console::Write(L"{0,3} |", products[i,j]);

      Console::WriteLine();                    // Wyślij znak nowego wiersza.
    }

    // Rysuj poziomą linię oddzielającą.
    for(int i = 0 ; i <= SIZE ; i++)
      Console::Write(L"_____");
    Console::WriteLine();                      // Wyślij znak nowego wiersza.

    return 0;
}
```

Wynik działania powyższego programu powinien być następujący:

```
Oto tabliczka mnożenia do 12:
_____________________________________________________________________
    |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 |
____|____|____|____|____|____|____|____|____|____|____|____|____|
  1 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 |
  2 |  2 |  4 |  6 |  8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 |
  3 |  3 |  6 |  9 | 12 | 15 | 18 | 21 | 24 | 27 | 30 | 33 | 36 |
  4 |  4 |  8 | 12 | 16 | 20 | 24 | 28 | 32 | 36 | 40 | 44 | 48 |
  5 |  5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 | 55 | 60 |
  6 |  6 | 12 | 18 | 24 | 30 | 36 | 42 | 48 | 54 | 60 | 66 | 72 |
  7 |  7 | 14 | 21 | 28 | 35 | 42 | 49 | 56 | 63 | 70 | 77 | 84 |
  8 |  8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 | 80 | 88 | 96 |
  9 |  9 | 18 | 27 | 36 | 45 | 54 | 63 | 72 | 81 | 90 | 99 |108 |
 10 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 |100 |110 |120 |
 11 | 11 | 22 | 33 | 44 | 55 | 66 | 77 | 88 | 99 |110 |121 |132 |
 12 | 12 | 24 | 36 | 48 | 60 | 72 | 84 | 96 |108 |120 |132 |144 |
_____________________________________________________________________
```

Jak to działa

Kodu wydaje się dość dużo, ale większa jego część dotyczy sterowania wyglądem wysyłanych danych. Za pomocą poniższej instrukcji tworzymy tablicę dwuwymiarową:

```
const int SIZE = 12;
array<int, 2>^ products = gcnew array<int, 2>(SIZE,SIZE);
```

Pierwszy wiersz definiuje stałą całkowitą przechowującą liczbę elementów każdego wymiaru tablicy. W drugim wierszu zdefiniowaliśmy dwuwymiarową tablicę, która zawiera po dwanaście elementów w dwóch wierszach. Zapisuje ona wyniki do tabeli o wymiarach 12×12.

Wartości elementów tej tablicy obliczamy za pomocą zagnieżdżonej pętli:

```
for (int i = 0 ; i < SIZE ; i++)
   for(int j = 0 ; j < SIZE ; j++)
      products[i,j] = (i+1)*(j+1);
```

Pętla zewnętrzna iteruje przez wiersze, a wewnętrzna przez kolumny. Wartością każdego elementu jest iloczyn wartości indeksowych wiersza i kolumny po uprzednim zwiększeniu ich o jeden. Reszta kodu w ciele funkcji `main()` dotyczy już wyłącznie prezentacji wysyłanych danych.

Po wysłaniu nagłówka tabeli tworzymy rząd pionowych kresek w celu oznaczenia górnej części tabeli:

```
for(int i = 0 ; i <= SIZE ; i++)
   Console::Write(L"_____");
Console::WriteLine();              // Wyślij znak nowego wiersza.
```

Każda iteracja pętli dodaje pięć znaków poziomej kreski. Jako że górny limit w pętli także się wlicza, rysujemy 13 zestawów pięciu kresek w celu utworzenia miejsca dla etykiet wierszy tabeli i dwunastu kolumn.

Następnie, za pomocą jeszcze jednej pętli, piszemy wiersz etykiet kolumn tabeli:

```
// Rysuj górną krawędź tabeli.
Console::Write(L"    |");
for(int i = 1 ; i <= SIZE ; i++)
  Console::Write(L"{0,3} |", i);
Console::WriteLine();                  // Wyślij znak nowego wiersza.
```

Miejsce nad etykietą wiersza musimy utworzyć oddzielnie, gdyż jest ono wyjątkowe i nie zawiera żadnej wartości. Wszystkie etykiety kolumn pisane są za pomocą pętli. Następnie wysyłamy znak nowego wiersza, przygotowując się na kolejne wiersze danych.

Dane w wierszach tworzone są w zagnieżdżonej pętli:

```
for(int i = 0 ; i<SIZE ; i++)
{
   Console::Write(L"{0,3} |", i+1);
   for(int j = 0 ; j<SIZE ; j++)
      Console::Write(L"{0,3} |", products[i,j]);
   Console::WriteLine();               // Wyślij znak nowego wiersza.
}
```

Pętla zewnętrzna iteruje przez wiersze, a kod w niej zawarty tworzy kompletny wiersz, włącznie z etykietą wiersza po lewej stronie. Pętla wewnętrzna wstawia wartości z tablicy `products`, które odpowiadają wierszom o numerze `i`. Wartości oddzielane są pionowymi kreskami.

Pozostały kod wysyła na wyjście dodatkowe linie poziome, wykańczając dół tabeli.

Tablice tablic

Elementy tablic mogą być dowolnego typu, a więc można utworzyć taką tablicę, w której elementami będą uchwyty śledzące do tablic. W ten sposób można utworzyć struktury zwane **tablicami postrzępionymi**, ponieważ każdy uchwyt do tablicy może zawierać inną liczbę elementów. Najłatwiej to zrozumieć na przykładzie. Przypuśćmy, że chcemy przyporządkować dzieci w klasie do odpowiednich grup w zależności od otrzymanej oceny w skali od 1 do 5. Najpierw utworzymy tablicę pięciu elementów, z których każdy przechowuje tablicę imion. Poniżej znajduje się potrzebna do tego instrukcja:

```
array< array< String^ >^ >^ grades = gcnew array< array< String^ >^ >(5);
```

Na pierwszy rzut oka kod z taką dużą liczbą daszków może wydawać się bardzo skomplikowany, ale to tylko pozory. Zmienna tablicowa `grades` jest uchwytem typu `array<type>^`.

Każdy element tablicy również jest uchwytem do tablicy, a więc typ elementów tablicy jest taki sam (`array<type>^`), w związku z czym musi zostać podany w nawiasach trójkątnych w specyfikacji typu oryginalnej tablicy i z tego powodu mamy zapis `array< array<type>^ >^`. Elementy przechowywane w tablicach są także obiektami typu `String`, a więc w ostatnim wyrażeniu musimy zamienić typ na `String^`. Dzięki temu otrzymujemy tablicę typu `array< array< String^ >^ >^`.

Mając już tablicę tablic, możemy przejść do tworzenia tablic imion. Poniżej przedstawiam przykładowy kod do wykonania tego zadania:

```
grades[0] = gcnew array<String^>{"Louise", "Jack"};                    // Ocena 5.
grades[1] = gcnew array<String^>{"Bill", "Mary", "Ben", "Joan"};       // Ocena 4.
grades[2] = gcnew array<String^>{"Jill", "Will", "Phil"};              // Ocena 3.
grades[3] = gcnew array<String^>{"Ned", "Fred", "Ted", "Jed", "Ed"};   // Ocena 2.
grades[4] = gcnew array<String^>{"Dan", "Ann"};                        // Ocena 1.
```

Wyrażenie `grades[n]` uzyskuje dostęp do n-tego elementu tablicy `grades` i jest to oczywiście za każdym razem uchwyt do tablicy uchwytów typu `String^`. A zatem każdy z powyższych pięciu wierszy tworzy tablicę uchwytów do obiektów typu `String` i zapisuje adres do jednego z elementów tablicy `grades`. Jak widać, tablice łańcuchów mają różne rozmiary, a więc w ten sposób możemy zarządzać zbiorem tablic o dowolnych rozmiarach.

Całą tablicę tablic można utworzyć i zainicjalizować za pomocą pojedynczej instrukcji:

```
array< array< String^ >^ >^ grades = gcnew array< array< String^ >^ >
   {
      gcnew array<String^>{"Louise", "Jack"},                          // Ocena 5.
      gcnew array<String^>{"Bill", "Mary", "Ben", "Joan"},             // Ocena 4.
      gcnew array<String^>{"Jill", "Will", "Phil"},                    // Ocena 3.
      gcnew array<String^>{"Ned", "Fred", "Ted", "Jed", "Ed"},         // Ocena 2.
      gcnew array<String^>{"Dan", "Ann"}                               // Ocena 1.
   };
```

Wartości początkowe elementów podane są w nawiasach klamrowych.

Spójrzmy teraz na przykładowy program, w którym zaprezentuję sposób przetwarzania tablic tablic.

spróbuj sam Używanie tablic tablic

Utwórz program konsolowy CLR i umieść w nim następujący kod źródłowy:

```
// Cw4_16.cpp: main project file.
// Używanie tablic tablic.

#include "stdafx.h"

using namespace System;

int main(array<System::String ^> ^args)
{
array< array< String^ >^ >^ grades = gcnew array< array< String^ >^ >
          {
```

```
            gcnew array<String^>{"Louise", "Jack"},                    // Ocena 5.
            gcnew array<String^>{"Bill", "Mary", "Ben", "Joan"},       // Ocena 4.
            gcnew array<String^>{"Jill", "Will", "Phil"},              // Ocena 3.
            gcnew array<String^>{"Ned", "Fred", "Ted", "Jed", "Ed"}, // Ocena 2.
            gcnew array<String^>{"Dan", "Ann"}                         // Ocena 1.
          };

  wchar_t gradeLetter = '5';

  for each(array< String^ >^ grade in grades)
  {
    Console::WriteLine("Uczniowie z oceną {0}:", gradeLetter--);

    for each( String^ student in grade)
      Console::Write("{0,12}",student);          // Wyślij bieżące imię.

    Console::WriteLine();                        // Wyślij nowy wiersz.
  }
  return 0;
}
```

Poniżej znajduje się wynik działania powyższego programu:

```
Uczniowie z oceną 5:
      Louise     Jack
Uczniowie z oceną 4:
      Bill       Mary        Ben        Joan
Uczniowie z oceną 3:
      Jill       Will        Phil
Uczniowie z oceną 2:
      Ned        Fred        Ted        Jed          Ed
Uczniowie z oceną 1:
      Dan        Ann
```

Jak to działa

Definicja tablicy jest identyczna, jak widzieliśmy wcześniej. Następnie definiujemy zmienną typu `wchar_t` o nazwie `gradeLetter` o wartości początkowej `1`. Posłuży nam ona do prezentacji ocen na ekranie.

Imiona uczniów oraz ich oceny wyświetlone zostały za pomocą pętli zagnieżdżonych. Zewnętrzna pętla `for each` iteruje przez elementy tablicy `grades`:

```
for each(array< String^ >^ grade in grades)
{
   // Przetwarzaj uczniów z bieżącą oceną...
}
```

Zmienna pętlowa `grade` jest typu `array< String^ >^`, ponieważ taki jest typ elementów w tablicy `grades`. Zmienna `grade` wskazuje po kolei każdą tablicę uchwytów typu `String^`, a więc w pierwszej iteracji wskazuje tablicę uczniów z oceną 5, w drugiej tablicę uczniów z oceną 4 i tak dalej aż do tablicy przechowującej imiona uczniów z oceną 1.

Przy każdej iteracji pętli zewnętrznej wykonywany jest następujący kod:

```
Console::WriteLine("Uczniowie z oceną {0}:", gradeLetter--);
```

```
for each( String^ student in grade)
   Console::Write("{0,12}",student);          // Wyślij bieżące imię.

Console::WriteLine();                         // Wyślij nowy wiersz.
```

Pierwsza instrukcja wysyła wiersz zawierający bieżącą wartość zmiennej `gradeLetter`, która początkowo ma wartość `5`. Wyrażenie to zmniejsza także wartość zmiennej `gradeLetter`, dzięki czemu w kolejnych powtórzeniach przyjmuje ona po kolei wartości `4`, `3`, `2` i `1`.

Następnie wewnętrzna pętla `for each` iteruje po kolei przez wszystkie imiona bieżącej tablicy ocen. Instrukcja wyjściowa użyta w tym przypadku to `Console::Write()`, a więc wszystkie imiona pojawią się w tym samym wierszu. Imiona wyrównane są do prawej w polach o szerokości 12 znaków. Po pętli funkcja `WriteLine()` wysyła nowy wiersz w celu przeniesienia danych dotyczących następnej oceny do następnego wiersza.

Jako pętli wewnętrznej mogliśmy również użyć pętli `for`:

```
for (int i = 0 ; i < grade->Length ; i++)
Console::Write("{0,12}",grade [i]);           // Wyślij bieżące imię.
```

Pętla ograniczona jest przez właściwość `Length` bieżącej tablicy imion wskazywanej przez zmienną `grade`.

Jako pętli zewnętrznej również mogliśmy użyć pętli `for`. W tym przypadku potrzebne by były dalsze zmiany w wewnętrznej pętli i wyglądałaby ona następująco:

```
for (int j = 0 ; j < grades->Length ; j++)
{
   Console::WriteLine("Uczniowe z oceną {0}:", gradeLetter+j);
   for (int i = 0 ; i < grades[j]->Length ; i++)
      Console::Write("{0,12}",grades [j][i]); // Wyślij bieżące imię.
   Console::WriteLine();
}
```

Teraz `grades[j]` wskazuje `j` tablicę elementów, a więc wyrażenie `grades[j][i]` wskazuje `i` imię w `j` tablicy imion.

Łańcuchy

Jak już wiemy, typ klasowy `String`, który jest zdefiniowany w przestrzeni nazw `System`, w języku C++/CLI reprezentuje łańcuch znaków (w rzeczywistości łańcuch składa się ze znaków Unicode). Mówiąc dokładniej, reprezentuje łańcuch składający się z sekwencji znaków typu `System::Char`. Obiekty klasy `String` mają bardzo dużą funkcjonalność, dzięki czemu przetwarzanie łańcuchów jest niezwykle proste. Zacznijmy od tworzenia łańcucha.

Obiekt klasy `String` można utworzyć w następujący sposób:

```
System::String^ saying = L"Co dwie głowy, to nie jedna.";
```

Zmienna `saying` jest uchwytem śledzącym do obiektu klasy `String`, który został zainicjalizowany łańcuchem po prawej stronie znaku =. Wskaźniki do obiektów klasy `String` zawsze przechowuje się w uchwytach śledzących. Podany w powyższym przykładzie literał łańcu-

chowy jest typu `wide character`, ponieważ przed nim stoi litera `L`. Jeżeli nie podamy litery `L`, to otrzymamy literał łańcuchowy zawierający znaki ośmiobitowe, ale kompilator przekonwertuje je do łańcuchów typu `wide-character`.

Dostęp do poszczególnych znaków można uzyskać za pomocą indeksów, tak jak w tablicy. Pierwszy znak w łańcuchu ma indeks 0. Poniższa instrukcja wysyła na wyjście trzeci znak łańcucha `saying`:

```
Console::WriteLine("Trzecim znakiem w łańcuchu jest {0}", saying[2]);
```

Pamiętajmy, że choć za pomocą wartości indeksowych można uzyskać dostęp do określonego znaku w łańcuchu, to nie można jednak zmienić jego zawartości. Obiekty klasy `String` są stałe i nie można ich modyfikować.

Liczbę znaków danego łańcucha można sprawdzić za pomocą jego właściwości `Length`. Długość łańcucha `saying` możemy wyświetlić za pomocą następującej instrukcji:

```
Console::WriteLine("Łańcuch ma {0} znaków.", saying->Length);
```

Jako że `saying` jest uchwytem śledzącym (który — jak wiemy — jest rodzajem wskaźnika), aby uzyskać dostęp do właściwości `Length` (lub jakiejkolwiek innej składowej obiektu), musimy użyć operatora `->`. Więcej na temat właściwości dowiemy się przy okazji szczegółowego omawiania klas w C++/CLI.

Łączenie łańcuchów

Do łączenia łańcuchów znaków i tworzenia nowych obiektów klasy `String` możemy używać operatora +. Na przykład:

```
String^ name1 = L"Beth";
String^ name2 = L"Betty";
String^ name3 = name1 + L" i " + name2;
```

Po wykonaniu tych instrukcji zmienna `name3` zawiera łańcuch `Beth i Betty`. Zauważ, w jaki sposób można łączyć obiekty klasy `String` z literałami łańcuchowymi za pomocą operatora +. Łączyć można również obiekty klasy `String` z wartościami liczbowymi i logicznymi, które zostaną automatycznie przekonwertowane do łańcuchów przed operacją łączenia. Poniżej znajdują się instrukcje ilustrujące to zjawisko:

```
String^ str = L"Wartość: ";
String^ str1 = str + 2.5;    // Nowy łańcuch "Wartość: 2.5".
String^ str2 = str + 25;     // Nowy łańcuch "Wartość: 25".
String^ str3 = str + true;   // Nowy łańcuch "Wartość: True".
```

Można również połączyć łańcuch typu `String` ze znakiem, ale wynik będzie zależny od typu znaku:

```
char ch = 'Z';
wchar_t wch = 'Z'
String^ str4 = str + ch;     // Nowy łańcuch "Wartość: 90".
String^ str5 = str + wch;    // Nowy łańcuch "Wartość: Z".
```

W komentarzach podane zostały wyniki każdej instrukcji. Znak typu `char` traktowany jest jako wartość liczbowa i dlatego do łańcucha została dołączona liczba. Znak typu `w_char` jest tego samego typu co znaki w obiekcie klasy `String` (typ `Char`), a więc do łańcucha dołączony został znak.

Pamiętajmy, że obiekty łańcuchowe są stałe. Nie można zmieniać ich zawartości po ich utworzeniu. Oznacza to, że w wyniku wszelkich operacji mających na celu zmianę zawartości obiektów klasy `String` zawsze otrzymujemy nowy obiekt klasy `String`.

W klasie `String` zdefiniowana jest również funkcja `join()`, która służy do łączenia w jeden kilku łańcuchów przechowywanych w tablicy z uwzględnieniem znaków oddzielających poszczególne łańcuchy. Poniższa instrukcja łączy w jeden łańcuch imiona, oddzielając je przecinkami:

```
array<String^>^ names = { "Jill", "Ted", "Mary", "Eve", "Bill"};
String^ separator = ", ";
String^ joined = String::Join(separator, names);
```

Po wykonaniu powyższych instrukcji zmienna `joined` zawiera łańcuch `"Jill, Ted, Mary, Eve, Bill"`. Łańcuch `separator` został wstawiony pomiędzy każdą parą łańcuchów z tablicy `names`. Oczywiście łańcuch ten może być dowolny — może to być na przykład `i` i wtedy otrzymamy wynik `"Jill i Ted i Mary i Eve i Bill"`.

Spójrzmy teraz na przykładowy program z zastosowaniem obiektów klasy `String`.

spróbuj sam Praca z łańcuchami

Przypuśćmy, że mamy tablicę liczb całkowitych, której wartości chcemy zaprezentować wyrównane w kolumnach. Chcemy, aby wartości były wyrównane i aby kolumny były wystarczająco szerokie, ponieważ zależy nam na zmieszczeniu największych wartości tablicy włącznie z przestrzenią pomiędzy kolumnami. Poniższy program odpowiada tym wymaganiom.

// Cw4_17.cpp: main project file.

```
// Tworzenie własnego łańcucha formatu.

#include "stdafx.h"

using namespace System;

int main(array<System::String ^> ^args)
{
  array<int>^ values = { 2, 456, 23, -46, 34211, 456, 5609, 112098,
    234, -76504, 341, 6788, -909121, 99, 10};
  String^ formatStr1 = "{0,";          // Pierwsza połowa łańcucha formatu.
  String^ formatStr2 = "}";            // Druga połowa łańcucha formatu.
  String^ number;                      // Przechowywanie liczby jako łańcucha.

  // Sprawdź długość najdłuższego łańcucha.
  int maxLength = 0;                   // Przechowuje największą znalezioną liczbę.
  for each(int value in values)
  {
    number = "" + value;               // Utwórz łańcuch z wartości.
    if(maxLength<number->Length)
```

```
        maxLength = number->Length;
    }

    // Utwórz łańcuch formatu do użycia przy wysyłaniu danych na wyjście.
    String^ format = formatStr1 + (maxLength+1) + formatStr2;

    // Wyślij wartości.
    int numberPerLine = 3;
    for(int i = 0 ; i< values->Length ; i++)
    {
      Console::Write(format, values[i]);
      if((i+1)%numberPerLine == 0)
        Console::WriteLine();
    }
      return 0;
  }
```

Rezultat działania tego programu jest następujący:

```
       2     456      23
     -46   34211     456
    5609  112098     234
  -76504     341    6788
 -909121      99      10
```

Jak to działa

Celem tego programu jest utworzenie łańcucha formatującego wyrównującego liczby całkowite z tablicy `values` w kolumnach o szerokości wystarczającej dla najdłuższej z nich. Łańcuch formatujący zaczynamy tworzyć w dwóch częściach:

```
String^ formatStr1 = "{0,"; // Pierwsza połowa łańcucha formatującego.
String^ formatStr2 = "}";    // Druga połowa łańcucha formatującego.
```

Powyższe dwa łańcuchy stanowią początek i koniec łańcucha formatującego, który chcemy otrzymać na koniec. Aby go uzupełnić, musimy pomiędzy dwiema połowami `formatStr1` oraz `formatStr2` umieścić długość najdłuższego łańcucha reprezentującego liczbę.

Tę wartość odszukujemy za pomocą poniższego kodu:

```
int maxLength = 0;                          // Przechowuje najdłuższą znalezioną liczbę.
for each(int value in values)
{
   number = "" + value;                     // Utwórz łańcuch z wartości.
   if(maxLength<number->Length)
      maxLength = number->Length;
}
```

Wewnątrz pętli każda liczba konwertowana jest do typu łańcuchowego poprzez dołączenie jej do pustego łańcucha. Właściwość `Length` każdego łańcucha porównujemy ze zmienną `maxLength` i jeżeli dana wartość jest od niej większa, to zmienna `maxLength` przyjmuje tę właśnie wartość.

Tworzenie łańcucha formatującego jest bardzo proste:

```
String^ format = formatStr1 + (maxLength+1) + formatStr2;
```

Do zmiennej maxLength musimy dodać 1 w celu utworzenia dodatkowego pola, kiedy wyświetlany jest najdłuższy łańcuch. Umieszczenie wyrażenia maxLength + 1 w nawiasach daje gwarancję, że zostanie ono obliczone jako wyrażenie arytmetyczne przed operacją łączenia łańcuchów.

Na zakończenie wysyłamy na wyjście wartości z tablicy za pomocą łańcucha format:

```
int numberPerLine = 3;
for(int i = 0 ; i< values->Length ; i++)
{
   Console::Write(format, values[i]);
   if((i+1)%numberPerLine == 0)
      Console::WriteLine();
}
```

Instrukcja wyjściowa w pętli używa łańcucha format jako łańcucha do wysyłania. Dzięki zmiennej maxLength w łańcuchu format dane są umieszczone w kolumnach o szerokości o jeden większej niż długość najdłuższej z wysyłanych wartości. Zmienna numberPerLine określa, ile wartości pojawia się w jednym wierszu, dzięki czemu pętla jest dość elastyczna, gdyż pozwala na zmianę liczby kolumn poprzez zmianę wartości zmiennej numberPerLine.

Modyfikowanie łańcuchów

Najczęściej spotykaną operacją na łańcuchach jest obcinanie spacji znajdujących się z przodu i z tyłu. Do tego celu służy funkcja Trim():

```
String^ str = {" Nie szata zdobi człowieka... "};
String^ newStr = str->Trim();
```

Funkcja Trim() w drugiej instrukcji usuwa wszystkie spacje z przodu i z tyłu łańcucha str i zwraca wynik w postaci nowego obiektu klasy String przechowywanego w zmiennej newStr. Oczywiście, jeżeli nie chcemy zachowywać oryginalnego łańcucha, możemy wynik zapisać z powrotem do zmiennej str.

Istnieje także inna wersja funkcji Trim(), która pozwala na określenie znaków do usunięcia z początku i końca łańcucha. Funkcja ta jest bardzo elastyczna, ponieważ umożliwia określenie znaków do usunięcia na więcej niż jeden sposób. Znaki te można zapisać do tablicy i uchwyt do niej przekazać jako argument do funkcji:

```
String^ toBeTrimmed = L"wełna wełna owca owca wełna wełna wełna";
array<wchar_t>^ notWanted = {L'w',L'e',L'ł', L'n', L'a', L' '};
Console::WriteLine(toBeTrimmed->Trim(notWanted));
```

W powyższym przykładzie mamy łańcuch o nazwie toBeTrimmed, który zawiera owcę „przykrytą" wełną. Tablica znaków do usunięcia z łańcucha została zdefiniowana pod nazwą notWanted, a więc przekazanie jej do funkcji Trim() zastosowanej dla łańcucha spowoduje usunięcie z jego końca i początku wszystkich znaków w niej podanych. Pamiętaj, że obiekty klasy String są stałe, a więc oryginalny łańcuch nie zostanie w żaden sposób zmieniony — w wyniku działania funkcji Trim() tworzony jest nowy łańcuch, który jest następnie zwracany. Wykonanie powyższego fragmentu kodu da następujący rezultat:

```
owca owca
```

Jeżeli literały znakowe podalibyśmy bez towarzyszącego im przedrostka `L`, to byłyby one typu `char` (który odpowiada typowi klasy wartości `SByte`). Mimo to kompilator sam zadba o ich konwersję do typu `wchar_t`.

Znaki do usunięcia przez funkcję `Trim()` można także podać wprost jako argumenty tej funkcji. W związku z tym ostatni wiersz poprzedniego fragmentu kodu moglibyśmy zapisać następująco:

```
Console::WriteLine(toBeTrimmed->Trim(L'w',L'e',L'?', L'n', L'a', L' '));
```

Kod ten da taki sam wynik jak poprzednia wersja tej instrukcji. Liczba argumentów typu `wchar_t` jest dowolna, choć jeśli jest ich bardzo dużo, to lepiej zdefiniować je w tablicy.

Jeżeli chcemy usunąć znaki tylko z jednej strony łańcucha, to możemy użyć funkcji `TrimEnd()` lub `TrimStart()`. Funkcje te występują w takich samych wersjach jak funkcja `Trim()`, a więc jeżeli nie podamy żadnych argumentów, to usunięte zostaną spacje. Jeżeli jako argument podamy uchwyt do tablicy, to usunięte zostaną znaki w niej zdefiniowane. Znaki do usunięcia można również podać wprost jako argumenty typu `wchar_t` funkcji.

Działaniem przeciwnym do usuwania znaków z łańcucha jest jego dopełnianie z obu stron spacjami lub innymi znakami. Dostępne są funkcje `PadLeft()` i `PadRight()`, które dopełniają łańcuch odpowiednio z lewej i prawej strony. Głównym zastosowaniem tych funkcji jest formatowanie wysyłanych na wyjście danych, gdy chcemy je wyrównać do prawej lub lewej strony w polach o ustalonej szerokości. Prostsze wersje funkcji `PadLeft()` oraz `PadRight()` akceptują pojedyncze argumenty określające długość łańcucha powstałego w wyniku operacji. Na przykład:

```
String^ value = L"3.142";
String^ leftPadded = value->PadLeft(10);    // Wynik to "    3.142".
String^ rightPadded = value->PadRight(10); // Wynik to "3.142    ".
```

Jeżeli długość łańcucha, podana jako argument funkcji, jest równa lub mniejsza niż długość oryginalnego łańcucha, to obie funkcje zwrócą nowy obiekt klasy `String` identyczny z oryginałem.

Aby dopełnić łańcuch znakiem innym niż spacja, należy go podać jako drugi argument funkcji. Poniżej znajduje się kilka przykładów takiego dopełniania:

```
String^ value = L"3.142";
String^ leftPadded = value->PadLeft(10, L'*');    // Wynik to "*****3.142".
String^ rightPadded = value->PadRight(10, L'#'); // Wynik to "3.142#####".
```

Oczywiście w każdym z powyższych przykładów moglibyśmy zapisać powstały łańcuch z powrotem do uchwytu do oryginalnego łańcucha, co spowodowałoby usunięcie oryginalnego łańcucha.

W klasie `String` dostępne są także funkcje `ToUpper()` oraz `ToLower()` zamieniające wszystkie litery w łańcuchu na wielkie lub małe. Spójrzmy na przykładowy kod z ich wykorzystaniem:

```
String^ proverb = L"Co dwie głowy, to nie jedna.";
String^ upper = proverb->ToUpper(); // Wynik: "CO DWIE GŁOWY, TO NIE JEDNA".
```

Funkcja `toUpper()` zwraca nowy łańcuch, który jest kopią oryginalnego, ale z wszystkimi literami wielkimi.

Funkcji `Insert()` używamy do wstawiania łańcuchów w określone miejsce w istniejącym już łańcuchu. Poniżej znajduje się przykład zastosowania tej funkcji:

```
String^ proverb = L"Co dwie głowy, to nie jedna.";
String^ newProverb = proverb->Insert(8, L"mądre ");
```

Funkcja ta wstawia łańcuch podany jako drugi argument w miejscu, którego początek został określony w starym łańcuchu przez pierwszy argument. W wyniku działania tego kodu powstanie nowy łańcuch:

```
Co dwie mądre głowy, to nie jedna.
```

Można także wszystkie wystąpienia jednego znaku zastąpić innym znakiem lub wszystkie wystąpienia jednego fragmentu łańcucha zastąpić innym fragmentem. W poniższym przykładzie wykonywane są oba rodzaje operacji:

```
String^ proverb = L"Co dwie głowy, to nie jedna.";
Console::WriteLine(proverb->Replace(L' ', L'*'));
Console::WriteLine(proverb->Replace(L"Co dwie", L"Co trzy"));
```

Wykonanie powyższego fragmentu kodu da następujący rezultat:

```
Co*dwie*głowy*to*nie*jedna.
Co trzy głowy, to nie jedna.
```

Pierwszy argument funkcji `Replace()` określa znak lub fragment łańcucha, który ma zostać zastąpiony, a drugi argument to, co ma zostać wstawione w zamian.

Przeszukiwanie łańcuchów

Jedną z najprostszych operacji przeszukiwania łańcucha jest sprawdzenie, czy na jego końcu lub początku znajduje się określony fragment łańcucha. Służą do tego funkcje `StartsWith()` oraz `EndsWith()`. Do każdej z tych funkcji należy przekazać uchwyt do poszukiwanego łańcucha. Funkcje zwrócą wartość logiczną określającą, czy łańcuch został odnaleziony, czy nie. Poniżej znajduje się przykład użycia funkcji `StartsWith()`:

```
String^ sentence = L"Krowy to miłe zwierzęta.";
if(sentence->StartsWith(L"Krowy"))
   Console::WriteLine("Zdanie rozpoczyna się słowem 'Krowy'.");
```

Wykonanie powyższego fragmentu kodu da następujący rezultat:

```
Zdanie rozpoczyna się słowem 'Krowy'.
```

Oczywiście do tego samego łańcucha możemy zastosować funkcję `EndsWith()`:

```
Console::WriteLine("Zdanie {0} kończy się słowem 'zwierzęta'.",
                                  sentence->EndsWith(L"zwierzęta") ? L"" : L" nie");
```

Do łańcucha wyjściowego wstawiony został wynik wyrażenia z użyciem operatora warunkowego. Jeżeli funkcja `EndsWith()` zwróci wartość `true`, to wstawiony zostanie pusty łańcuch, a jeżeli `false`, to wstawiony zostanie łańcuch `nie`. W tym przypadku funkcja zwróci wartość `false` (ponieważ na końcu łańcucha znajduje się jeszcze kropka).

Funkcja `IndexOf()` przeszukuje łańcuch w celu odnalezienia pierwszego wystąpienia określonego znaku lub łańcucha oraz zwraca indeks, jeżeli go odnajdzie, lub `-1` w przeciwnym przypadku. Poszukiwany znak lub łańcuch określamy jako argument funkcji. Na przykład:

```
String^ sentence = L"Krowy to fajne zwierzęta.";
int ePosition = sentence->IndexOf(L'w');      // Zwraca 3.
int thePosition = sentence->IndexOf(L"to"); // Zwraca 6.
```

Pierwsze wyszukiwanie dotyczy litery `w`, a drugie słowa `to`. Wartości zwrócone przez funkcję `IndexOf()` podane zostały w komentarzach.

Częściej jednak chcemy znaleźć wszystkie miejsca wystąpienia danego znaku lub łańcucha. Do tego celu służy inna wersja funkcji `IndexOf()`, której można używać wielokrotnie. W tym przypadku podajemy drugi argument, określający miejsce, od którego ma się rozpocząć poszukiwanie. Poniżej znajduje się przykład użycia tej funkcji:

```
String^ words = "wełna wełna owca owca wełna wełna wełna";
String^ word = "wełna";
int index = 0;
int count = 0;
while((index = words->IndexOf(word,index)) >= 0)
{
   index += word->Length;
   ++count;
}
Console::WriteLine(L"Słowo '{0}' zostało znalezione {1} razy w:\n{2}", word, count,
words);
```

Powyższy fragment kodu liczy, ile razy w łańcuchu `words` wystąpiło słowo `wełna`. Operacja poszukiwania znajduje się w warunku pętli `while`, a wynik przechowywany jest w zmiennej `index`. Pętla powtarza się, dopóki zmienna `index` nie ma wartości ujemnej, czyli aż do momentu, kiedy funkcja `IndexOf()` zwróci `-1`. Wartość zmiennej `index` zwiększana jest wewnątrz ciała pętli o długość słowa `word`, co powoduje przesunięcie pozycji indeksu do znaku znajdującego się po znalezionym słowie i pętla gotowa jest do nowej iteracji. Zmienna `count` wewnątrz pętli jest zwiększana za każdym razem, kiedy odnajdywane jest szukane słowo, dzięki czemu przechowuje ona liczbę odnalezionych słów `word` w łańcuchu `words`. Wykonanie powyższego fragmentu kodu da następujący rezultat:

```
Słowo 'wełna' zostało znalezione 5 razy w:
wełna wełna owca owca wełna wełna wełna
```

Funkcja `LastIndexOf()` jest podobna do funkcji `IndexOf()`, z tym wyjątkiem, że rozpoczyna przeszukiwanie od tyłu lub wstecz od określonego indeksu. Poniższy kod wykonuje tę samą czynność co poprzedni za pomocą funkcji `LastIndexOf()`:

```
int index = words->Length - 1;
int count = 0;
while(index >= 0 && (index = words->LastIndexOf(word,index)) >= 0)
```

```
{
   --index;
   ++count;
}
```

Przy użyciu tych samych łańcuchów `word` i `words` co wcześniej, kod da identyczny rezultat. Jako że funkcja `LastIndexOf()` przeszukuje wstecz, to indeks, od którego ma zacząć przeszukiwanie, określa ostatni znak łańcucha — `words->Length-1`. Kiedy zostaje znalezione słowo `word`, zmniejszamy `index` o jeden, dzięki czemu następne poszukiwanie rozpocznie się od znaku poprzedzającego bieżące słowo `word`. Jeżeli poszukiwane słowo znajduje się na samym początku łańcucha `words` (w pozycji indeksowej 0), zmniejszenie wartości zmiennej `index` spowoduje ustawienie jej na wartość `-1`. Taka wartość nie jest prawidłowym argumentem funkcji `LastIndexOf`, ponieważ przeszukiwanie zawsze musi rozpoczynać się od jakiegoś miejsca w łańcuchu. Dodatkowe sprawdzanie wartości ujemnych zmiennej `index` w warunku pętli zapobiega wystąpieniu takiej sytuacji. Jeżeli prawy operand operatora `&&` ma wartość `false`, to wartość lewego nie jest sprawdzana.

Ostatnia funkcja przeszukiwania, o której chcę napisać, to funkcja `IndexOfAny()`. Przeszukuje ona łańcuch w celu odnalezienia pierwszego miejsca wystąpienia jakiegokolwiek znaku w tablicy typu `array<wchar_t>`, którą podajemy jako argument. Podobnie jak funkcja `IndexOf()`, funkcja `IndexOfAny()` może rozpocząć przeszukiwanie łańcucha od samego początku lub od określonego miejsca. Poniżej znajduje się przykładowy program z wykorzystaniem funkcji `IndexOfAny()`.

spróbuj sam Poszukiwanie jednego z kilku znaków

Poniższy program poszukuje znaków interpunkcyjnych w ciągu:

```
// Cw4_18.cpp: main project file.
// Poszukiwanie znaków interpunkcyjnych.

#include "stdafx.h"

using namespace System;

int main(array<System::String ^> ^args)
{
  array<wchar_t>^ punctuation = {L'"', L'\'', L'.', L',', L':', L';', L'!', L'?'};
  String^ sentence = L"\"Zimno tu\", chłodno powiedziała matka do synka.";

  // Utwórz tablicę spacji o takiej samej długości jak zdanie.
  array<wchar_t>^ indicators = gcnew array<wchar_t>(sentence->Length){L' '};

  int index = 0;                          // Liczba znalezionych znaków.
  int count = 0;                          // Liczba znaków interpunkcyjnych.
  while((index = sentence->IndexOfAny(punctuation, index)) >= 0)
  {
    indicators[index] = L'^';             // Ustaw znacznik.
    ++index;                              // Zwiększ do następnego znaku.
    ++count;                              // Zwiększ licznik.
  }
```

```
    Console::WriteLine(L"W łańcuchu są {0} znaki interpunkcyjne:", count);
    Console::WriteLine(L"\n{0}\n{1}", sentence, gcnew String(indicators));
    return 0;
}
```

Wynik działania powyższego programu powinien być następujący:

```
W łańcuchu są 4 znaki interpunkcyjne:
"Zimno tu", chłodno powiedziała matka do synka.
^        ^^                                  ^
```

Jak to działa

Najpierw tworzymy tablicę znaków do odnalezienia oraz łańcuch, który chcemy przeszukać:

```
array<wchar_t>^ punctuation = {L'"', L'\'', L'.', L',', L':', L';', L'!', L'?'};
String^ sentence = L"\"Zimno tu\", chłodno powiedziała matka do synka.";
```

Zauważmy, że znak pojedynczego cudzysłowu musieliśmy zapisać za pomocą kodu znaku specjalnego, gdyż pełni on rolę ogranicznika literałów znakowych. W literałach znakowych możemy stosować podwójne cudzysłowy bez kodowania, ponieważ nie ma ryzyka, że w tym kontekście zostaną one zinterpretowane jako znaki oddzielające.

Następnie definiujemy tablicę znaków, której elementy zostały zainicjalizowane znakiem spacji:

```
array<wchar_t>^ indicators = gcnew array<wchar_t>(sentence->Length){L' '};
```

Tablica ta ma tyle elementów, ile znaków jest w łańcuchu `sentence`. Tablicy tej w danych wyjściowych będziemy używać do zaznaczania, gdzie w łańcuchu `sentence` znajdują się znaki interpunkcyjne. Za każdym razem, gdy zostanie odnaleziony znak interpunkcyjny, odpowiedni element tablicy przyjmuje wartość ^. Zauważ, że pojedynczy inicjalizator umieszczony w nawiasach klamrowych po specyfikacji tablicy może być używany do inicjalizowania wszystkich elementów tablicy.

Poszukiwanie odbywa się w pętli `while`:

```
while((index = sentence->IndexOfAny(punctuation, index)) >= 0)
{
  indicators[index] = L'^';          // Ustaw znacznik.
  ++index;                           // Zwiększ do następnego znaku.
  ++count;                           // Zwiększ licznik.
}
```

Warunek pętli jest w zasadzie taki sam jak w poprzednich fragmentach kodu. Wewnątrz pętli wartość elementu znajdującego się w miejscu `index` w tablicy `indicators` jest zmieniana na ^ przed zwiększeniem indeksu przed następną iteracją. Po zakończeniu pętli zmienna `count` zawiera liczbę znalezionych znaków interpunkcyjnych, a tablica `indicators` zawiera znaki ^ w tych miejscach, gdzie zostały one znalezione.

Dane na wyjście wysyłane są za pomocą następujących instrukcji:

```
Console::WriteLine(L"W łańcuchu są {0} znaki interpunkcyjne:", count);
Console::WriteLine(L"\n{0}\n{1}" sentence, gcnew String(indicators));
```

Druga z powyższych instrukcji tworzy nowy obiekt klasy `String` na stercie z tablicy `indicators` poprzez przekazanie tablicy do **konstruktora** klasy `String`. Konstruktor klasy to funkcja tworząca nowy obiekt klasy w momencie jego wywołania. Więcej na temat konstruktorów dowiesz się, kiedy dojdziemy do definiowania własnych klas.

Referencje śledzące

Referencje śledzące są podobne do referencji w natywnym C++ pod tym względem, że stanowią alias czegoś znajdującego się na stercie. Można je tworzyć do typów wartości na stosie i do uchwytów na oczyszczonej stercie. Same referencje śledzące zawsze tworzone są na stosie, a ich zawartość jest automatycznie uaktualniana, gdy obiekt przez nie wskazywany zostanie przesunięty przez mechanizm usuwający nieużytki.

Referencję śledzącą definiujemy za pomocą operatora `%`. Poniższy przykład tworzy referencję śledzącą do typu wartości:

```
int value = 10;
int% trackValue = value;
```

Druga z powyższych instrukcji definiuje referencję śledzącą o nazwie `trackValue` do zmiennej `value`, która została utworzona na stosie. Za pomocą referencji `trackValue` możemy teraz modyfikować zmienną `value`:

```
trackValue *= 5;
Console::WriteLine(value);
```

Jako że referencja śledząca `trackValue` jest aliasem zmiennej `value`, druga z powyższych instrukcji zwróci wartość `50`.

Wskaźniki wewnętrzne

Mimo że adresów przechowywanych przez uchwyty śledzące nie można używać w działaniach arytmetycznych, w języku C++/CLI istnieje rodzaj wskaźnika, który do tego celu może być używany. Nazywa się on **wskaźnikiem wewnętrznym** i definiuje się go za pomocą słowa kluczowego `interior_ptr`. Adres przez niego przechowywany może być w razie potrzeby automatycznie aktualizowany przez system zbierający nieużytki. Wskaźnikiem wewnętrznym jest zawsze zmienna automatyczna o zasięgu lokalnym w funkcji.

Poniżej znajduje się przykładowa definicja wskaźnika wewnętrznego zawierającego adres pierwszego elementu tablicy:

```
array<double>^ data = {1.5, 3.5, 6.7, 4.2, 2.1};
interior_ptr<double> pstart = &data[0];
```

Obiekt wskazywany przez wskaźnik wewnętrzny podajemy w nawiasach trójkątnych po słowie kluczowym `interior_ptr`. W drugiej z powyższych instrukcji wskaźnik wewnętrzny inicjalizujemy adresem pierwszego elementu tablicy za pomocą operatora `&`, podobnie jak w przypadku wskaźników w natywnym C++. Jeżeli nie podamy wartości początkowej wskaźnika,

to domyślnie zostanie on zainicjalizowany wartością `nullptr`. Pamięć dla tablicy jest zawsze przydzielana na stercie, a więc jest to sytuacja, w której system usuwania nieużytków może dostosować adres zawarty we wskaźniku wewnętrznym.

Istnieją ograniczenia dotyczące określania typu wskaźnika wewnętrznego. Wskaźnik wewnętrzny może zawierać adres obiektu klasy wartości na stosie lub adres uchwytu do obiektu na stercie CLR. Nie może zawierać adresu całego obiektu na stercie CLR. Wskaźnik wewnętrzny może także wskazywać natywny obiekt klasy lub wskaźnik natywny.

Wskaźnika wewnętrznego można również użyć do przechowywania adresu obiektu klasy wartości, który jest częścią obiektu na stercie, takiego jak np. elementu tablicy CLR. W ten sposób możemy utworzyć wskaźnik wewnętrzny przechowujący adres uchwytu śledzącego do obiektu `System::String`, ale nie możemy utworzyć wskaźnika wewnętrznego do przechowywania adresu samego obiektu klasy `String`. Na przykład:

```
interior_ptr<String^> pstr1; // OK — wskaźnik do uchwytu.
interior_ptr<String> pstr2;  // Nie skompiluje się — wskaźnik obiektu String.
```

Z zastosowaniem wskaźnika wewnętrznego można wykonywać takie same działania arytmetyczne co ze wskaźnikami natywnymi. Można także użyć operatora inkrementacji lub dekrementacji w celu zmiany zawartego w nim adresu na poprzedni lub następny element. Można również dodawać i odejmować liczby całkowite oraz porównywać wskaźniki wewnętrzne. Spójrzmy na przykład z zastosowaniem tego, co zostało opisane.

spróbuj sam Tworzenie i używanie wskaźników wewnętrznych

Poniższy program jest ćwiczeniem zastosowania wskaźników wewnętrznych z wartościami liczbowymi i łańcuchami.

```
// Cw4_19.cpp: main project file.
// Tworzenie i używanie wskaźników wewnętrznych.

#include "stdafx.h"

using namespace System;

int main(array<System::String ^> ^args)
{
  // Uzyskaj dostęp do elementów tablicy za pomocą wskaźnika.
  array<double>^ data = {1.5, 3.5, 6.7, 4.2, 2.1};
  interior_ptr<double> pstart = &data[0];
  interior_ptr<double> pend = &data[data->Length - 1];
  double sum = 0.0;
  while(pstart <= pend)
    sum += *pstart++;

  Console::WriteLine(L"Łączna suma elementów tablicy data = {0}\n", sum);

  // W celu pokazania, że jest to możliwe, uzyskaj dostęp do łańcuchów
  // za pomocą wskaźnika wewnętrznego.
  array<String^>^ strings = { L"Ahoj, widać ląd!",
                              L"Wychyl kielicha!",
```

```
                              L"Nie trzęś się!",
                              L"Nie rzucaj słów na wiatr!"
                            };

   for(interior_ptr<String^> pstrings = &strings[0] ;
              pstrings-&strings[0] < strings->Length ; ++pstrings)
     Console::WriteLine(*pstrings);

   return 0;
}
```

Rezultat działania tego programu jest następujący:

```
Łączna suma elementów tablicy data = 18
Ahoj, widać ląd!
Wychyl kielicha!
Nie trzęś się!
Nie rzucaj słów na wiatr!
```

Jak to działa

Po utworzeniu tablicy `data` zawierającej elementy typu `double` definiujemy dwa wskaźniki wewnętrzne:

```
interior_ptr<double> pstart = &data[0];
interior_ptr<double> pend = &data[data->Length - 1];
```

Pierwsza z powyższych instrukcji tworzy wskaźnik `pstart` do typu `double` i inicjalizuje go adresem pierwszego elementu tablicy — `data[0]`. Wskaźnik `pend` został zainicjalizowany adresem ostatniego elementu tablicy — `data[data->Length -1]`. Jako że wyrażenie `data->Length` oznacza liczbę elementów tablicy, odjęcie jeden od jego wartości daje w wyniku indeks ostatniego jej elementu.

Pętla `while` oblicza sumę elementów tablicy:

```
while(pstart <= pend)
   sum += *pstart++;
```

Pętla będzie się powtarzać tak długo, aż wskaźnik wewnętrzny `pstart` będzie zawierał adres nie większy od adresu we wskaźniku `pend`. Warunek pętli mogliśmy równie dobrze zapisać w następujący sposób: `!pstart > pend`.

Na początku działania pętli wskaźnik `pstart` zawiera adres pierwszego elementu tablicy. Wartość pierwszego elementu uzyskujemy poprzez wyłuskanie wskaźnika za pomocą wyrażenia `*pstart`, a jego wynik dodajemy do zmiennej `sum`. Następnie adres we wskaźniku jest zwiększany o jeden za pomocą operatora `++`. Przy ostatniej iteracji pętli wskaźnik `pstart` zawiera adres ostatniego elementu, czyli taki sam jak wskaźnik `pend`. W związku z tym zwiększenie wskaźnika `pstart` sprawia, że warunek pętli daje wynik `false`, ponieważ wskaźnik `pstart` stał się większy niż `pend`. Po zakończeniu działania pętli wartość zmiennej `sum` wysłana zostaje na wyjście, dzięki czemu mamy potwierdzenie, że pętla `while` działa tak, jak powinna.

Następnie tworzymy tablicę czterech łańcuchów:

```
array<String^>^ strings = { L"Ahoj, widać ląd!",
                            L"Wychyl kielicha!",
                            L"Nie trzęś się!",
                            L"Nie rzucaj słów na wiatr!"
                          };
```

Pętla `for` wysyła każdy z tych łańcuchów do wiersza poleceń:

```
for(interior_ptr<String^> pstrings = &strings[0] ;
          pstrings-&strings[0] < strings->Length ; ++pstrings)
  Console::WriteLine(*pstrings);
```

Pierwsze wyrażenie w warunku pętli `for` deklaruje wskaźnik wewnętrzny `pstrings` i inicjalizuje go adresem pierwszego elementu tablicy `strings`. Drugie wyrażenie, które decyduje, czy pętla kontynuuje działanie, to:

```
pstrings-&strings[0] < strings->Length
```

Dopóki wskaźnik `pstrings` zawiera adres prawidłowego elementu tablicy, różnica pomiędzy tym adresem a adresem pierwszego elementu w tablicy jest mniejsza niż liczba elementów w tablicy zwrócona przez wyrażenie `strings->Length`. Kiedy różnica ta jest równa liczbie elementów w tablicy, działanie pętli zostaje zakończone. Z danych wyjściowych wynika, że wszystko działa zgodnie z przewidywaniami.

Wskaźników wewnętrznych najczęściej używa się do wskazywania obiektów, które są częścią obiektów na stercie CLR. Więcej na ten temat będziemy mówić w dalszej części książki.

Podsumowanie

Znamy już wszystkie podstawowe typy wartości w C++, potrafimy tworzyć tablice tych typów i ich używać oraz tworzyć wskaźniki i z nich korzystać. Wspomnieliśmy także o referencjach. Oczywiście, nie powiedzieliśmy jeszcze wszystkiego na te tematy. Do tematów tablic, wskaźników i referencji wrócimy jeszcze w dalszej części książki. Poniżej znajduje się lista najważniejszych zagadnień poruszonych w tym rozdziale:

- Dzięki tablicom możemy zarządzać zbiorem danych tego samego typu, posługując się prostą pojedynczą nazwą. Każdy wymiar tablicy definiowany jest pomiędzy nawiasami kwadratowymi po nazwie tablicy w jej deklaracji.
- Każdy wymiar tablicy indeksowany jest od zera. W związku z tym piąty element tablicy ma indeks cztery.
- Tablice można inicjalizować, wstawiając w deklaracji wartości początkowe umieszczone pomiędzy nawiasami klamrowymi.
- Wskaźnik jest typem zmiennej, która zawiera adres innej zmiennej. Wskaźniki deklaruje się jako „wskaźniki do typu” i można im przypisywać tylko adresy zmiennych o podanym typie.

- Wskaźnik może wskazywać obiekt stały. Można go ponownie przypisać do innego obiektu. Wskaźnik można również zdefiniować jako `const` i w takim przypadku nie można go ponownie przypisać.
- Referencja jest aliasem zmiennej i może być używana w tych samych miejscach co zmienna, którą wskazuje. Referencja musi zostać zainicjalizowana w momencie deklaracji.
- Raz przypisanej referencji nie można przypisać do innej zmiennej.
- Operator `sizeof` zwraca liczbę bajtów zajmowanych przez obiekt podany jako jego argument. Jego argumentem może być zmienna lub nazwa typu otoczona nawiasami okrągłymi.
- Operator `new` dynamicznie przydziela pamięć w wolnej przestrzeni w programach w natywnym C++. Po przydzieleniu pamięci zwraca wskaźnik do początku przydzielonego obszaru. Jeżeli pamięć z jakiegoś powodu nie może zostać przydzielona, powstaje wyjątek i program zostaje zamknięty.

Mechanizm działania wskaźnika może być czasami trudny do zrozumienia, gdyż operuje on na różnych poziomach w jednym programie. Czasami działa jako adres, a czasami może działać z wartościami przechowywanymi pod danym adresem. Bardzo ważne jest, aby dobrze zrozumieć istotę tego mechanizmu, a więc mając jakiekolwiek problemy ze zrozumieniem sposobu działania wskaźników, należy przećwiczyć ich użycie na kilku przykładach, aby sprawnie się nimi posługiwać.

Najważniejsze zagadnienia, których nauczyliśmy się o programowaniu dla CLR, to:

- W programach dla CLR pamięć przydzielana jest na oczyszczonej stercie za pomocą operatora `gcnew`.
- Obiektom klasy referencji, a w szczególności obiektom klasy `String` pamięć przydzielana jest zawsze na stercie CLR.
- Pracując w programach CLR, używamy obiektów klasy `String`.
- CLR posiada własne typy tablicowe posiadające większą funkcjonalność niż typy tablicowe w natywnym C++.
- Tablice CLR zawsze są tworzone na stercie CLR.
- Uchwyt śledzący jest typem wskaźnika używanego do wskazywania zmiennych zdefiniowanych na stercie CLR. Uchwyt śledzący jest automatycznie aktualizowany, jeżeli to, do czego się odwołuje, zostało przeniesione przez mechanizm usuwania nieużytków.
- Zmienne odnoszące się do obiektów i tablic na stercie są zawsze uchwytami śledzącymi.
- Referencja śledząca podobna jest do referencji w natywnym C++, z tym wyjątkiem, że zawarty w niej adres jest automatycznie aktualizowany, jeżeli obiekt przez nią wskazywany zostanie przeniesiony przez mechanizm usuwania nieużytków.
- Wskaźnik wewnętrzny to typ wskaźnika C++/CLI. Można go stosować do wykonywania tych samych operacji, które wykonuje się za pomocą wskaźnika natywnego.

- Adres zawarty we wskaźniku wewnętrznym można modyfikować za pomocą działań arytmetycznych i nadal utrzymać poprawny adres, nawet odnosząc się do czegoś przechowywanego na stercie CLR.

Ćwiczenia

Kod źródłowy wszystkich przykładów w tej książce oraz rozwiązania do ćwiczeń można pobrać ze strony *www.helion.pl*.

1. Napisz program w natywnym C++ pozwalający na podanie dowolnego zbioru liczb, które będą przechowywane w tablicy ulokowanej w obszarze pamięci wolnej. Program powinien wysyłać na wyjście po pięć z podanych liczb, a na końcu podawać ich średnią. Początkowo tablica powinna mieć rozmiar pięciu elementów. W razie potrzeby program powinien tworzyć nową tablicę z dodatkowymi pięcioma elementami oraz kopiować wartości ze starej tablicy do nowej.
2. Powtórz poprzednie ćwiczenie, ale z użyciem notacji wskaźnikowej zamiast tablic.
3. Zadeklaruj tablicę znaków i zainicjalizuj ją do odpowiedniego łańcucha. Za pomocą pętli zamień co drugą literę na wielką.

   Wskazówka: w zestawie znaków ASCII wartości wielkich liter są o 32 mniejsze niż ich małych odpowiedników.
4. Napisz program w C++/CLI tworzący tablicę zawierającą losową liczbę elementów typu `int`. Tablica powinna zawierać nie mniej niż 10 i nie więcej niż 20 elementów. Wartości elementów powinny być także losowe i zawierać się w zbiorze od 100 do 1000. Zawartość elementów wyświetl w porządku malejącym bez sortowania tablicy. Na przykład znajdź najmniejszy element i go wyświetl, następnie kolejny najmniejszy itd.
5. Napisz program w C++/CLI generujący losową liczbę całkowitą większą od 10 000. Wyświetl tę liczbę na ekranie, a następnie wyświetl słowne odpowiedniki poszczególnych cyfr. Jeżeli na przykład program wygenerował liczbę 345 678, to wynik powinien być następujący:

   ```
   Wartość wygenerowana to: 345678
   trzy cztery pięć sześć siedem osiem
   ```
6. Napisz program w C++/CLI, który stworzy tablicę zawierającą następujące łańcuchy:

   ```
   "Kobyła ma mały bok."
   "Kawa i wuzetka to zestaw obowiązkowy."
   "Jeż leje lwa, paw leje lżej."
   "Ala ma kota, kot ma Alę."
   "Pętaka pętaj, a tępaka tęp."
   ```

   Program powinien przeanalizować po kolei wszystkie łańcuchy i wyświetlić je z informacją, czy są one palindromami, czy nie (tzn. czy nie ma różnicy, czy się je czyta od przodu, czy od tyłu, pomijając znaki interpunkcyjne).