Wydanie III

Wprowadzenie do C++

Efektywne nauczanie

Cay Horstmann

WILEY

Helion

Tytuł oryginału: Big C++: Late Objects, 3rd Edition

Tłumaczenie: Krzysztof Bąbol

ISBN: 978-83-283-6728-9

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/wpcpp3
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/wpcpp3.zip

Printed in Poland.

Spis treści

ALFABETYCZNA LISTA SKŁADNI

Wskazówka dla programistów

Temat specjalny

Informatyka a społeczeństwo

Wskazówka dla programistów

Temat specjalny

Informatyka a społeczeństwo

Wskazówka dla programistów

Temat specjalny

Informatyka a społeczeństwo

6

Tablice i wektory

W wielu programach trzeba gromadzić dużą liczbę danych. Standard C++ pozwala na używanie w tym celu tablic i wektorów. Tablice są podstawową strukturą języka. Standardowa biblioteka C++ zawiera konstrukcję `vector`, która przy pracy z kolekcjami o nieznanym rozmiarze jest wygodniejszą alternatywą dla tablic. W rozdziale tym poznasz tablice i wektory oraz popularne algorytmy służące do ich przetwarzania.

Cele rozdziału:

- Dowiesz się, jak gromadzić wartości w tablicach i wektorach.
- Poznasz najczęściej używane algorytmy przetwarzania tablic i wektorów.
- Napiszesz funkcje przetwarzające tablice i wektory.
- Nauczysz się używać tablic dwuwymiarowych.

6.1. Tablice

Na początku tego rozdziału przedstawimy typ danych zwany **tablicą** (ang. *array*). Tablice stanowią w języku C++ podstawowy mechanizm gromadzenia wielu wartości. W następnych punktach dowiesz się, jak definiować tablice i jak uzyskiwać dostęp do ich elementów.

6.1.1. Definiowanie tablic

Wyobraź sobie, że piszesz program, który odczytuje serię wartości, a potem je wyświetla, oznaczając tę, która jest z nich największa, np. tak:

```
32
54
67.5
29
34.5
80
115 <= największa liczba
```

```
44.5
100
65
```

Przed poznaniem wszystkich liczb nie wiadomo, którą z nich oznaczyć jako największą. Przecież może to być ostatnia. Przed wypisaniem liczb należy więc najpierw zapamiętać wszystkie w programie.

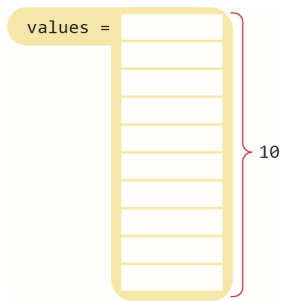
Czy można po prostu zapisać każdą wartość w oddzielnej zmiennej? Jeśli wiadomo, że danych wejściowych będzie dziesięć, można je zapisać w dziesięciu zmiennych `value1`, `value2`, `value3`, …, `value10`. Taki zestaw zmiennych nie jest jednak zbyt praktyczny w użyciu. Trzeba by było dziesięć razy pisać całkiem spory kawał kodu dla każdej ze zmiennych. Aby rozwiązać ten problem, należy użyć tablicy, struktury przeznaczonej do gromadzenia sekwencji wartości.

Tak definiujemy tablicę do przechowywania dziesięciu liczb:

```
double values[10];
```

Tablica służy do zbierania sekwencji wartości tego samego typu.

To definicja zmiennej `values` typu „tablica wartości `double`". Oznacza to, że zmienna `values` przechowuje serię liczb zmiennoprzecinkowych. Zapis `[10]` to określenie *rozmiaru* tablicy (rysunek 1). Rozmiar tablicy musi być zmienną znaną w czasie kompilacji.



Rysunek 1. Tablica o rozmiarze 10

Podczas definiowania tablicy można podać jej wartości początkowe, np. tak:

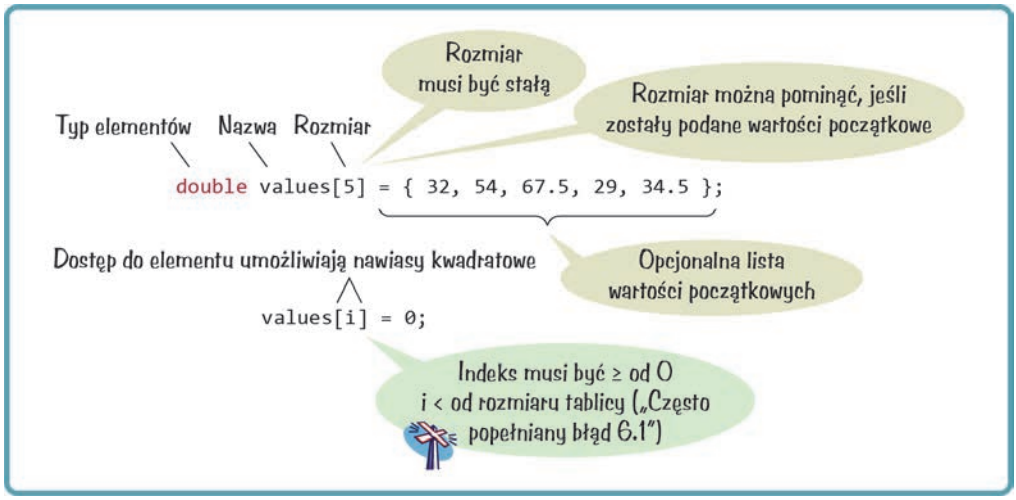
```
double values[] = { 32, 54, 67.5, 29, 34.5, 80, 115, 44.5, 100, 65 };
```

Jeśli zostały podane wartości początkowe, nie trzeba określać rozmiaru tablicy. Kompilator wyznaczy go po policzeniu liczby wartości. Przykładowe definicje tablic zawiera tabela 1.

Tabela 1. Definiowanie tablic

`int numbers[10];`	Tablica dziesięciu liczb całkowitych
`const int SIZE = 10;` `int numbers[SIZE];`	Do określenia rozmiaru dobrze jest użyć stałej nazwanej.
`int size = 10;` `int numbers[size];`	**Uwaga**: W standardzie języka C++ rozmiar musi być stałą. Ta definicja tablicy nie zadziała w przypadku wszystkich kompilatorów.
`int squares[5] = { 0, 1, 4, 9, 16 };`	Tablica pięciu liczb całkowitych z wartościami początkowymi.
`int squares[] = { 0, 1, 4, 9, 16 };`	W razie podania wartości początkowych można nie podawać rozmiaru tablicy. Jest on ustalany na podstawie liczby wartości początkowych.
`int squares[5] = { 0, 1, 4 };`	W razie podania mniejszej liczby wartości początkowych, niż wynosi rozmiar tablicy, pozostałym elementom nadawana jest wartość 0. Tablica ta zawiera liczby 0, 1, 4, 0, 0.
`string names[3];`	Tablica trzech ciągów.

Składnia 6.1. Definiowanie tablicy



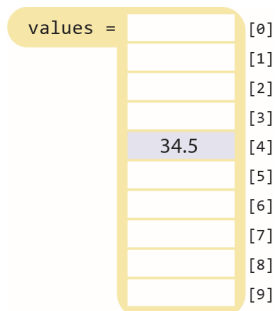
6.1.2. Dostęp do elementów tablicy

Dostęp do poszczególnych *wartości* tablicy jest możliwy przy użyciu notacji `values[i]`, w której indeks `i` jest liczbą całkowitą.

Wartości zapisane w tablicy są nazywane jej **elementami**. Każdy element ma numer pozycyjny, zwany **indeksem**. Aby uzyskać dostęp do elementu tablicy `values`, należy podać żądany indeks. Służy do tego operator []:

```
values[4] = 34.5;
```

Teraz do elementu o indeksie 4 wprowadzono wartość 34,5 (rysunek 2).



Rysunek 2. Nadawanie wartości elementowi tablicy

Do wyświetlenia treści elementu o indeksie 4 służy polecenie:

```
cout << values[4] << endl;
```

Element tablicy może być używany jak każda zmienna.

Jak widać, element `values[4]` może być używany jak każda inna zmienna typu `double`.

W języku C++ pozycja w tablicy jest liczona w sposób, który może wzbudzić zdziwienie. Jeśli przypatrzysz się dokładnie rysunkowi 2, odkryjesz, że przy zmianie `values[4]` wartość została wpisana do *piątego* elementu. W języku C++ numeracja w tablicach *zaczyna się od 0*. Oznacza to, że w tablicy `values` zawarte są elementy:

`values[0]` — pierwszy,
`values[1]` — drugi,
`values[2]` — trzeci,
`values[3]` — czwarty,
`values[4]` — piąty,
...
`values[9]` — dziesiąty.

Dlaczego w języku C++ wybrano taki schemat numeracji, dowiesz się w rozdziale 7.

> Wartość indeksu tablicy musi wynosić co najmniej zero, ale mniej niż jej rozmiar.

Należy uważać na wartości indeksów. Próba dostępu do nieistniejącego elementu jest poważnym błędem. Gdyby np. tablica `values` miała dwadzieścia elementów, nie wolno byłoby stosować zapisu `values[20]`.

> Przekroczenie indeksu, które występuje wtedy, gdy zostanie podany niewłaściwy indeks tablicy, może doprowadzić do uszkodzenia danych lub spowodować awarię programu.

Próba dostępu do elementu, którego indeks nie mieści się w dopuszczalnym zakresie, jest nazywana **przekroczeniem indeksu** (ang. *bounds error*). Kompilator nie wyłapuje błędów tego typu. Nawet uruchomiony program nie wygeneruje *żadnego komunikatu o błędzie*. Po przekroczeniu indeksu niepostrzeżenie odczytywane lub nadpisywane jest inne miejsce w pamięci. W rezultacie program może mieć losowe błędy, a nawet ulec awarii.

Przekroczenie indeksu wygląda najczęściej następująco:

```
double values[10];
cout << values[10];
```

W dziesięcioelementowej tablicy nie ma elementu `values[10]` — dozwolone wartości indeksu to zakres od 0 do 9.

Aby odwiedzić wszystkie elementy tablicy, do ich indeksowania należy użyć zmiennej. Załóżmy, że tablica `values` ma dziesięć elementów, a zmienna typu całkowitego `i` przyjmuje wartości 0, 1, 2 itd. aż do 9. Wówczas wyrażenie `values[i]` zwraca po kolei każdy element. Aby wyświetlić wszystkie, należy użyć np. takiej pętli:

```
for (int i = 0; i < 10; i++)
{
   cout << values[i] << endl;
}
```

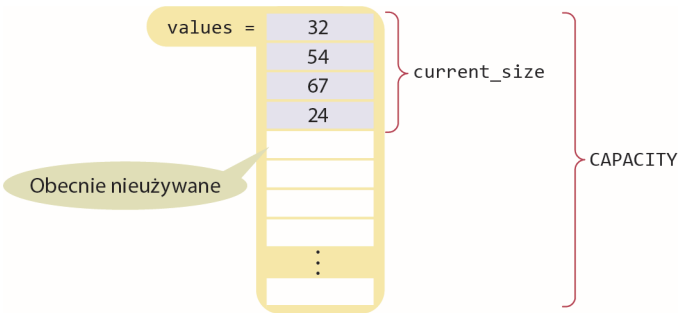
Warto zwrócić uwagę, że w warunku pętli indeks jest *mniejszy niż* 10, bo w tablicy nie ma elementu `values[10]`.

6.1.3. Częściowo wypełnione tablice

Po uruchomieniu programu nie można zmienić rozmiaru tablicy. To problem, jeśli nie wiadomo z góry, ilu potrzeba elementów. W takiej sytuacji należy właściwie dobrać liczbę elementów tablicy, czyli jej **pojemność** (ang. *capacity*). Możemy np. zadecydować, aby czasem przechowywać więcej niż dziesięć wartości, ale nigdy ponad 100:

```
const int CAPACITY = 100;
double values[CAPACITY];
```

Po uruchomieniu programu zwykle tylko część tablicy zawiera faktycznie elementy. Taką tablicę nazywamy *częściowo wypełnioną*. Należy utrzymywać zmienną towarzyszącą (ang. *companion variable*), zliczającą elementy będące naprawdę w użyciu. Na rysunku 3 nosi ona nazwę `current_size`.



Rysunek 3. Częściowo wypełniona tablica

W poniższej pętli pobierane są liczby i wypełniana jest nimi tablica `values`.

```
int current_size = 0;
double input;
while (cin >> input)
{
   if (current_size < CAPACITY)
   {
      values[current_size] = input;
      current_size++;
   }
}
```

W przypadku częściowo wypełnionej tablicy należy zapisywać jej aktualny rozmiar w zmiennej towarzyszącej.

Po zakończeniu pętli zmienna `current_size` zawiera faktyczną liczbę elementów tablicy. Należy pamiętać o tym, że jeśli tablica się zapełni, trzeba przestać przyjmować liczby.

Również podczas przetwarzania zebranych elementów nie należy kierować się pojemnością tablicy, ale korzystać ze zmiennej towarzyszącej. Częściowo wypełnioną tablicę wypisuje się w następującej pętli:

```
for (int i = 0; i < current_size; i++)
{
   cout << values[i] << endl;
}
```

Często popełniany błąd 6.1

Przekroczenie zakresu

Prawdopodobnie najczęstszym błędem związanym z użyciem tablic jest dostęp do nieistniejącego elementu.

```
double values[10];
values[10] = 5.4;
// Błąd — tablica values ma 10 elementów o indeksach od 0 do 9
```

Jeśli program uzyskuje dostęp do indeksu tablicy poza jej zakresem, najczęściej nie jest pokazywany komunikat o błędzie. Zamiast tego program po cichu (albo i nie) uszkadza pewne dane w pamięci. Poza najkrótszymi programami, w których problem ten może pozostać niezauważony, uszkodzenie danych powoduje nieprzewidziane działanie programu. W niektórych środowiskach do programowania w C++ podjęto wysiłek raportowania pewnych błędów, ale nie są one wykrywane we wszystkich przypadkach. Przekroczenie zakresu to poważny problem (patrz też sekcja „Informatyka a społeczeństwo 6.1"). Powinieneś wyrobić sobie nawyk sprawdzania wszystkich operacji dostępu do elementów tablic w swoich programach i zastanawiania się, czy są one poprawne.

Wskazówka dla programistów 6.1

Używaj tablic do przechowywania serii związanych ze sobą wartości

Tablice są przeznaczone do przechowywania serii wartości mających to samo znaczenie. Rozsądne jest np. przechowywanie w niej ocen z testów:

```
int scores[NUMBER_OF_SCORES];
```

Złym przykładem jest natomiast tablica

```
double personal_data[3];
```

w której na pozycjach 0, 1 i 2 zapisane są odpowiednio wiek pewnej osoby, saldo jej konta i rozmiar buta. Dla programisty uciążliwe będzie pamiętanie, gdzie w tablicy znajdują się wartości określonych danych. W tej sytuacji dużo lepiej użyć trzech oddzielnych zmiennych albo obiektu, o którym dowiesz się w rozdziale 9.

Informatyka a społeczeństwo 6.1 Wirusy komputerowe

W listopadzie 1988 r. Robert Morris, student Cornell University, uruchomił tzw. program wirusowy, który zainfekował znaczną część komputerów podłączonych do internetu (który wówczas był dużo mniejszy niż obecnie). Morris został skazany na karę pozbawienia wolności w zawieszeniu na trzy lata, 400 godzin prac społecznych i grzywnę w wysokości 10 000 dolarów.

Aby wirus mógł atakować komputery, jego autor musi znaleźć sposób na to, by zostały wykonane instrukcje wirusa. W tym konkretnym przypadku atak polegał na przepełnieniu bufora wskutek przesłania do programu uruchomionego na innym komputerze niezwykle dużych danych wejściowych. W programie tym alokowana była tablica o rozmiarze 512 znaków, ponieważ zakładano, że nikt nigdy nie poda takich długich danych wejściowych. Niestety program ten był napisany w języku C, w którym podobnie jak w C++ nie sprawdza się, czy indeks tablicy jest mniejszy

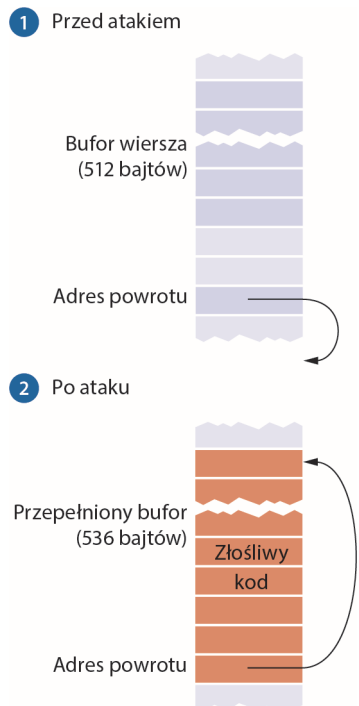
niż jej długość. Podczas zapisywania w tablicy wartości na pozycji o zbyt dużym indeksie nadpisywane jest po prostu miejsce w pamięci należące do innych obiektów. Od programistów języka C oczekuje się sprawdzania danych pod kątem bezpieczeństwa, ale w atakowanym programie tego zaniechano. Program wirusowy celowo wprowadzał do tablicy o wielkości 512 znaków 536 bajtów. Tymi nadmiarowymi 24 bajtami nadpisywano przechowywany zaraz po tablicy adres powrotu, o którym wiedział przeprowadzający atak. Gdy funkcja odczytująca dane wejściowe kończyła działanie, sterowanie nie było zwracane do miejsca wywołania, ale do przesłanego kodu wirusa (patrz rysunek). Kod ten był wykonywany na zdalnej maszynie i ją infekował.

W ostatnich latach ataki na komputery stały się bardziej intensywne, a motywy groźniejsze. Wirusy, zamiast uniemożliwiać pracę zaatakowanych komputerów, lokują się w nich na stałe. Przestępcy wynajmują moc obliczeniową przejętych komputerów w celu wysyłania spamu. Jeszcze inne wirusy monitorują każde naciśnięcie klawisza i wszystko to, co wygląda na numery kart kredytowych lub hasła do banku, wysyłają do swoich nadzorców, albo też szyfrują wszystkie pliki na dysku, a za ich odblokowanie domagają się od nieszczęśliwego użytkownika okupu pieniężnego.

Komputery są zwykle infekowane dlatego, że użytkownik uruchamia kod pobrany z internetu przez kliknięcie ikony lub odnośnika prowadzących rzekomo do interesującej gry lub klipu wideo. Programy antywirusowe sprawdzają wszystkie pobrane programy pod kątem stale rosnącej listy znanych wirusów.

Jeśli korzystasz z komputera w celu zarządzania finansami, musisz być świadomy ryzyka infekcji. Jeśli wirus odczyta Twoje hasło bankowe i opróżni Twoje konto, trudno będzie Ci przekonać instytucję finansową, że nie było to Twoje działanie, i najprawdopodobniej utracisz swoje pieniądze. Dobrze jest korzystać z usług banków stosujących przy większych transakcjach uwierzytelnianie dwupoziomowe, np. oddzwanianie na telefon komórkowy.

Wirusy komputerowe są używane nawet w celach militarnych. W 2010 r. wirus o nazwie Stuxnet rozprzestrzeniał się w systemach Microsoft Windows i infekował klucze USB. Kod wirusa wyszukiwał komputery przemysłowe firmy Siemens i nieznacznie je przeprogramowywał. Okazało się, że wirus został zaprojektowany w celu uszkodzenia wirówek używanych w przeprowadzanych w Iranie operacjach wzbogacania uranu. Komputery sterujące wirówkami nie były połączone z internetem, ale do ich konfiguracji używano kluczy USB, a niektóre z nich były zainfekowane. Specjaliści ds. zabezpieczeń utrzymują, że wirus został opracowany w amerykańskich i izraelskich agencjach wywiadowczych oraz że spowodował spowolnienie irańskiego programu nuklearnego. Żadne z tych państw nie potwierdziło oficjalnie swojego udziału w tych atakach ani też nie zaprzeczyło.



Atak „przepełnienia bufora”

6.2. Typowe algorytmy tablicowe

W następnych punkach omówimy niektóre z algorytmów stosowanych często do przetwarzania serii wartości. Zaprezentujemy te algorytmy po to, byś mógł ich używać w przypadku całkowicie lub częściowo wypełnionych tablic oraz wektorów (które przedstawimy w podrozdziale 6.7). Gdy zobaczysz odwołanie do *rozmiaru* wartości, zamień je na stałą lub zmienną zawierającą liczbę elementów tablicy (albo wyrażenie `values.size()`, jeśli zmienna `values` jest wektorem).

6.2.1. Wypełnianie wartościami

W tej pętli tablica jest zapełniana zerami:

```
for (int i = 0; i < rozmiar tablicy values; i++)
{
   values[i] = 0;
}
```

Teraz wprowadźmy do tablicy `squares` liczby 0, 1, 4, 9, 16 itd. Zauważmy, że element o indeksie 0 ma wartość 0^2, element o indeksie 1 — 1^2 itd.

```
for (int i = 0; i < rozmiar tablicy squares; i++)
{
   squares[i] = i * i;
}
```

6.2.2. Kopiowanie

Weźmy pod uwagę dwie tablice:

```
int squares[5] = { 0, 1, 4, 9, 16 };
int lucky_numbers[5];
```

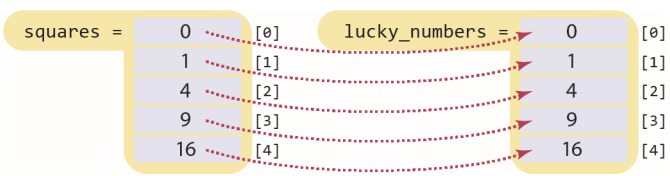
Aby skopiować tablicę, należy skopiować wszystkie jej elementy do nowej tablicy przy użyciu pętli.

Załóżmy, że chcemy skopiować wszystkie wartości z jednej tablicy do drugiej. Pokazane poniżej przypisanie jest błędne:

```
lucky_numbers = squares; // Błąd
```

W języku C++ nie można po prostu przypisać jednej tablicy do drugiej, ale jak pokazano na rysunku 4, trzeba skopiować wszystkie elementy przy użyciu pętli:

```
for (int i = 0; i < 5; i++)
{
   lucky_numbers[i] = squares[i];
}
```



Rysunek 4. Kopiowanie elementów w celu skopiowania tablicy

6.2.3. Suma i średnia

Z tym algorytmem spotkałeś się już w punkcie 4.7.1. Oto kod obliczający sumę wszystkich elementów tablicy:

```
double total = 0;
for (int i = 0; i < rozmiar tablicy values; i++)
{
total = total + values[i];
}
```

Aby otrzymać średnią, trzeba podzielić sumę przez liczbę elementów tablicy:

```
double average = total / rozmiar tablicy values;
```

Należy się upewnić, że rozmiar tablicy nie jest równy zero.

6.2.4. Maksimum i minimum

Należy skorzystać z algorytmu z punktu 4.7.5, w którym utrzymywana była zmienna do przechowywania największego napotkanego elementu. Oto jego implementacja dla tablic:

```
double largest = values[0];
for (int i = 1; i < rozmiar tablicy values; i++)
{
   if (values[i] > largest)
   {
      largest = values[i];
   }
}
```

Warto zauważyć, że pętlę rozpoczynamy od liczby 1, bo zmienną `largest` zainicjowaliśmy wartością `values[0]`.

Aby obliczyć najmniejszą wartość, wystarczy odwrócić porównanie.

Algorytmy te wymagają, by tablica zawierała co najmniej jeden element.

6.2.5. Separatory elementów

> Podczas rozdzielania elementów przed pierwszym nie należy wstawiać separatora.

Podczas wyświetlania elementów kolekcji zwykle trzeba je rozdzielić przecinkami lub pionowymi kreskami, np. tak:

```
1 | 4 | 9 | 16 | 25
```

Należy pamiętać o tym, że separatorów powinno być o jeden mniej niż liczb. Separator wypisuje się przed każdym elementem *z wyjątkiem pierwszego* (o indeksie 0):

```
for (int i = 0; i < rozmiar tablicy values; i++)
{
   if (i > 0)
   {
      cout << " | ";
```

```
   }
   cout << values[i];
}
```

6.2.6. Zliczanie pasujących elementów

Załóżmy, że poproszono Cię o policzenie, ile elementów tablicy spełnia określone kryterium. Odwiedź każdy z elementów, sprawdź, czy pasuje, a jeśli tak, zwiększ licznik. W pokazanej poniżej pętli zliczane są elementy o wartości co najmniej 100:

```
int count = 0;
for (int i = 0; i < rozmiar tablicy values; i++)
{
   if (values[i] >= 100)
   {
      count++;
   }
}
```

6.2.7. Wyszukiwanie liniowe

Podczas wyszukiwania liniowego elementy są sprawdzane po kolei aż do znalezienia dopasowania.

Często zachodzi potrzeba wyszukania pozycji elementu po to, by go zastąpić lub usunąć. Należy odwiedzić wszystkie elementy do chwili znalezienia pasującego lub dojścia do końca tablicy. Tak szukamy pozycji pierwszego elementu o wartości 100:

```
int pos = 0;
bool found = false;
while (pos < rozmiar tablicy values && !found)
{
   if (values[pos] == 100)
   {
      found = true;
   }
   else
   {
      pos++;
   }
}
```

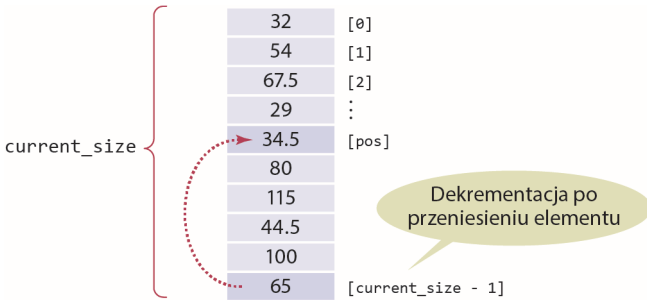
Jeśli zmienna `found` ma wartość `true`, to `pos` zawiera pozycję pierwszego trafienia.

W celu wyszukania konkretnego elementu trzeba odwiedzić wszystkie elementy i zatrzymać się po znalezieniu pasującego

6.2.8. Usuwanie elementu

Rozważmy częściowo wypełnioną tablicę `values`, której aktualny rozmiar jest zapisany w zmiennej `current_size`. Załóżmy, że usuwamy z tablicy element o indeksie `pos`. Jeśli elementy nie są w żaden sposób uporządkowane, zadanie to jest proste. Wystarczy zastąpić usuwany element *ostatnim*, a potem zmniejszyć o jeden zmienną służącą do śledzenia rozmiaru tablicy (rysunek 5).

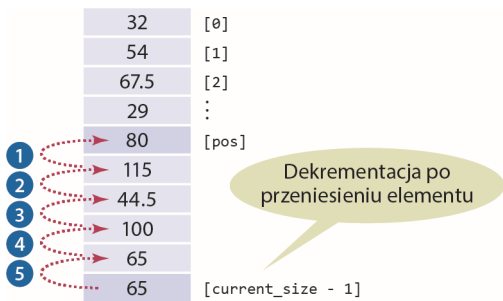
```
values[pos] = values[current_size - 1];
current_size--;
```



Rysunek 5. Usuwanie elementu z tablicy nieuporządkowanej

Jeśli kolejność elementów ma znaczenie, sytuacja jest bardziej złożona. Wszystkie elementy następujące po usuwanym trzeba przenieść na pozycje o niższych indeksach, a potem zmniejszyć o jeden zmienną przechowującą rozmiar tablicy (rysunek 6).

```
for (int i = pos + 1; i < current_size; i++)
{
   values[i - 1] = values[i];
}
current_size--;
```

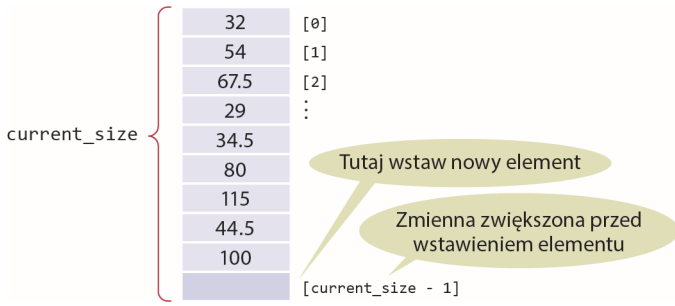


Rysunek 6. Usuwanie elementu z tablicy uporządkowanej

6.2.9. Wstawianie elementu

Jeśli kolejność elementów nie ma znaczenia, to nowy można po prostu wstawić na końcu tablicy i zwiększyć o jeden zmienną śledzącą jej rozmiar (rysunek 7). W przypadku częściowo wypełnionej tablicy wygląda to tak:

```
if (current_size < CAPACITY)
{
   current_size++;
   values[current_size - 1] = new_element;
}
```



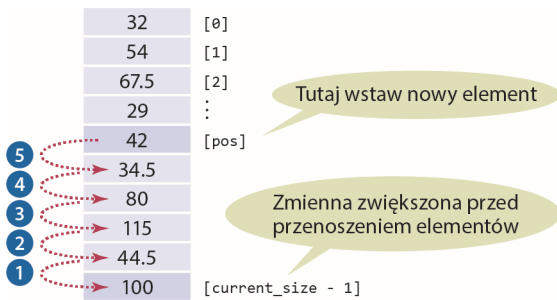
Rysunek 7. Wstawianie elementu do tablicy nieuporządkowanej

Przed wstawieniem elementu do tablicy należy przenieść elementy na jej koniec, *począwszy od ostatniego*.

Więcej pracy wymaga wstawienie elementu na konkretnej pozycji w środku sekwencji. Najpierw należy zwiększyć o jeden zmienną zawierającą aktualny rozmiar, następnie przenieść wszystkie elementy powyżej miejsca wstawiania na pozycje o wyższych indeksach, a na koniec wstawić nowy element. Oto kod dla częściowo wypełnionej tablicy:

```
if (current_size < CAPACITY)
{
   current_size++;
   for (int i = current_size - 1; i > pos; i--)
   {
      values[i] = values[i - 1];
   }
   values[pos] = new_element;
}
```

Należy zwrócić uwagę na kolejność przenoszenia elementów: przy usuwaniu najpierw przenosi się na pozycję o niższym indeksie element następny po usuwanym, potem kolejny i tak dalej, aż do końca tablicy. Przy wstawianiu należy zacząć od elementu z końca tablicy, przenieść go na pozycję o wyższym indeksie, potem przenieść wcześniejszy i tak dalej, aż do chwili dojścia do miejsca wstawiania (rysunek 8).



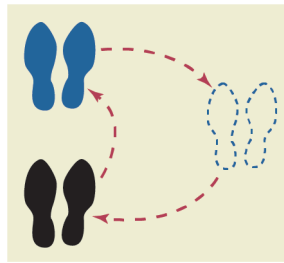
Rysunek 8. Wstawianie elementu do tablicy uporządkowanej

6.2.10. Przestawianie elementów

Często elementy tablicy trzeba przestawić. Wielokrotnego powtórzenia tej operacji wymaga np. algorytm sortowania opisany w sekcji „Temat specjalny 6.2”.

Zastanówmy się nad przestawieniem elementów znajdujących się na pozycjach `i` i `j` tablicy `values`. Elementowi `values[i]` chcemy nadać wartość `values[j]`. Wtedy jednak wartość przechowywana obecnie na pozycji `values[i]` zostanie nadpisana, więc musimy ją najpierw zapamiętać:

```
double temp = values[i];
values[i] = values[j];
```



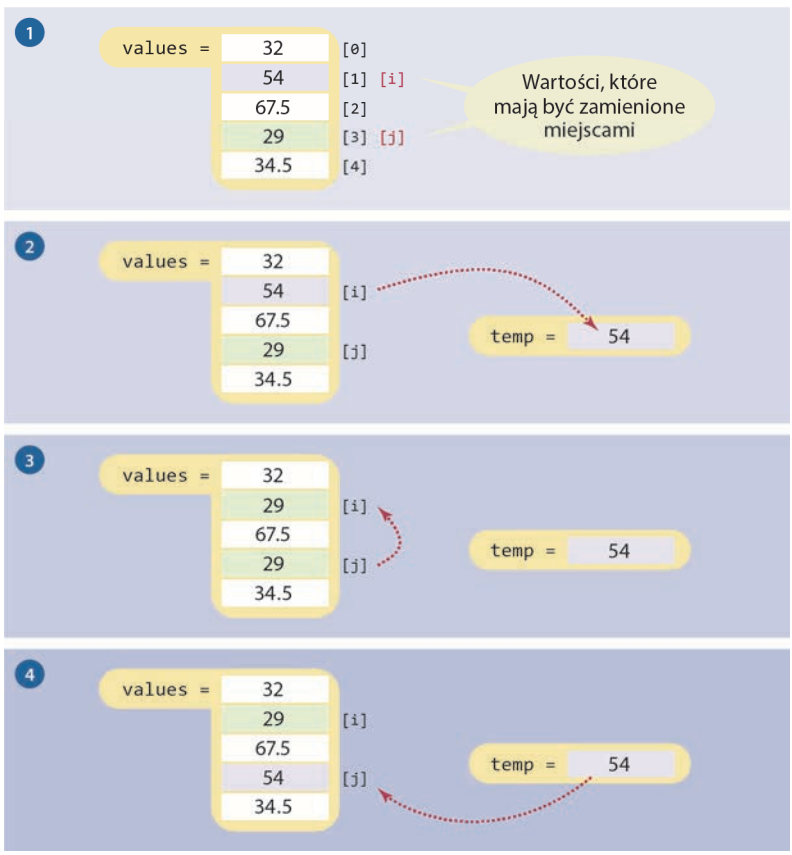
Aby przestawić dwa elementy, potrzebna jest zmienna tymczasowa

Przy przestawianiu dwóch elementów należy użyć zmiennej tymczasowej.

Teraz możemy nadać elementowi `values[j]` zapamiętaną wartość.

```
values[j] = temp;
```

Proces ten pokazano na rysunku 9.



Rysunek 9. Przestawianie elementów tablicy

6.2.11. Odczyt danych wejściowych

Jeśli wiadomo, ile danych wejściowych poda użytkownik, ich umieszczenie w tablicy jest łatwe:

```
double values[NUMBER_OF_INPUTS];
for (i = 0; i < NUMBER_OF_INPUTS; i++)
{
   cin >> values[i];
}
```

Sposób ten jednak nie zadziała, jeśli liczba danych wejściowych nie jest ustalona. W tym przypadku wartości są dodawane do tablicy aż do chwili, gdy nastąpi koniec danych wejściowych.

```
double values[CAPACITY];
int current_size = 0;
double input;
while (cin >> input)
{
   if (current_size < CAPACITY)
   {
      values[current_size] = input;
      current_size++;
   }
}
```

W rezultacie tablica `values` zostaje częściowo zapełniona, a w zmiennej towarzyszącej `current_size` zapisana jest liczba danych wejściowych.

W przypadku tej pętli wszystkie dane wejściowe, które nie zmieszczą się w tablicy, zostaną odrzucone. Lepszym podejściem w razie wyczerpania pojemności tablicy `values` byłoby skopiowanie jej do nowej, większej tablicy (podrozdział 7.4).

Przedstawiony poniżej program jest rozwiązaniem zadania, które postawiliśmy sobie na początku tego rozdziału, czyli oznaczenia największej wartości w serii danych wejściowych:

podr02/largest.cpp

```
#include <iostream>

using namespace std;

int main()
{
   const int CAPACITY = 1000;
   double values[CAPACITY];
   int current_size = 0;

   cout << "Proszę wprowadzić liczby, W - wyjście:" << endl;
   double input;
   while (cin >> input)
   {
      if (current_size < CAPACITY)
      {
         values[current_size] = input;
         current_size++;
      }
   }

   double largest = values[0];
```

```
23    for (int i = 1; i < current_size; i++)
24    {
25       if (values[i] > largest)
26       {
27          largest = values[i];
28       }
29    }
30
31    for (int i = 0; i < current_size; i++)
32    {
33       cout << values[i];
34       if (values[i] == largest)
35       {
36          cout << " <== największa liczba";
37       }
38       cout << endl;
39    }
40
41    return 0;
42 }
```

Przebieg działania programu

```
Proszę wprowadzić liczby, W - wyjście: 34.5 80 115 44.5 W
34.5
80
115 <== największa liczba
44.5
```

Temat specjalny 6.1
Sortowanie za pomocą biblioteki C++

Elementy tablicy lub wektora często trzeba posortować. W sekcji „Temat specjalny 6.2” zostanie pokazany algorytm sortowania, który jest względnie prosty, ale niezbyt wydajny. Efektywne algorytmy są znacznie bardziej złożone. Na szczęście w bibliotece C++ dostępna jest wydajna funkcja `sort`. Aby posortować tablicę a, której liczba elementów wynosi `size`, należy wywołać funkcję

```
sort(a, a + size);
```

Aby posortować wektor `values`, należy dokonać wywołania

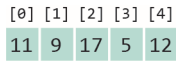
```
sort(values.begin(), values.end());
```

Aby w pełni zrozumieć, dlaczego w ten sposób wywołuje się funkcję `sort`, musiałbyś poznać zaawansowane zagadnienia języka C++, które wykraczają poza zakres tej książki. Nie wahaj się jednak wywoływać funkcji `sort`, gdy tylko będziesz potrzebować posortować tablicę lub wektor.

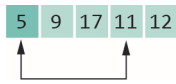
Aby używać funkcji `sort`, należy dołączyć do programu nagłówek `<algorithm>`.

Temat specjalny 6.2
Algorytm sortowania

Algorytm sortowania zmienia rozmieszczenie elementów w sekwencji tak, by były przechowywane w postaci uporządkowanej. Oto prosty algorytm zwany **sortowaniem przez wybieranie** (ang. *selection sort*). Pomyślmy, jak posortować pokazaną poniżej tablicę `values`:



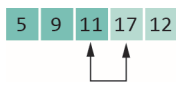
Oczywistym pierwszym krokiem będzie znalezienie najmniejszego elementu. W tym przypadku najmniejszym elementem jest liczba 5, przechowywana na pozycji `values[3]`. Liczbę 5 należy przenieść na początek tablicy. Oczywiście na pozycji `values[0]` jest już zapisany element, a mianowicie liczba 11. Nie można zatem po prostu przenieść zawartości elementu `values[3]` do `values[0]` bez przeniesienia liczby 11 w inne miejsce. Nie wiadomo jeszcze, gdzie ostatecznie powinna trafić liczba 11, ale z pewnością nie powinna być na pozycji `values[0]`. Po prostu usuń ją z drogi przez *zamianę miejscami* z elementem `values[3]`:



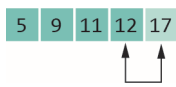
Pierwszy element jest już na właściwym miejscu. Na powyższym rysunku ciemniejszym kolorem wskazano fragment tablicy, który jest już posortowany.

Teraz znajdź najmniejszą z pozostałych pozycji `values[1]...values[4]`. Ta najmniejsza liczba, 9, jest już na właściwym miejscu. W związku z tym nie musisz nic robić, po prostu rozszerz posortowany obszar o jedną pozycję w lewo:

Powtórz ten proces. Minimalną wartością w nieposortowanym regionie jest liczba 11, którą trzeba zamienić miejscami z pierwszą wartością w tym regionie, czyli liczbą 17:



Nieposortowany region obejmuje już tylko dwa elementy; kontynuuj tę samą pomyślną strategię. Najmniejszym elementem jest liczba 12. Zamień ją miejscami z pierwszą wartością, 17:



Pozostał nieprzetworzony region o długości 1, ale taki region oczywiście zawsze jest posortowany. To wszystko.

Oto kod w języku C++:

```
for (int unsorted = 0; unsorted < size - 1; unsorted++)
{
   // Znajdź pozycję najmniejszej liczby.
   int min_pos = unsorted;
   for (int i = unsorted + 1; i < size; i++)
   {
      if (values[i] < values[min_pos]) { min_pos = i; }
   }
   // Zmień miejsce najmniejszej liczby przez wstawienie jej do obszaru posortowanego.
   if (min_pos != unsorted)
   {
      double temp = values[min_pos];
      values[min_pos] = values[unsorted];
      values[unsorted] = temp;
   }
}
```

Ten algorytm łatwo zrozumieć, ale nie jest zbyt wydajny. Informatycy drobiazgowo przestudiowali ten temat i wynaleźli znacznie lepsze algorytmy. Jeden z nich jest dostępny w funkcji `sort` ze standardowej biblioteki C++ — patrz „Temat specjalny 6.1”.

Temat specjalny 6.3
Wyszukiwanie binarne

Jeśli tablica jest posortowana, do jej przeszukania można użyć dużo szybszego algorytmu niż opisane w punkcie 6.2.7 wyszukiwanie liniowe.

Rozważmy przedstawioną poniżej uporządkowaną tablicę `values`:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	5	8	9	12	17	20	32

Chcielibyśmy zobaczyć, czy występuje w niej liczba 15. Zawęźmy pole naszych wyszukiwań przez sprawdzenie, czy liczba ta znajduje się w pierwszej, czy w drugiej połowie tablicy. Na ostatnim miejscu w pierwszej połowie zbioru danych, `values[3]`, znajduje się liczba 9, mniejsza niż ta, której szukamy. Pasującego elementu powinniśmy więc szukać w drugiej połowie tablicy, czyli w sekwencji:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	5	8	9	12	17	20	32

Teraz ostatnią liczbą w pierwszej połowie tej sekwencji jest 17, zatem szukana wartość musi znajdować się w sekwencji:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	5	8	9	12	17	20	32

Ostatnią liczbą w pierwszej połowie tej bardzo krótkiej sekwencji jest 12. To mniej niż poszukiwana przez nas wartość, musimy więc szukać w drugiej połowie:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	5	8	9	12	17	20	32

Wciąż nie mamy trafienia, bo 15 ≠ 17, a tej podsekwencji nie można już podzielić. Gdybyśmy chcieli wstawić do tej serii liczbę 15, musiałaby znaleźć się tuż przed elementem `values[5]`.

Ten proces nosi nazwę **wyszukiwania binarnego**, ponieważ w każdym kroku tniemy rozmiar wyszukiwania na pół. Możemy tak zrobić tylko dlatego, że wiemy, iż sekwencja liczb jest uporządkowana. Oto implementacja w języku C++:

```
bool found = false;
int low = 0;
int high = size - 1;
int pos = 0;
while (low <= high && !found)
{
   pos = (low + high) / 2; // Punkt środkowy podsekwencji
   if (values[pos] == searched_value) { found = true; }
   else if (values[pos] < searched_value) { low = pos + 1; } // Szukaj w drugiej połowie
   else { high = pos - 1; } // Szukaj w pierwszej połowie
}
if (found) { cout << "Znaleziono na pozycji " << pos; }
else { cout << "Nie znaleziono. Wstaw przed pozycją " << pos; }
```

6.3. Tablice a funkcje

W tym podrozdziale zbadamy, jak pisać funkcje przetwarzające tablice.

> Przy przesyłaniu do funkcji tablicy należy również przekazać jej rozmiar.

Funkcja, która przetwarza wartości tablicy, musi mieć informację na temat właściwej liczby jej elementów. Oto na przykład funkcja `sum`, obliczająca sumę wszystkich elementów tablicy:

```
double sum(double values[], int size)
{
   double total = 0;
   for (int i = 0; i < size; i++)
   {
      total = total + values[i];
   }
   return total;
}
```

Warto zwrócić uwagę na specjalną składnię tablicowych zmiennych parametrycznych. Przy zapisie takiego parametru należy po jego nazwie umieścić pustą parę znaków `[]`. W nawiasach kwadratowych nie podaje się rozmiaru tablicy.

Przy wywoływaniu funkcji należy podać zarówno nazwę tablicy, jak i jej rozmiar, np.

```
const int NUMBER_OF_SCORES = 10;
double scores[NUMBER_OF_SCORES]
   = { 32, 54, 67.5, 29, 34.5, 80, 115, 44.5, 100, 65 };
double total_score = sum(scores, NUMBER_OF_SCORES);
```

Do funkcji można też przekazać mniejszy rozmiar:

```
double partial_score = sum(scores, 5);
```

W trakcie tego wywołania zostanie policzona suma pierwszych pięciu elementów tablicy `scores`. Należy pamiętać o tym, że funkcja sama z siebie nie ma żadnych informacji na temat ilości elementów w tablicy. Korzysta po prostu z rozmiaru dostarczonego przez wywołującego.

> Parametry tablicowe są zawsze typu referencyjnego.

Parametry tablicowe są *zawsze typu referencyjnego*. (Dlaczego tak jest, dowiesz się z rozdziału 7.). Funkcje mogą modyfikować argumenty tablicowe, a zmiany te mają wpływ na tablicę przekazaną do funkcji. Przedstawiona poniżej funkcja `multiply` aktualizuje wszystkie elementy tablicy:

```
void multiply(double values[], int size, double factor)
{
   for (int i = 0; i < size; i++)
   {
      values[i] = values[i] * factor;
   }
}
```

W tym przypadku *nie* używa się symbolu `&`, który oznacza parametr referencyjny.

Funkcja nie może zwracać typu tablicowego.

Chociaż tablice mogą być argumentami funkcji, to funkcje nie mogą zwracać typu tablicowego. Jeśli w funkcji obliczanych jest wiele wartości, element wywołujący musi do przechowywania wyników dostarczyć tablicową zmienną parametryczną.

```
void squares(int n, int result[])
{
   for (int i = 0; i < n; i++)
   {
      result[i] = i * i;
   }
}
```

Jeśli funkcja modyfikuje rozmiar tablicy, musi poinformować o tym wywołującego.

Jeśli funkcja zmienia rozmiar tablicy, należy wskazać wywołującemu, ile elementów ma tablica po wywołaniu. Najprostszym na to sposobem jest zwrócenie nowego rozmiaru. Oto przykład — funkcja, która dodaje do tablicy dane wejściowe:

```
int read_inputs(double inputs[], int capacity)
{
   int current_size = 0;
   double input;
   while (cin >> input)
   {
      if (current_size < capacity)
      {
         inputs[current_size] = input;
         current_size++;
      }
   }
   return current_size;
}
```

Funkcja dodająca elementy do tablicy musi mieć informację o jej pojemności.

Warto zwrócić uwagę, że funkcja ta musi też posiadać informacje o pojemności tablicy. Każda funkcja, która dodaje do tablicy elementy, musi mieć informacje o jej pojemności. Funkcję tę wywołuje się następująco:

```
const int MAXIMUM_NUMBER_OF_VALUES = 1000;
double values[MAXIMUM_NUMBER_OF_VALUES];
int current_size = read_inputs(values, MAXIMUM_NUMBER_OF_VALUES);
// Tablica values jest częściowo wypełniona; jej rozmiar określa zmienna current_size.
```

Rozmiar można też podać w parametrze referencyjnym. Jest to bardziej stosowne w przypadku funkcji modyfikujących istniejącą tablicę:

```
void append_inputs(double inputs[], int capacity, int& current_size)
{
   double input;
   while (cin >> input)
   {
      if (current_size < capacity)
```

```
      {
         inputs[current_size] = input;
         current_size++;
      }
   }
}
```

Funkcję tę wywołuje się następująco:

```
append_inputs(values, MAXIMUM_NUMBER_OF_VALUES, current_size);
```

Po wywołaniu zmienna `current_size` zawiera nowy rozmiar.

Pokazany poniżej przykładowy program odczytuje liczby ze standardowego wejścia, podwaja je i wyświetla wyniki. W programie używane są trzy funkcje:

- Funkcja `read_inputs` wprowadza do tablicy wartości podane przez użytkownika. Zwraca liczbę wczytanych elementów.
- Funkcja `multiply` modyfikuje zawartość otrzymanej tablicy i stanowi demonstrację tego, że tablice są przekazywane przez referencję.
- Funkcja `print` nie modyfikuje zawartości otrzymanej tablicy.

podr03/functions.cpp

```
#include <iostream>

using namespace std;

/**
   Odczytuje sekwencję liczb zmiennoprzecinkowych.
   @param inputs tablica zawierająca liczby
   @param capacity pojemność tej tablicy
   @return liczba danych wejściowych przechowywanych w tablicy
*/
int read_inputs(double inputs[], int capacity)
{
   int current_size = 0;
   cout << "Proszę wprowadzić liczby, W - wyjście:" << endl;
   bool more = true;
   while (more)
   {
      double input;
      cin >> input;
      if (cin.fail())
      {
         more = false;
      }
      else if (current_size < capacity)
      {
         inputs[current_size] = input;
         current_size++;
      }
   }
   return current_size;
}

/**
```

```
34    Mnoży wszystkie elementy tablicy przez dany czynnik.
35    @param values częściowo wypełniona tablica
36    @param size liczba elementów tablicy values
37    @param factor liczba, przez którą mnożony jest każdy z elementów
38 */
39 void multiply(double values[], int size, double factor)
40 {
41    for (int i = 0; i < size; i++)
42    {
43       values[i] = values[i] * factor;
44    }
45 }
46
47 /**
48    Wyświetla elementy tablicy rozdzielone przecinkami.
49    @param values częściowo wypełniona tablica
50    @param size liczba elementów tablicy values
51 */
52 void print(double values[], int size)
53 {
54    for (int i = 0; i < size; i++)
55    {
56       if (i > 0) { cout << ", "; }
57       cout << values[i];
58    }
59    cout << endl;
60 }
61
62 int main()
63 {
64    const int CAPACITY = 1000;
65    double values[CAPACITY];
66    int size = read_inputs(values, CAPACITY);
67    multiply(values, size, 2);
68    print(values, size);
69
70    return 0;
71 }
```

Przebieg działania programu

```
Proszę wprowadzić liczby, W - wyjście: 12 25 20 W
24, 50, 40
```

Temat specjalny 6.4
Stałe parametry tablicowe

Jeśli funkcja nie modyfikuje parametru tablicowego, w dobrym stylu będzie dodanie do niego słowa zastrzeżonego `const`, tak jak tutaj:

```
double sum(const double values[], int size)
```

Słowo zastrzeżone `const` pomaga w czytaniu kodu, bo informuje, że funkcja nie zmienia wartości elementów tablicy. Jeśli w implementacji funkcji nastąpi próba modyfikacji tablicy, kompilator zgłosi ostrzeżenie.

6.4. Rozwiązywanie problemów: dostosowywanie algorytmów

Złożone zadania programistyczne można rozwiązywać dzięki łączeniu podstawowych algorytmów.

W podrozdziale 6.2 przedstawiliśmy kilka podstawowych algorytmów tablicowych. Algorytmy te stanowią elementy konstrukcyjne wielu programów przetwarzających tablice. Dobrą strategią rozwiązywania problemów jest zwykle posiadanie w swoim repertuarze podstawowych algorytmów, a potem ich łączenie i adaptowanie.

Zastanów się nad następującym przykładowym problemem: dostałeś wyniki klasówek uczniów. Masz policzyć ostateczny wynik z klasówek, będący sumą wszystkich wyników z pominięciem najmniejszego z nich. Jeśli wyniki wynoszą np.

```
8 7 8.5 9.5 7 4 10
```

to ostatecznym wynikiem będzie liczba 50.

Nie mamy gotowego algorytmu do zastosowania w tej sytuacji. Należy jednak zastanowić się, jakie algorytmy mogą tu wchodzić w grę. Należą do nich:

- obliczanie sumy (punkt 6.2.3),
- znajdowanie najmniejszej wartości (punkt 6.2.4),
- usuwanie elementu (punkt 6.2.8).

Możemy teraz sformułować plan działania będący połączeniem tych algorytmów.

Odszukaj najmniejszą liczbę.
Usuń ją z tablicy.
Oblicz sumę.

Spróbujmy go zastosować w naszym przypadku. Najmniejszą liczbą spośród

[0]	[1]	[2]	[3]	[4]	[5]	[6]
8	7	8.5	9.5	7	4	10

jest 4. Jak ją usunąć?

Mamy problem. W algorytmie z punktu 6.2.8 element do usunięcia jest lokalizowany na podstawie *pozycji* elementu, a nie jego wartości.

Mamy jednak do tego inny algorytm:

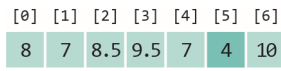
- wyszukiwanie liniowe (punkt 6.2.7).

Musimy poprawić nasz plan działania:

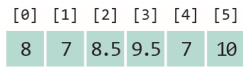
Odszukaj najmniejszą liczbę.
Znajdź jej pozycję.
Usuń element znajdujący się na tej pozycji z tablicy.
Oblicz sumę.

Czy to się sprawdzi? Kontynuujmy pracę nad przykładem.

Znaleźliśmy najmniejszą liczbę, wynoszącą 4. Dzięki wyszukiwaniu liniowemu ustaliliśmy, że wartość 4 występuje na pozycji 5.



Usuwamy ją:



Na koniec obliczamy sumę: 8 + 7 + 8,5 + 9,5 + 7 + 10 = 50.

Zademonstrowaliśmy tym samym, że strategia działa.

Czy można to zrobić lepiej? Po znalezieniu najmniejszej liczby ponowne przechodzenie przez tablicę w celu ustalenia jej pozycji wydaje się nieco nieefektywne.

> Należy poznać implementację podstawowych algorytmów po to, by móc je dostosowywać do potrzeb.

Algorytm znajdowania najmniejszej liczby możemy zmodyfikować tak, by zwracał jej pozycję. Oto wersja źródłowa algorytmu:

```
double smallest = values[0];
for (int i = 1; i <rozmiar tablicy values; i++)
{
   if (values[i] < smallest)
   {
      smallest = values[i];
   }
}
```

Gdy znajdziemy najmniejszą liczbę, będziemy też chcieli odczytać jej pozycję:

```
if (values[i] < smallest)
{
   smallest = values[i];
   smallest_position = i;
}
```

W zasadzie nie ma już potrzeby ustalania najmniejszej liczby. To po prostu element `values[small`
`↪est_position]`. Dzięki temu spostrzeżeniu możemy zmodyfikować algorytm następująco:

```
int smallest_position = 0;
for (int i = 1; i < rozmiar tablicy values; i++)
{
   if (values[i] < values[smallest_position])
   {
      smallest_position = i;
   }
}
```

Dzięki tej adaptacji rozwiązaliśmy problem za pomocą następującej strategii:

Znajdź pozycję najmniejszej liczby.
Usuń z tablicy element znajdujący się na tej pozycji.
Oblicz sumę.

Przykładowy kod

Program implementujący tę strategię znajduje się w podkatalogu *podr04* kodów źródłowych do tego rozdziału. Alternatywne podejście niewymagające zmian w tablicy zostanie pokazane w sekcji „Jak to zrobić 6.1".

W następnej sekcji zostanie pokazana technika wynajdowania nowego algorytmu w sytuacji, gdy do zadania nie można przystosować żadnego z podstawowych.

Jak to zrobić 6.1
Praca z tablicami

W wielu sytuacjach związanych z obróbką danych konieczne jest przetworzenie sekwencji wartości. W tym przewodniku „Jak to zrobić" przedstawione zostaną kroki niezbędne do zapisania danych wejściowych w tablicy i przeprowadzania obliczeń na jej elementach.

Definicja problemu. Rozważmy jeszcze raz problem z podrozdziału 6.4: końcowy wynik z klasówek jest obliczany w drodze zsumowania wszystkich wyników z wyjątkiem najniższego. Jeśli wyniki wynoszą np.

```
8 7 8.5 9.5 7 5 10
```

to ostatecznym wynikiem będzie 50.

Krok 1. Rozłóż zadanie na poszczególne kroki.

Zadanie należy zwykle rozbić na wiele kroków, np.

- wczytanie danych do tablicy,
- przetworzenie danych w jednym lub kilku krokach,
- wyświetlenie wyników.

Do podjęcia decyzji o tym, jak przetworzyć dane, potrzebna jest znajomość algorytmów tablicowych z podrozdziału 6.2. Większość zadań związanych z obróbką danych można rozwiązać za pomocą jednego z tych algorytmów lub ich połączenia.

W tym przykładowym problemie będziemy odczytywać dane. Potem usuniemy najmniejszą liczbę i policzymy sumę. Jeśli danymi wejściowymi będą np. `8 7 8.5 9.5 7 5 10`, usuniemy najmniejszą liczbę, czyli 5, i pozostanie `8 7 8.5 9.5 7 10`. Suma tych liczb da ostateczny wynik 50.

W tej sposób określiliśmy trzy kroki:

Odczytaj dane wejściowe.
Usuń najmniejszą liczbę.
Oblicz sumę.

Krok 2. Ustal, którego algorytmu (lub których algorytmów) potrzebujesz.

Czasem pojedynczy krok idealnie odpowiada jednemu z podstawowych algorytmów z podrozdziału 6.2. Tak będzie w przypadku obliczania sumy (punkt 6.2.3) i odczytu danych wejściowych (punkt 6.2.11). Kiedy indziej trzeba połączyć ze sobą kilka algorytmów. Aby usunąć najmniejszą liczbę, można np. najpierw ją znaleźć (punkt 6.2.4), potem ustalić jej pozycję (punkt 6.2.7), a następnie usunąć z tej pozycji element (punkt 6.2.8).

Nasz udoskonalony plan wygląda następująco:

Odczytaj dane wejściowe.
Odszukaj najmniejszą liczbę.
Znajdź jej pozycję.
Usuń element z tej pozycji.
Oblicz sumę.

Plan ten będzie skuteczny — patrz podrozdział 6.4. Oto jednak inne podejście. Można z łatwością obliczyć sumę i odjąć od niej najmniejszą liczbę. Wtedy nie musielibyśmy szukać jej pozycji. Poprawiony plan wygląda tak:

Odczytaj dane wejściowe.
Odszukaj najmniejszą liczbę.
Oblicz sumę.
Odejmij od sumy najmniejszą liczbę.

Krok 3. Użyj funkcji, by nadać strukturę programowi.

Chociaż zapewne można by umieścić wszystkie te kroki w funkcji `main`, rzadko kiedy jest to właściwe. Lepiej przekształcić każdy krok przetwarzania w osobną funkcję. W tym przypadku zaimplementujemy trzy funkcje:

- `read_inputs`,
- `sum`,
- `minimum`.

Funkcja `main` będzie po prostu zawierać ich wywołania:

```
const int CAPACITY = 1000;
double scores[CAPACITY];
int scores_size = read_inputs(scores, CAPACITY);
double total = sum(scores, scores_size) - minimum(scores, scores_size);
```

Do każdej funkcji przetwarzającej tablicę trzeba przekazać samą tę tablicę i jej rozmiar, np. tak:

```
double sum(double values[], int size)
```

Jeśli funkcja zmienia rozmiar tablicy, musi o tym poinformować wywołującego. Nowy rozmiar zwraca np. funkcja odczytująca dane wejściowe:

```
int read_inputs(double inputs[], int capacity) // Zwraca rozmiar
```

Krok 4. Poskładaj i przetestuj program.

Połącz funkcje w jeden program. Przejrzyj kod i sprawdź, czy zawiera obsługę zarówno sytuacji zwyczajnych, jak i wyjątkowych. Co się stanie z pustą tablicą? Albo taką, która zawiera jeden element? Co będzie, jeśli nic nie będzie pasowało? Albo trafień będzie kilka? Zastanów się nad tymi warunkami brzegowymi i upewnij się, że program działa poprawnie.

W naszym przykładzie nie można wyznaczyć najmniejszej wartości, jeśli tablica będzie pusta. W tym przypadku należy przerwać program i wyświetlić komunikat o błędzie, *zanim* nastąpi próba wywołania funkcji `minimum`.

Co zrobić, jeśli najmniejsza wartość wystąpi więcej niż raz? Oznacza to, że uczeń uzyskał ten sam niski wynik z więcej niż jednego testu. Odejmiemy tylko jedno w wystąpień tego niskiego wyniku i to jest pożądane działanie.

W poniższej tabeli pokazano przypadki testowe wraz z oczekiwanymi wynikami:

Przypadek testowy	Spodziewany wynik	Komentarz
8 7 8.5 9.5 7 5 10	50	Patrz krok 1.
8 7 7 9	24	Należy usunąć tylko jedno wystąpienie niskiego wyniku.
8	0	Po usunięciu najniższego wyniku nie pozostaje nic.
(brak danych)	**Błąd**	Takie dane wejściowe nie są dopuszczalne.

Taka funkcja `main` dopełnia rozwiązanie. Oto cały program:

jak_to_zrobic_1/scores.cpp

```
#include <iostream>

using namespace std;

/**
   Odczytuje sekwencję liczb zmiennoprzecinkowych.
   @param inputs tablica zawierająca liczby
   @param capacity pojemność tej tablicy
   @return liczba danych wejściowych przechowywanych w tablicy
*/
int read_inputs(double inputs[], int capacity)
{
   int current_size = 0;
   cout << "Proszę wprowadzić liczby, W - wyjście:" << endl;
   bool done = false;
   while (!done)
   {
      double input;
      cin >> input;
      if (cin.fail())
      {
         done = true;
      }
      else if (current_size < capacity)
      {
         inputs[current_size] = input;
         current_size++;
      }
   }
   return current_size;
}

/**
   Pobiera z tablicy najmniejszą liczbę.
   @param values częściowo wypełniona tablica o rozmiarze >= 1
   @param liczba elementów tablicy values
   @return najmniejsza liczba w tablicy values
*/
double minimum(double values[], int size)
{
```

```
41    double smallest = values[0];
42    for (int i = 1; i < size; i++)
43    {
44       if (values[i] < smallest)
45       {
46          smallest = values;
47       }
48    }
49    return smallest;
50 }
51
52 /**
53    Oblicza sumę elementów tablicy.
54    @param values częściowo wypełniona tablica
55    @param size liczba elementów tablicy values
56    @return suma elementów tablicy values
57 */
58 double sum(double values[], int size)
59 {
60    double total = 0;
61    for (int i = 0; i < size; i++)
62    {
63       total = total + values[i];
64    }
65    return total;
66 }
67
68 int main()
69 {
70    const int CAPACITY = 1000;
71    double scores[CAPACITY];
72    int size = read_inputs(scores, CAPACITY);
73    if (size == 0)
74    {
75       cout << "Niezbędny jest przynajmniej jeden wynik." << endl;
76    }
77    else
78    {
79       double final_score = sum(scores, size) - minimum(scores, size);
80       cout << "Wynik końcowy: " << final_score << endl;
81    }
82    return 0;
83 }
```

Przykładowy kod

Rozwiązanie z użyciem wektorów zamiast tablic znajduje się w kodach źródłowych do tego rozdziału w pliku *jak_to_zrobic_1/scores_vector*.

Przykład 6.1

Rzut kostką

Dowiedz się, jak ze zbioru wyników rzutów kostką wywnioskować, czy kostka jest „sprawiedliwa”. Odpowiedni kod znajdziesz w archiwum pobranym z serwera FTP wydawnictwa Helion (*ftp://ftp.helion.pl/przyklady/wpcpp3.zip*).

6.5. Rozwiązywanie problemów: odkrywanie algorytmów przez manipulację obiektami fizycznymi

W podrozdziale 6.4 dowiedziałeś się, jak rozwiązać problem przez połączenie i adaptację znanych algorytmów. Co jednak zrobić, gdy do rozwiązania zadania to nie wystarczy? W tym podrozdziale poznasz technikę wynajdowania niestandardowych algorytmów przez manipulację obiektami fizycznymi.

Zastanów się nad następującym zadaniem. Dysponujesz tablicą, której rozmiar jest liczbą parzystą, i masz zamienić miejscami pierwszą i drugą połówkę. Jeśli tablica zawiera np. osiem liczb

9	13	21	4	11	7	1	3

należy ją zamienić na

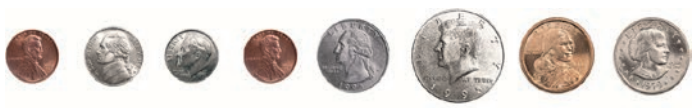
11	7	1	3	9	13	21	4

Dla wielu osób wymyślenie algorytmu wydaje się sporym wyzwaniem. Być może wiedzą, że niezbędna jest pętla, i zdają sobie sprawę, że trzeba wstawić (punkt 6.2.9) albo zamienić (punkt 6.2.10) elementy, ale brak im intuicji, by rysować schematy, opisać algorytm czy napisać pseudokod.

Do wizualizacji tablicy wartości można użyć zbioru monet, zestawu kart do gry lub zabawek.

Jedną z przydatnych technik wynajdowania algorytmów jest manipulacja obiektami fizycznymi. Rozpocznij od umieszczenia w rzędzie obiektów symbolizujących tablicę. Dobrym wyborem będą monety, karty do gry lub małe zabawki.

Tym razem ułożymy osiem monet.



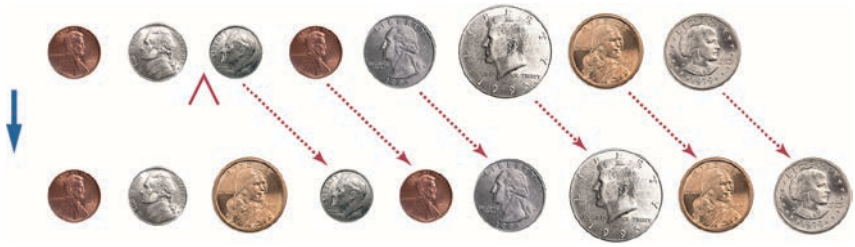
coins: © jamesbenet/iStockphoto; dollar coins: © JordiDelgado/iStockphoto.

Teraz cofnijmy się nieco i popatrzmy, w jaki sposób można zmienić kolejność monet.

Możemy usunąć monetę (punkt 6.2.8):

Wizualizacja usuwania elementu z tablicy

Możemy wstawić monetę (punkt 6.2.9):



Wizualizacja wstawiania elementu do tablicy

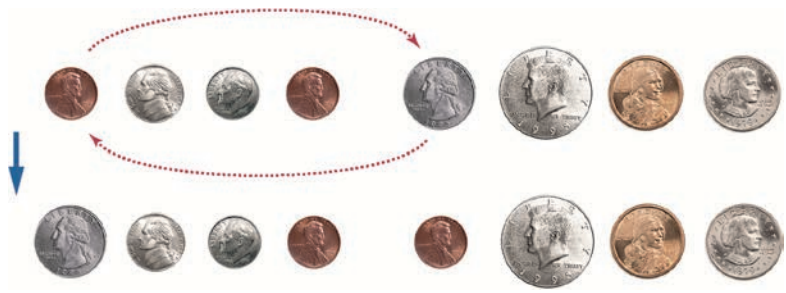
Możemy też zamienić dwie monety miejscami (punkt 6.2.10).

Wizualizacja przestawiania dwóch elementów

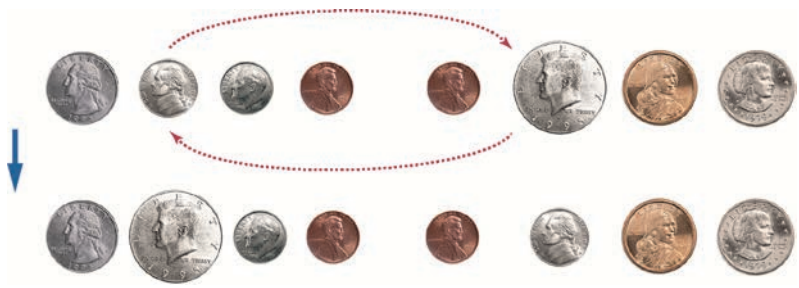
Śmiało ułóż teraz w rzędzie parę monet i spróbuj wykonać te trzy operacje, by poczuć, o co w tym chodzi.

Jak to może pomóc w problemie zamiany pierwszej i drugiej połowy tablicy?

Przestawmy pierwszą monetę na właściwe miejsce przez zamianę miejscami z piątą. Jako programiści C++ powiemy jednak, że zamieniamy monety na pozycjach 0 i 4:



Teraz zamienimy miejscami monety na pozycjach 1 i 5:



Jeszcze dwie zamiany i gotowe:

Teraz algorytm staje się oczywisty:

```
i = 0
j = . . . // Pomyślimy o tym za chwilę
Dopóki // Jeszcze nie wiadomo
  Zamień miejscami elementy na pozycjach i oraz j.
  i++
  j++
```

Jaka ma być początkowa wartość zmiennej *j*? Jeśli mamy osiem monet, to ta z pozycji zero jest przenoszona na pozycję 4. Ogólnie rzecz biorąc, jest przesuwana do środka tablicy, czyli na pozycję *rozmiar / 2*.

A ile wykonamy iteracji? Musimy zamienić miejscami wszystkie monety z pierwszej połowy tablicy, czyli *rozmiar / 2* monet. Pseudokod wygląda następująco:

```
i = 0
 j = rozmiar / 2
Dopóki i < rozmiar / 2
  Zamień miejscami elementy na pozycjach i oraz j.
  i++
  j++
```

Za wskaźniki pozycji lub liczniki mogą służyć spinacze do papieru.

Dobrze jest wykonać przegląd pseudokodu (podrozdział 4.2). Pozycje zmiennych *i* i *j* można zaznaczyć spinaczami do papieru. Jeśli przegląd zakończy się powodzeniem, będziemy wiedzieć, że w pseudokodzie nie ma „pomyłki o jeden".

Dla wielu osób manipulacja obiektami fizycznymi nie jest tak przerażająca, jak rysowanie schematów lub wyobrażanie sobie algorytmów. Wypróbuj tę metodę, gdy będziesz projektować nowy algorytm!

6.6. Tablice dwuwymiarowe

Często zachodzi potrzeba przechowywania kolekcji wartości, które mają układ dwuwymiarowy. Takie zbiory danych często pojawiają się w zastosowaniach finansowych i naukowych. Wartości zgrupowane w wierszach i kolumnach noszą nazwę **tablicy dwuwymiarowej** lub *macierzy*.

Zbadajmy, jak zapisać przykładowe dane przedstawione na rysunku 10.: liczbę medali w jeździe figurowej na lodzie zdobytych na zimowej olimpiadzie w 2014 r.

	Złote	Srebrne	Brązowe
Kanada	0	3	0
Włochy	0	0	1
Niemcy	0	0	1
Japonia	1	0	0
Kazachstan	0	0	1
Rosja	3	1	1
Korea Południowa	0	1	0
Stany Zjednoczone	1	0	1

Rysunek 10. Liczba medali w jeździe figurowej na lodzie

6.6.1. Definiowanie tablic dwuwymiarowych

Dane tabelaryczne przechowuje się w tablicy dwuwymiarowej.

W języku C++ tablice dwuwymiarowe zapisywane są w postaci tablic z indeksami wierszy i kolumn. Oto np. definicja tablicy o 8 wierszach i 3 kolumnach, nadającej się do przechowywania danych o liczbie medali:

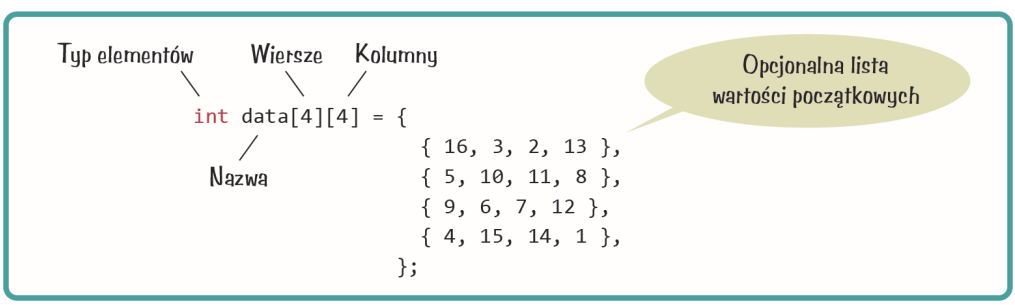
```
const int COUNTRIES = 8;
const int MEDALS = 3;
int counts[COUNTRIES][MEDALS];
```

Przy inicjalizacji tablicy grupuje się każdy wiersz danych w następujący sposób:

```
int counts[COUNTRIES][MEDALS] =
{
   { 0, 3, 0 },
   { 0, 0, 1 },
   { 0, 0, 1 },
   { 1, 0, 0 },
   { 0, 0, 1 },
   { 3, 1, 1 },
   { 0, 1, 0 }
   { 1, 0, 1 }
};
```

Podobnie jak jest w przypadku tablic jednowymiarowych, po zdefiniowaniu tablicy dwuwymiarowej nie można zmienić jej rozmiaru.

Składnia 6.2. Definicja tablicy dwuwymiarowej

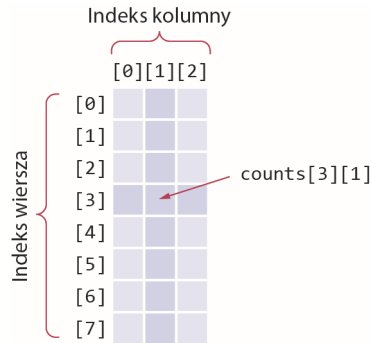


6.6.2. Dostęp do elementów

Dostęp do poszczególnych elementów tablicy dwuwymiarowej odbywa się przy użyciu dwóch indeksów, *tablica*[i][j].

Aby uzyskać dostęp do konkretnego elementu tablicy dwuwymiarowej, należy podać dwa indeksy w oddzielnych nawiasach kwadratowych w celu wybrania, odpowiednio, wiersza i kolumny (patrz sekcja „Składnia 6.2" i rysunek 11):

```
int value = counts[3][1];
```



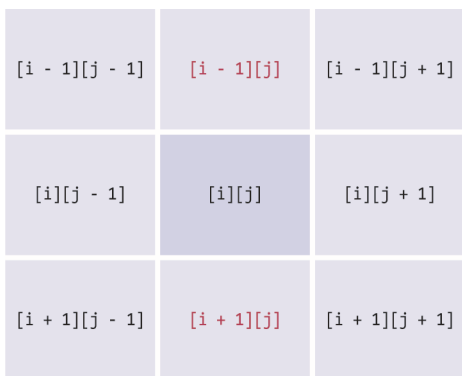
Rysunek 11. Dostęp do elementu tablicy dwuwymiarowej

Aby uzyskać dostęp do wszystkich wartości tablicy dwuwymiarowej, używa się dwóch zagnieżdżonych pętli. W pokazanej poniżej pętli wyświetlane są wszystkie elementy tablicy counts.

```
for (int i = 0; i < COUNTRIES; i++)
{
   // Przetwarzaj wiersz nr i
   for (int j = 0; j < MEDALS; j++)
   {
      // Przetwarzaj kolumnę nr j w wierszu nr i
      cout << setw(8) << counts[i][j];
}
cout << endl; // Po zakończeniu wiersza tablicy przejdź do nowego wiersza na ekranie
}
```

6.6.3. Lokalizowanie sąsiadujących elementów

W niektórych programach korzystających z tablic dwuwymiarowych trzeba zlokalizować elementy przyległe do podanego. Szczególnie często dokonuje się tego w grach. Wyznaczanie indeksów sąsiednich elementów zobrazowano na rysunku 12.



Rysunek 12. Sąsiadujące lokalizacje w tablicy dwuwymiarowej

Sąsiadami elementu counts[3][1] z lewej i z prawej strony są odpowiednio counts[3][0] i counts[3][2]. Sąsiadami z góry i z dołu są odpowiednio counts[2][1] i counts[4][1].

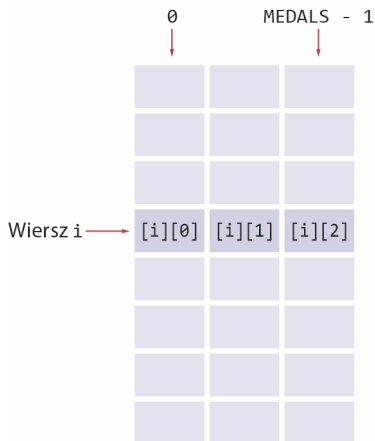
Należy zachować ostrożność podczas wyznaczania sąsiadów elementów znajdujących się na granicy tablicy. Element `counts[0][1]` nie ma np. sąsiada z góry. Zastanówmy się nad zadaniem obliczenia sumy wartości elementów przyległych do `count[i][j]` od góry i od dołu. Trzeba wtedy sprawdzić, czy element znajduje się w górnym lub dolnym wierszu tablicy:

```
int total = 0;
if (i > 0) { total = total + counts[i - 1][j]; }
if (i < ROWS - 1) { total = total + counts[i + 1][j]; }
```

6.6.4. Obliczanie sum wierszy i kolumn

Często spotykanym zadaniem jest obliczanie sum wartości w wierszach lub kolumnach. W naszym przypadku sumy wierszy informują o łącznej liczbie medali zdobytych przez reprezentantów każdego kraju.

Znalezienie odpowiednich wartości indeksów jest nieco skomplikowane i dobrze jest wykonać szybki szkic. Aby obliczyć sumę wartości w wierszu `i`, musimy odwiedzić następujące elementy:

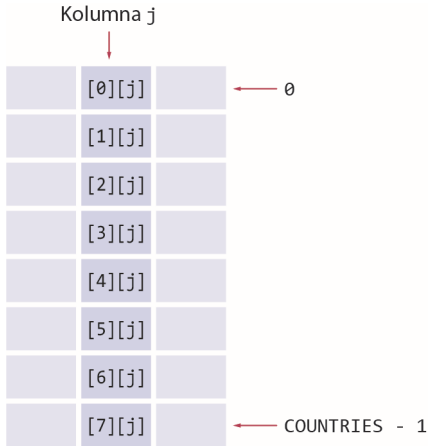


Jak widać, musimy obliczyć łączną wartość elementów `counts[i][j]`, gdzie `j` należy do zakresu od `0` do `MEDALS - 1`. Do obliczenia sumy służy następująca pętla:

```
int total = 0;
for (int j = 0; j < MEDALS; j++)
{
   total = total + counts[i][j];
}
```

Obliczanie sumy wartości w kolumnie wygląda podobnie. Należy policzyć łączną wartość elementów `counts[i][j]`, gdzie `i` należy do zakresu od `0` do `COUNTRIES - 1`.

```
int total = 0;
for (int i = 0; i < COUNTRIES; i++)
{
   total = total + counts[i][j];
}
```



6.6.5. Dwuwymiarowe parametry tablicowe

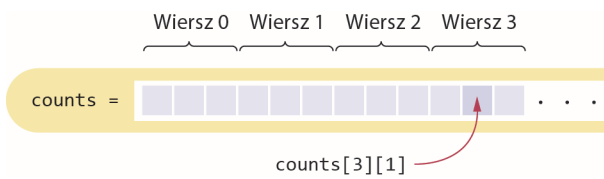
Dwuwymiarowy parametr tablicowy musi mieć stałą liczbę kolumn.

Przy definiowaniu funkcji z dwuwymiarowym parametrem tablicowym liczbę kolumn należy podać *jako stałą* w typie parametru. Ta funkcja oblicza sumę danego wiersza:

```
const int COLUMNS = 3;
int row_total(int table[][COLUMNS], int row)
{
   int total = 0;
   for (int j = 0; j < COLUMNS; j++)
   {
      total = total + table[row][j];
   }
   return total;
}
```

Funkcja ta oblicza sumę wiersza dwuwymiarowej tablicy o dowolnej liczbie wierszy, ale koniecznie 3 kolumnach. Aby policzyć sumy wierszy w dwuwymiarowej tablicy o 4 kolumnach, trzeba napisać kolejną funkcję.

Aby zrozumieć to ograniczenie, trzeba wiedzieć o tym, jak elementy tablicy są zapisane w pamięci. Chociaż tablica wygląda na dwuwymiarową, jej elementy są wciąż przechowywane w sekwencji liniowej. Na rysunku 13 widać, w jaki sposób wiersz po wierszu jest przechowywana tablica `counts`.



Rysunek 13. Tablica dwuwymiarowa jest przechowywana jako sekwencja wierszy

Aby dotrzeć np. do elementu

```
counts[3][1]
```

program musi pominąć wiersze 0, 1 i 2, a następnie zlokalizować w wierszu 3 przesunięcie (ang. *offset*) o 1. Przesunięcie od początku tablicy wynosi

$$3 \cdot \textit{liczba kolumn} + 1$$

teraz zastanówmy się nad funkcją `row_total`. Kompilator generuje kod znajdujący element

```
table[i][j]
```

przez obliczenie przesunięcia

```
i * COLUMNS + j
```

Kompilator korzysta z wartości podanej w drugiej parze nawiasów kwadratowych przy deklaracji parametru:

```
int row_total(int table[][COLUMNS], int row)
```

Warto pamiętać, że pierwsza para nawiasów kwadratowych musi być pusta, tak samo jak w przypadku tablic jednowymiarowych.

Funkcja `row_total` nie potrzebowała informacji o liczbie wierszy w tablicy. Jeśli to niezbędne, liczbę wierszy należy przekazać w zmiennej, tak jak w tym przykładzie:

```
int column_total(int table[][COLUMNS], int rows, int column)
{
   int total = 0;
   for (int i = 0; i < rows; i++)
   {
      total = total + table[i][column];
   }
   return total;
}
```

Pracę z tablicami dwuwymiarowymi zilustrowano w pokazanym poniżej programie. Program ten wyświetla liczby poszczególnych medali i sumy wierszy.

podr06/medals.cpp

```
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

const int COLUMNS = 3;

/**
   Oblicza sumę wiersza tablicy.
   @param table tablica o trzech kolumnach
   @param row wiersz, którego sumę trzeba obliczyć
   @return suma wszystkich elementów w podanym wierszu
*/
```

```
15 double row_total(int table[][COLUMNS], int row)
16 {
17    int total = 0;
18    for (int j = 0; j < COLUMNS; j++)
19    {
20       total = total + table[row][j];
21    }
22    return total;
23 }
24
25 int main()
26 {
27    const int COUNTRIES = 8;
28    const int MEDALS = 3;
29
30    string countries[] =
31    {
32       "Kanada",
33       "Włochy",
34       "Niemcy",
35       "Japonia",
36       "Kazachstan",
37       "Rosja",
38       "Korea Płd.",
39       "USA"
40    };
41
42    int counts[COUNTRIES][MEDALS] =
43    {
44       { 0, 3, 0 },
45       { 0, 0, 1 },
46       { 0, 0, 1 },
47       { 1, 0, 0 },
48       { 0, 0, 1 },
49       { 3, 1, 1 },
50       { 0, 1, 0 },
51       { 1, 0, 1 }
52    };
53
54    cout << "            Kraj    Złote Srebrne Brązowe    Suma" << endl;
55
56    // Wyświetl kraje, liczby poszczególnych medali i sumy wierszy.
57    for (int i = 0; i < COUNTRIES; i++)
58    {
59       cout << setw(15) << countries[i];
60       // Przetwarzaj wiersz nr i.
61       for (int j = 0; j < MEDALS; j++)
62       {
63          cout << setw(8) << counts[i][j];
64       }
65       int total = row_total(counts, i);
66       cout << setw(8) << total << endl;
67    }
68
69    return 0;
70 }
```

Przebieg działania programu

```
      Kraj   Złote Srebrne Brązowe    Suma
   Kanada       0       3       0       3
   Włochy       0       0       1       1
   Niemcy       0       0       1       1
  Japonia       1       0       0       1
Kazachstan      0       0       1       1
    Rosja       3       1       1       5
Korea Płd.      0       1       0       1
      USA       1       0       1       2
```

Często popełniany błąd 6.2

Pomijanie rozmiaru kolumny w dwuwymiarowym parametrze tablicowym

Przy przekazywaniu do funkcji tablicy jednowymiarowej w osobnej zmiennej parametrycznej podaje się rozmiar tablicy:

```
void print(double values[], int size)
```

Dzięki tej funkcji można wypisać zawartość tablic o dowolnym rozmiarze. W przypadku tablic dwuwymiarowych nie można jednak po prostu przekazać w zmiennych parametrycznych liczby wierszy i kolumn:

```
void print(double table[][], int rows, int cols) // NIE!
```

Funkcja musi *w czasie kompilacji* mieć informację o tym, ile kolumn ma tablica dwuwymiarowa. Liczbę kolumn należy podać w tablicowej zmiennej parametrycznej. Musi to być stała:

```
const int COLUMNS = 3;
void print(const double table[][COLUMNS], int rows) // OK
```

Dzięki tej funkcji można wyświetlić zawartość tablic o dowolnej liczbie wierszy, ale stałym rozmiarze kolumn.

Przykład 6.2

Tabela danych o ludności świata

Dowiedz się, jak wyświetlić dane o ludności świata w tabeli z nagłówkami wierszy i kolumn oraz sumami dla każdej kolumny danych. Odpowiedni kod znajdziesz w archiwum pobranym z serwera FTP wydawnictwa Helion (*ftp://ftp.helion.pl/przyklady/wpcpp3.zip*).

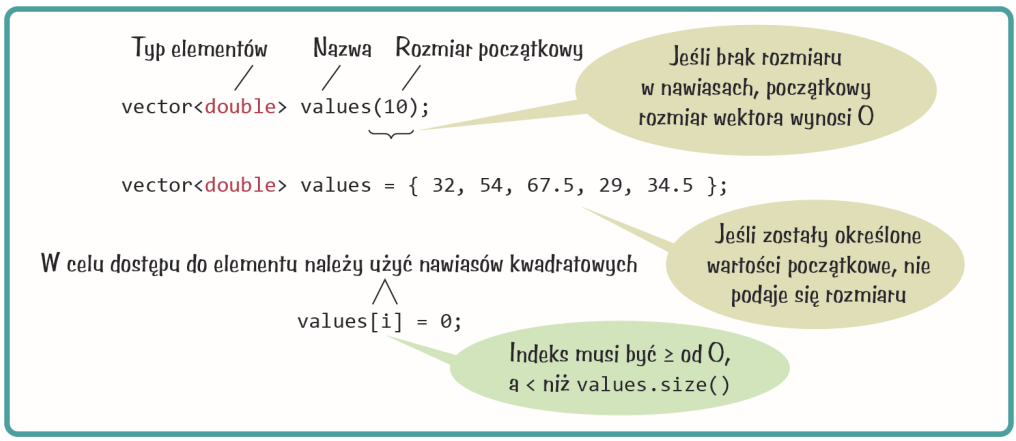
6.7. Wektory

Wektor przechowuje sekwencję wartości, której rozmiar może ulegać zmianie.

Podczas pisania programu zbierającego dane wprowadzane przez użytkownika nie zawsze wiadomo, ile ich będzie. Rozmiar tablicy musi być niestety znany *podczas kompilacji programu*.

W punkcie 6.1.3 zobaczyłeś, jak rozwiązać ten problem za pomocą częściowo wypełnionych tablic. Wygodniejsze rozwiązanie zapewnia omawiana w następnych punktach struktura zwana **wektorem**. Podobnie jak tablica służy on do zbierania sekwencji wartości, ale jego rozmiar może ulegać zmianie.

Składnia 6.3. Definiowanie wektora



6.7.1. Definiowanie wektorów

Przy definiowaniu wektora należy w nawiasach trójkątnych podać typ elementów, tak jak tu:

```
vector<double> values;
```

Opcjonalnie można podać rozmiar początkowy. Oto np. definicja wektora o rozmiarze początkowym 10:

```
vector<double> values(10);
```

Jeśli przy definiowaniu wektora nie zostanie podany rozmiar początkowy, będzie on wynosił 0. Chociaż nie ma sensu definiować tablic o takim rozmiarze, to często przydają się wektory, których *początkowy* rozmiar wynosi 0, a potem jest zwiększany w miarę potrzeb.

Zgodnie ze standardem C++ 11 i następnymi wektorom można nadać wartości początkowe przy użyciu tej samej notacji, co tablicom:

```
vector<double> values = { 32, 54, 67.5, 29, 34.5 };
```

Aby móc korzystać w programie z wektorów, należy dołączyć nagłówek `<vector>`.

Dostęp do elementów wektora uzyskuje się podobnie jak w przypadku tablic przy użyciu zapisu `values[i]`.

> Aby pobrać aktualny rozmiar wektora, należy użyć jego funkcji składowej `size`.

Aktualny rozmiar wektora zwraca jego funkcja składowa `size`. W pętli, w której odwiedzane są wszystkie elementy wektora, funkcji tej używa się następująco:

```
for (int i = 0; i < values.size(); i++)
{
   cout << values[i] << endl;
}
```

Inne przykłady definicji wektorów zawiera tabela 2.

Tabela 2. Definiowanie wektorów

`vector<int> numbers(10);`	Wektor dziesięciu liczb całkowitych.
`vector<string> names(3);`	Wektor trzech ciągów.
`vector<double> values;`	Wektor o rozmiarze 0.
`vector<double> values();`	**Błąd:** To nie jest definicja wektora, ale funkcji zwracającej wektor.

`vector<int> numbers;`
`for (int i = 1; i <= 10; i++)`
`{`
`  numbers.push_back(i);`
`}`
	Wektor z dziesięcioma liczbami całkowitymi, wypełniony wartościami 1, 2, 3, …, 10.

`vector<int> numbers(10);`
`for (int i = 0; i < numbers.size(); i++)`
`{`
`  numbers[i] = i + 1;`
`}`
	Inny sposób definiowania wektora dziesięciu liczb całkowitych i wprowadzenia do niego wartości 1, 2, 3, …, 10.

`vector<int> numbers =` `  { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };`	Ta składnia jest obsługiwana w standardzie C++ 11 i nowszych.

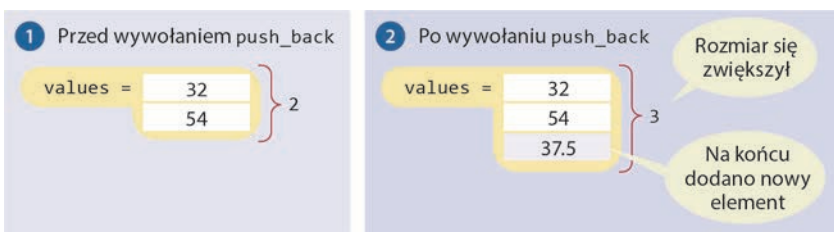
6.7.2. Powiększanie i zmniejszanie wektorów

Aby dodać do wektora więcej elementów, należy użyć funkcji składowej `push_back`. Aby zmniejszyć jego rozmiar, używa się funkcji `pop_back`.

Jeśli potrzebny jest kolejny element wektora, można dodać go na koniec przy użyciu funkcji `push_back`, która powoduje zwiększenie rozmiaru wektora o 1. Ponieważ jest to funkcja składowa, wywołuje się ją za pomocą zapisu kropkowego, np. tak:

```
values.push_back(37.5);
```

Załóżmy, że przed wywołaniem funkcji `push_back` wektor `values` miał 2 elementy o wartościach 32 i 54 (rysunek 14). Po wywołaniu rozmiar wektora wynosi 3, a element `values[2]` zawiera wartość 37,5.



Rysunek 14. Dodawanie elementu za pomocą funkcji push_back

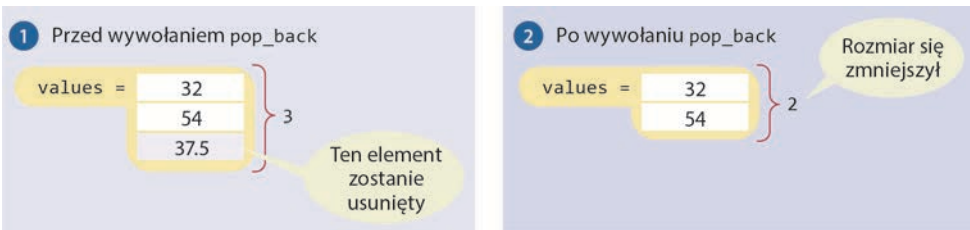
Funkcja składowa push_back jest często używana do wprowadzenia do wektora danych wejściowych.

```
vector<double> values; // Początkowo pusty
double input;
while (cin >> input)
{
   values.push_back(input);
}
```

Warto zauważyć, że odczyt danych wejściowych w takiej pętli jest *dużo* prostszy niż w tej z punktu 6.2.11.

Inna funkcja składowa, pop_back, usuwa ostatni element wektora i zmniejsza jego rozmiar o jeden (rysunek 15):

```
values.pop_back();
```



Rysunek 15. Usuwanie elementu za pomocą funkcji pop_back

6.7.3. Wektory a funkcje

Wektory mogą być używane jako argumenty funkcji i wartości zwracane.

Wektorów, zupełnie tak samo jak innych wartości, można używać w roli argumentów funkcji. Przedstawiona poniżej funkcja oblicza sumę elementów wektora liczb zmiennoprzecinkowych:

```
double sum(vector<double> values)
{
   double total = 0;
   for (int i = 0; i < values.size(); i++)
   {
      total = total + values[i];
   }
   return total;
}
```

Aby móc modyfikować zawartość wektora, należy użyć parametru referencyjnego.

W funkcji tej elementy wektora są odwiedzane, ale nie modyfikowane. Jeśli funkcja ma modyfikować elementy, należy użyć parametru referencyjnego. Przedstawiona poniżej funkcja mnoży wszystkie liczby w wektorze przez podany czynnik.

```
void multiply(vector<double>& values, double factor) // Proszę zwrócić uwagę na znak &
{
   for (int i = 0; i < values.size(); i++)
   {
```

```
      values[i] = values[i] * factor;
   }
}
```

W przypadku parametrów wektorowych, które nie są modyfikowane, dobrze jest używać stałych referencji („Temat specjalny 5.2”), np.:

```
double sum(const vector<double>& values) // dla polepszenia wydajności dodano const &
```

Funkcja może zwracać wektor.

Funkcja może zwracać wektor. Również i w tym przypadku wektory niczym się nie różnią od innych wartości. Należy po prostu zbudować wynik w obrębie funkcji i zwrócić go. W tym przypadku funkcja `squares` zwraca wektor zawierający drugie potęgi liczb, od 0^2 aż do $(n - 1)^2$:

```
vector<int> squares(int n)
{
   vector<int> result;
   for (int i = 0; i < n; i++)
   {
      result.push_back(i * i);
   }
   return result;
}
```

Jak widać, użycie wektorów w funkcjach jest proste — nie ma żadnych specjalnych reguł, o których trzeba pamiętać.

Przykładowy kod

Program demonstrujący użycie wektorów i funkcji znajduje się w podkatalogu *podr07_03* kodów źródłowych do tego rozdziału.

6.7.4. Algorytmy związane z wektorami

Większość algorytmów z podrozdziału 6.2 można bez zmian zastosować do wektorów — wystarczy zamienić *rozmiar* tablicy `values` na wywołanie `values.size()`. W tym punkcie omówimy algorytmy, które wykazują różnice w wersji dla wektorów.

Kopiowanie

Jak wspomniano w punkcie 6.2.2, w celu wykonania kopii tablicy trzeba w sposób jawny użyć pętli. W przypadku wektora jest to znacznie łatwiejsze. Wystarczy przypisać go do innego wektora. Warto spojrzeć na ten przykład:

```
vector<int> squares;
for (int i = 0; i < 5; i++) { squares.push_back(i * i); }
vector<int> lucky_numbers; // Na początku pusty
lucky_numbers = squares;   // Teraz wektor lucky_numbers zawiera te same elementy, co squares
```

Znajdowanie pasujących elementów

W punkcie 6.2.7 pokazano, jak znaleźć pierwszy pasujący element, ale czasem potrzebne są wszystkie. Zebranie dopasowań w tablicy jest uciążliwe, ale w przypadku wektora jest to łatwe. Oto przykład, jak zgromadzić w nim wszystkie elementy większe od 100:

```
vector<double> matches;
for (int i = 0; i < values.size(); i++)
{
   if (values[i] > 100)
   {
      matches.push_back(values[i]);
   }
}
```

Usuwanie elementu

Przy usuwaniu z wektora elementu trzeba skorygować rozmiar wektora przez wywołanie jego funkcji składowej `pop_back`. Oto kod pozwalający usunąć element z pozycji `[pos]` w przypadku, gdy kolejność nie ma znaczenia.

```
int last_pos = values.size() - 1;
values[pos] = values[last_pos]; // Zastąp element na pozycji pos ostatnim.
values.pop_back();              // Usuń ostatni element.
```

Przy usuwaniu elementu z wektora uporządkowanego najpierw trzeba przenieść elementy, a potem zmniejszyć rozmiar wektora:

```
for (int i = pos + 1; i < values.size(); i++)
{
   values[i - 1] = values[i];
}
values.pop_back();
```

Wstawianie elementu

Wstawianie elementu na końcu wektora nie wymaga specjalnego kodu. Wystarczy zastosować funkcję składową `push_back`.

Przy wstawianiu elementu w środku również trzeba wywołać funkcję `push_back`, by zwiększyć rozmiar wektora. Używa się następującego kodu:

```
int last_pos = values.size() - 1;
values.push_back(values[last_pos]);
for (int i = last_pos; i > pos; i--)
{
   values[i] = values[i - 1];
}
values[pos] = new_element;
```

Przykładowy kod

W podkatalogu *podr07_04* kodów źródłowych do tego rozdziału znajduje się program, w którym problem postawiony w podrozdziale 6.1 rozwiązano przy użyciu wektorów.

6.7.5. Wektory dwuwymiarowe

Wektory dwuwymiarowe jako takie nie istnieją, ale do przechowywania wierszy i kolumn można użyć wektora wektorów. Liczby medali z podrozdziału 6.6 można zapisać w zmiennej:

```
vector<vector<int>> counts;
```

Potraktuj `counts` jako wektor wierszy. Każdy z nich to `vector<int>`.

Taki wektor wektorów trzeba zainicjować, by znalazły się wiersze i kolumny dla wszystkich elementów.

```
for (int i = 0; i < COUNTRIES; i++)
{
   vector<int> row(MEDALS);
   counts.push_back(row);
}
```

Teraz dostęp do elementów `counts[i][j]` można uzyskać tak samo, jak w przypadku tablic dwuwymiarowych. Wyrażenie `counts[i]` oznacza wiersz nr `i`, a `counts[i][j]` to wartość kolumny nr `j` w tym wierszu.

Wektory wektorów w porównaniu z tablicami dwuwymiarowymi mają pewną zaletę. W chwili kompilacji rozmiary wierszy i kolumn nie muszą być ustalone. Nawet jeśli liczba krajów i typów medali nie będzie od początku znana, wektor `counts` da się skonstruować:

```
int countries = . . .;
int medals = . . .;
vector<vector<int>> counts;
for (int i = 0; i < countries; i++)
{
   vector<int> row(medals);
   counts.push_back(row);
}
```

Nie można natomiast zadeklarować tablicy `int[countries][medals]`, ponieważ jej granice nie są stałe.

Podczas przetwarzania wektora wektorów liczbę wierszy i kolumn pobiera się następująco:

```
vector<vector<int>> values = . . .;
int rows = values.size();
int columns = values[0].size();
```

Należy pamiętać o tym, że poszczególne elementy wektora `values` stanowią wiersze, więc wartość funkcji `values.size()` to ich liczba. Rozmiar dowolnego wiersza stanowi liczbę kolumn.

Możliwe jest też zadeklarowanie wektora wektorów z różnymi rozmiarami wierszy. Aby uzyskać np. taki kształt trójkąta:

```
t[0][0]
t[1][0] t[1][1]
t[2][0] t[2][1] t[2][2]
t[3][0] t[3][1] t[3][2] t[3][3]
```

należy dodawać wiersze o odpowiednich rozmiarach w następujący sposób:

```
vector<vector<int>> t;
for (int i = 0; i < 3; i++)
{
   vector<int> row(i + 1);
   t.push_back(row);
}
```

Każdy z elementów tablicy jest nadal dostępny jako `t[i][j]`. Wyrażenie `t[i]` pozwala wybrać wiersz nr `i`, a operator `[j]` — element nr `j` w tym wierszu.

Wskazówka dla programistów 6.2
Stosuj wektory zamiast tablic

W większości zadań programistycznych łatwiej użyć wektorów niż tablic. Wektory można powiększać i zmniejszać, a jeśli nawet cały czas mają taki sam rozmiar, wygodne jest to, że go pamiętają. Przed pojawieniem się standardu C++ 11 niektórzy programiści wybierali tablice, ponieważ nie było wygodnej składni inicjacji wektorów. Zaawansowani programiści czasem preferują użycie tablic, bo są one odrobinę bardziej wydajne. Poza tym umiejętność korzystania z tablic jest niezbędna przy pracy nad starszymi programami.

Temat specjalny 6.5
Pętla for oparta na zakresie

W standardzie C++ 11 wprowadzono wygodną składnię odwiedzania wszystkich elementów w „zakresie”, czyli pewnej sekwencji.

W tej pętli wyświetlane są wszystkie elementy wektora:

```
vector<int> values = {1, 4, 9, 16, 25, 36};
for (int v : values)
{
   cout << v << " ";
}
```

W każdej iteracji pętli zmiennej v jest przypisywany element wektora. Warto zauważyć, że nie używa się zmiennej indeksującej. Wartością zmiennej v jest element, a nie jego indeks.

Aby móc modyfikować elementy, należy zadeklarować zmienną pętli jako referencję:

```
for (int& v : values)
{
   v++;
}
```

W pętli tej inkrementowane są wszystkie elementy wektora.

Przy określaniu typu zmiennej elementowej można użyć słowa kluczowego `auto`, przedstawionego w sekcji „Temat specjalny 2.3”:

```
for (auto v : values) { cout << v << " "; }
```

Oparta na zakresie pętla `for` działa również w przypadku tablic, których rozmiar jest znany w czasie kompilacji[KB4]:

```
int primes[] = { 2, 3, 5, 7, 11, 13 };
for (int p : primes)
{
   cout << p << " ";
}
```

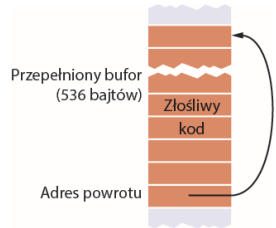
Pętla `for` oparta na zakresie (ang. *range based for loop*) to wygodny, skrócony sposób zapisu operacji odwiedzania lub aktualizacji wszystkich elementów wektora lub tablicy. W tej książce nie jest ona używana, bo ten sam wynik można osiągnąć dzięki przejściu w pętli wszystkich indeksów tablicy. Jeśli jednak spodobała Ci się ta zwięźlejsza forma i korzystasz ze standardu C++ 11 lub nowszego, powinieneś pomyśleć, czy nie warto jej stosować.

Przykładowy kod

Program demonstrujący opartą na zakresie pętlę `for` znajduje się w podkatalogu *temat_specjalny_5* kodów źródłowych do tego rozdziału.

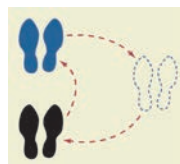
Podsumowanie rozdziału

Tablice służą do zbierania wartości.



- Tablica służy do zbierania sekwencji wartości tego samego typu.
- Dostęp do poszczególnych *wartości* tablicy jest możliwy przy użyciu notacji `values[i]`, gdzie indeks `i` jest liczbą całkowitą.
- Element tablicy może być używany jak każda zmienna.
- Wartość indeksu tablicy musi wynosić co najmniej zero, ale mniej niż jej rozmiar.
- Przekroczenie indeksu, które występuje wtedy, gdy zostanie podany niewłaściwy indeks tablicy, może doprowadzić do uszkodzenia danych lub spowodować awarię programu.
- W przypadku częściowo wypełnionej tablicy należy zapisywać jej aktualny rozmiar w zmiennej towarzyszącej.

Używanie typowych algorytmów tablicowych.



- Aby skopiować tablicę, należy skopiować wszystkie jej elementy do nowej tablicy przy użyciu pętli.
- Podczas rozdzielania elementów przed pierwszym nie należy wstawiać separatora.
- Podczas wyszukiwania liniowego elementy są sprawdzane po kolei aż do znalezienia dopasowania.
- Przed wstawieniem elementu do tablicy należy przenieść elementy na jej koniec, *począwszy od ostatniego*.
- Przy zamianie dwóch elementów trzeba użyć zmiennej tymczasowej.

Implementacja funkcji przetwarzających tablice.

- Przy przesyłaniu do funkcji tablicy należy również przekazać jej rozmiar.
- Parametry tablicowe są zawsze typu referencyjnego.
- Funkcja nie może zwracać typu tablicowego.
- Jeśli funkcja modyfikuje rozmiar tablicy, musi poinformować o tym wywołującego.
- Funkcja dodająca elementy do tablicy musi mieć informację o jej pojemności.

Łączenie i adaptowanie algorytmów w celu rozwiązania zadania programistycznego.

- Złożone zadania programistyczne można rozwiązywać dzięki łączeniu podstawowych algorytmów.
- Należy poznać implementację podstawowych algorytmów po to, by móc je dostosowywać do potrzeb.

Odkrywanie algorytmów przez manipulację obiektami fizycznymi.

- Do wizualizacji tablicy wartości można użyć zbioru monet, zestawu kart do gry lub zabawek.
- Za wskaźniki pozycji lub liczniki mogą służyć spinacze do papieru.

Wykorzystanie tablic dwuwymiarowych do przechowywania danych uporządkowanych w wierszach i kolumnach.

- Dane tabelaryczne przechowuje się w tablicy dwuwymiarowej.
- Dostęp do poszczególnych elementów tablicy dwuwymiarowej odbywa się przy użyciu dwóch indeksów, `tablica[i][j]`.
- Dwuwymiarowy parametr tablicowy musi mieć stałą liczbę kolumn.

Wykorzystanie wektorów do zarządzania kolekcjami o zmiennym rozmiarze.

- Wektor przechowuje sekwencję wartości, której rozmiar może ulegać zmianie.
- Aby pobrać aktualny rozmiar wektora, należy użyć jego funkcji składowej `size`.
- Aby dodać do wektora więcej elementów, należy użyć funkcji składowej `push_back`. Aby zmniejszyć jego rozmiar, używa się funkcji `pop_back`.
- Wektory mogą być używane jako argumenty funkcji i wartości zwracane.
- Aby móc modyfikować zawartość wektora, należy użyć parametru referencyjnego.
- Funkcja może zwracać wektor.

Bądź profesjonalistą od pierwszej linii kodu!

C++ wyjątkowo dobrze nadaje się do nauki jako „pierwszy poważny język programowania", a równocześnie jest znakomitym, wszechstronnym narzędziem do tworzenia nowoczesnych aplikacji. Programowanie w nim wymaga nieco wysiłku, ale pozwala zarówno szybko przyswoić najważniejsze paradygmaty informatyki, jak i wdrożyć się w pisanie kodu zgodnie z dobrymi praktykami. Przy tym C++ ma cechy nowoczesnego języka programowania, a jego możliwości są poszerzane dzięki bibliotekom. Aby adept sztuki programowania mógł skorzystać z tych wszystkich zalet, potrzebuje pomocy dobrego nauczyciela lub dobrego podręcznika. Tylko w ten sposób uniknie typowych błędów popełnianych na skutek niewystarczającego zrozumienia podstaw programowania i najważniejszych koncepcji programistycznych.

Oto polska edycja popularnego podręcznika opartego na naukowych podstawach skutecznego uczenia. To znakomite wprowadzenie do podstawowych technik programowania i umiejętności projektowania pozwala łatwo, a przy tym dogłębnie opanować elementarne pojęcia. W przystępny sposób omawia najistotniejsze kwestie działania algorytmów i rodzaje struktur danych. Krok po kroku przeprowadza czytelnika od podstaw do bardziej zaawansowanych tematów związanych ze współczesnymi aplikacjami, takich jak GUI i programowanie XML. Poszczególne koncepcje zostały tu wyjaśnione z wykorzystaniem trafnie dobranych schematów i grafik. Nie zabrakło też wskazówek, przykładów i obszernych fragmentów świetnie napisanego kodu, które ułatwiają naukę, podobnie jak liczne ćwiczenia i studia przypadków.

W tej książce między innymi:

- podstawowe koncepcje programowania: struktury sterujące, tablice, wskaźniki
- programowanie obiektowe, dziedziczenie, polimorfizm
- struktury danych: liniowe i oparte na drzewach
- wprowadzenie do algorytmów
- szablony i zarządzanie pamięcią

Cay Horstmann — autor najpopularniejszych w Polsce podręczników do nauki programowania, ceniony za bogate doświadczenie dydaktyczne także w innych krajach. Jest profesorem informatyki na Uniwersytecie Stanowym w San Jose. Zdobył tytuł Java Champion, często występuje jako prelegent na konferencjach programistycznych. Urodził się w północnych Niemczech, obecnie mieszka w USA.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-6728-9

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 199,00 zł

WILEY