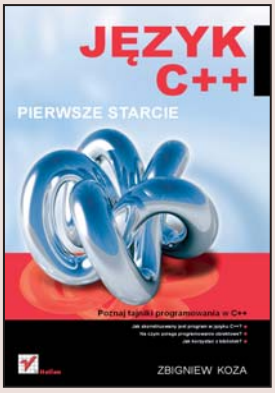
Język C++. Pierwsze starcie

Autor: Zbigniew Koza
ISBN: 978-83-246-1481-3
Format: B5, stron: 288



Poznaj tajniki programowania w C++

- Jak skonstruowany jest program w języku C++?
- Na czym polega programowanie obiektowe?
- Jak korzystać z bibliotek?

C++ to jeden z najpopularniejszych języków programowania. Stosowany jest zarówno przez profesjonalistów, jak i hobbystów. Wszyscy jego użytkownicy doceniają elastyczność, ogromne możliwości i szybkość działania napisanych w nim programów. Ogromną zaletą C++ jest to, że nie wymusza na programistach stosowania określonego stylu programowania. Z racji swoich możliwości jest to język bardzo złożony, a efektywne programowanie w nim wymaga poznania wielu technik i pojęć oraz umiejętności wykorzystania tej wiedzy w praktyce.

Książka „C++. Pierwsze starcie" to podręcznik, dzięki któremu opanujesz zasady programowania w tym języku i zdobędziesz solidne podstawy do dalszego rozwijania swoich umiejętności. Znajdziesz w niej opis aktualnego standardu C++ oraz omówienia narzędzi programistycznych i bibliotek. Poznasz elementy języka, zasady programowania obiektowego i tworzenia złożonych aplikacji. Przeczytasz o szablonach, bibliotece STL i obsłudze błędów. Dowiesz się, jak stosować biblioteki przy tworzeniu aplikacji dla systemów Windows i Linux.

- Struktura programu w C++
- Elementy języka
- Korzystanie z funkcji
- Programowanie obiektowe
- Dynamiczne struktury danych
- Stosowanie bibliotek
- Szablony
- Biblioteka STL
- Obsługa błędów

Zrób pierwszy krok na drodze do profesjonalnego tworzenia oprogramowania

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

Spis treści

Rozdział 4

Tablice i wskaźniki

Wskaźniki. Tablice. Tablice znaków a literały napisowe. Tablice a wskaźniki. Operatory `new` *i* `delete`*. Pamięć wolna.*

Poznane w rozdziale 2. typy arytmetyczne (np. `int`, `double`, `bool`, `char`) nie wyczerpują zestawu typów wbudowanych języka C++. Do omówienia pozostały m.in. tablice i wskaźniki, czyli te typy danych, które są charakterystyczne dla języka C, a których użyteczność w C++ jest mocno ograniczona. Niemniej jednak jest to bardzo ważny rozdział, gdyż: (i) bez zrozumienia zasad posługiwania się wskaźnikami i tablicami nie można korzystać z żadnej biblioteki C ani zrozumieć sposobu funkcjonowania wielu bibliotek C++; (ii) w pewnych sytuacjach wskaźniki i tablice są niezastąpione także w C++; (iii) wskaźnik jest naturalnym przykładem iteratora.

4.1. Wskaźniki

4.1.1. Definiowanie wskaźników

Wskaźnik to obiekt przechowujący adres innego obiektu. Oto prosty przykład użycia wskaźnika:

Wydruk 4.1. Przykład użycia wskaźnika

```
3  int main()
   {
5      int x = 7;
       std::cout << "Zmienna x znajduje sie pod adresem " << &x << "\n";
       int* wsk = &x;                    // <-- zmienna 'wsk' przechowuje adres x
       std::cout << "Pod adresem " << wsk << " znajduje sie liczba " << *wsk << "\n";
       *wsk = 10;
10     std::cout << "Teraz x = " << x << "\n";
   }
```

```
Zmienna x znajduje sie pod adresem 0x22ff74
Pod adresem 0x22ff74 znajduje sie liczba 7
Teraz x = 10
```

Do odczytywania adresu obiektu w pamięci operacyjnej służy operator `&`. W tym kontekście został on użyty w wierszu 6 do wyświetlenia na ekranie adresu zmiennej `x`. Fakt, że operator pobrania adresu zapisywany jest za pomocą tego samego symbolu co operator koniunkcji bitowej i symbol referencji, sprawia początkującym pewne problemy. Z formalnego punktu widzenia zasadnicza różnica pomiędzy operatorem pobrania adresu a operatorem koniunkcji bitowej polega na tym, że pierwszy z nich jest operatorem jednoargumentowym, a drugi – dwuargumentowym. Z kolei wybór symbolu `&` na oznaczenie referencji odzwierciedla fakt, że przekazywanie argumentów funkcji przez referencję oznacza tak naprawdę przekazywanie adresów argumentów faktycznych.

Skoro potrafimy pobrać adres operatorem `&`, dobrze byłoby mieć też możliwość zapisania go w specjalnej zmiennej. Taką zmienną – zwaną *wskaźnikiem* – należy oczywiście wpierw zdefiniować. W tym celu między nazwą klasy obiektu, którego adres nasza zmienna ma przechowywać, a jej nazwą wstawiamy gwiazdkę – i to wszystko! Przykład zastosowania tej procedury znajduje się w wierszu 7, w którym zdefiniowano zmienną o nazwie `wskaznik`.

```
int* wsk = &x;                    // <-- zmienna 'wsk' przechowuje adres x
```

W definicji tej na lewo od gwiazdki znajduje się identyfikator typu (tu: `int`), dlatego zmienna `wskaznik` może przechowywać wyłącznie adresy liczb typu `int`. Typem zmiennej `wskaznik` jest `int*`.

W wierszu 7 skorzystano też z okazji, by zadośćuczynić regule 1.5 (ze s. 28) i definicję zmiennej adresowej uzupełnić informacją o jej wartości początkowej – w tym przypadku jest nią adres zmiennej `x`.

Skoro mamy już zdefiniowaną zmienną wskaźnikową, pora jej użyć. Możemy z nią zrobić dwie rzeczy. Po pierwsze, możemy uzyskać dostęp do obiektu, którego adres jest w niej zapisany. W tym celu należy poprzedzić jej nazwę gwiazdką, która w tym kontekście zwie się operatorem wyłuskania (gwiazdka „wyłuskuje" wartość obiektu wskazywanego przez wskaźnik). Operator `*` ma więc tę samą właściwość, co operator `&`: występuje jako operator jednoargumentowy (i zwie się operatorem wyłuskania) lub dwuargumentowy (jako operator mnożenia). Zastosowanie operatora wyłuskania ilustrują wiersze 9 i 10. Pierwszy z nich dowodzi, że za pośrednictwem operatora `*` możemy odczytywać wartość obiektu wskazywanego przez wskaźnik. Wiersz 10 pokazuje natomiast, że za pośrednictwem tego operatora można także modyfikować zmienną wskazywaną przez wskaźnik.

Drugą czynnością, którą można zrobić ze zmienną wskaźnikową, to zmienienie jej wartości. Ilustruje to następujący przykład:

```
int x = 0, y = 1;   // definiujemy dwie różne zmienne całkowite
int* p = &x;        // p wskazuje na x
...                 // używamy p jako wskaźnika na x
p = &y;             // odtąd p wskazuje na y
```

Jak widać, dopóki nie używamy operatora wyłuskania, wskaźniki zachowują się jak zwyczajne zmienne. Należy tylko pamiętać o prostej zasadzie:

Reguła 4.1. *Jeżeli* `X` *i* `Y` *oznaczają różne typy, to (z wyjątkiem nielicznych sytuacji opisanych w dalszej części podręcznika) wartości wskaźnika typu* `X*` *nie można przypisywać zmiennej wskaźnikowej typu* `Y*`.

Różnica między referencjami i wskaźnikami jest w sumie niewielka – referencje są na stałe przypisane obiektom, do których się odnoszą, przy czym poprawność tego przy-

pisania gwarantowana jest przez kompilator; natomiast wskaźniki mogą zmieniać swoją wartość, czyli wskazywać na co i rusz inne obiekty, a nawet na obszar pamięci, w którym nie ma żadnych obiektów. Różnica niby niewielka, ale zasadnicza. Dzięki ich „stałości" i udogodnieniom składniowym (brak „gwiazdek") stosowanie referencji jest bezpieczne, natomiast stosowanie wskaźników wymaga wyobraźni i lat doświadczenia.

Reguła 4.2. *Jeżeli naprawdę nie musisz, nie używaj wskaźników.*

4.1.2. Wskaźniki typu `void*`, czyli wycieczka w stronę C

Ponieważ w języku C nie ma referencji, wskaźniki są w nim wręcz wszechobecne – stanowią bowiem jedyną metodę przekazywania do funkcji obiektów tak, aby funkcja mogła je modyfikować. Jak wiemy, w C++ zamiast wskaźników używa się w tym celu referencji. Jednak miliony wierszy kodu napisanego w C korzystają wyłącznie ze wskaźników, musimy więc poznać pewną popularną sztuczkę, którą wprowadzono do C w celu ominięcia reguły 4.1. Otóż wprowadzono specjalny typ wskaźników, `void*`, które charakteryzują się tym, że można im przypisać wartość wskaźnika dowolnego typu:

```
int x = 0
int* p = &x;     // p wskazuje na x
void* q = p;     // zapisujemy w q wartość p, czyli adres x
```

Ta karkołomna konstrukcja służy w C do przekazywania do funkcji argumentów dowolnego typu. W C++ ten sam cel, lecz w bardziej bezpieczny sposób, realizują szablony (por. s. 191).

Reguła 4.3. *Naucz się korzystać ze wskaźników typu* `void*` *używanych w cudzym kodzie, ale nigdy, przenigdy nie definiuj ich we własnych programach.*

4.1.3. Wskaźnik zerowy

W języku C++ przyjęto zasadę, że żaden obiekt nie może mieć adresu o wartości `0`. Z drugiej strony każdemu wskaźnikowi można przypisać tę wartość. Oznacza to, że zero jest wartością specjalną, która sygnalizuje jakąś sytuację wyjątkową. Najczęściej `0` przypisuje się wskaźnikowi, aby zasygnalizować, że w danym momencie nie wskazuje on żadnego obiektu.

W języku C zamiast wartości `0` stosuje się stałą symboliczną `NULL`.

4.1.4. Czym grozi nieumiejętne użycie wskaźników?

Wskaźniki to niezwykle niebezpieczne narzędzie, gdyż w źle napisanym programie łatwo mogą wymknąć się spod kontroli i wskazywać zupełnie coś innego, niż wydaje się autorowi programu. Oto kilka najczęściej spotykanych rodzajów błędów związanych z nieumiejętnym użyciem wskaźników:

- Wskaźnik wskazuje na inną zmienną, niż wydaje się to programiście.
- Wskaźnik wskazuje zmienną innego typu, niż wydaje się to programiście.
- Programista usiłuje zapisać dane poprzez wskaźnik zerowy.
- Wskaźnik wskazuje na przypadkowe miejsce w pamięci operacyjnej.
- Programista nie jest świadomy, że posługuje się wskaźnikiem.

- Wskaźnik wciąż wskazuje na miejsce w pamięci, które jakiś czas temu przestało należeć do programu.
- Programista zmienia wartość wskaźnika, który był jedynym dojściem do fragmentu pamięci operacyjnej przydzielonej programowi.

Wszystkie opisane tu kłopoty są naprawdę poważne: albo powodują pad programu, albo wyciek pamięci, albo wejście programu w niekontrolowaną turbulencję. Powtórzmy więc raz jeszcze:

Reguła 4.4. *Jeżeli naprawdę nie musisz, nie używaj wskaźników.*

4.1.5. Wskaźniki stałe i wskaźniki na stałe

Jeśli z jakichś względów nie można uniknąć użycia wskaźników, powinno się przynajmniej spróbować ograniczyć liczbę potencjalnych niespodzianek, które mogą nas spotkać z ich strony. Najłatwiejszy sposób osiągnięcia tego celu to zredukowanie liczby operacji, jakie można wykonać na wskaźnikach lub poprzez wskaźniki. Podstawowa metoda polega na użyciu słowa kluczowego `const`. Za jego pomocą można definiować wskaźniki stałe, wskaźniki na stałe i stałe wskaźniki na stałe, co ilustruje następujący przykład:

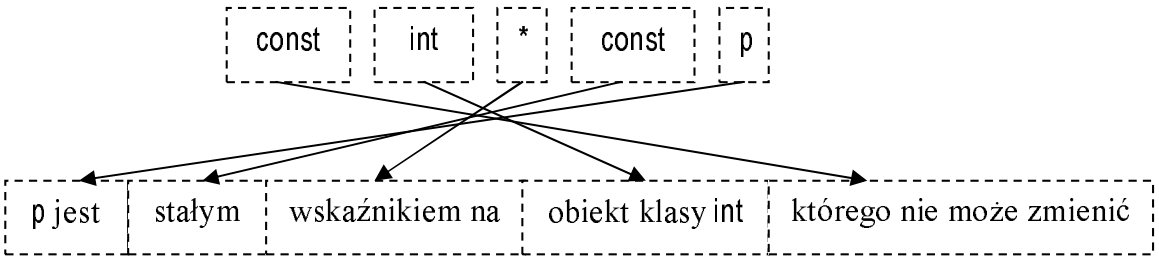
```
int x = 0;
int * const cp = &x;          // cp jest stałym wskaźnikiem na x
const int * pc = &x;          // pc jest wskaźnikiem na stałą
const int * const cpc = &x; // cpc jest stałym wskaźnikiem na stałą
```

Wskaźniki stałe to wskaźniki, które zawsze mają tę samą wartość (czyli zawsze wskazują ten sam obiekt). Ponadto wskaźniki stałe *muszą* być zainicjowane adresem odpowiedniego obiektu (lub zerem). Powyższe dwie właściwości eliminują większość możliwości błędnego użycia stałego wskaźnika. Pod względem funkcjonalnym stałe wskaźniki są niemal równoważne referencjom.

Wskaźniki na stałą to wskaźniki umożliwiające wyłącznie odczyt stanu wskazywanego przez siebie obiektu.

Stałe wskaźniki na stałą to połączenie obu powyższych rodzajów wskaźników. W sensie funkcjonalnym są niemal równoważne stałym referencjom.

Najłatwiejszy sposób radzenia sobie ze skomplikowanymi deklaracjami wskaźników polega na odczytywaniu ich od tyłu, co ilustruje rysunek 4.1.



Rysunek 4.1. Definicje wskaźników najłatwiej odczytuje się od tyłu

4.1.6. Wskaźniki na wskaźniki

Oczywiście obiekt wskazywany przez wskaźnik sam może być wskaźnikiem. Oto przykład:

```
int x = 0;
int* px = &x;
int** pp = &px;
```

W języku C wskaźniki na wskaźniki stosowane są m.in. jako argumenty funkcji (gdy funkcja ma zmienić wartość wskaźnika przekazanego w jej argumencie) lub do tworzenia tablic dwuwymiarowych (szczególnie takich, w których poszczególne wiersze mogą mieć różne długości).

4.2. Tablice

Tablice są niskopoziomową[1] wersją poznanych w rozdziale 2. wektorów, odziedziczoną jeszcze z języka C. Tablica to po prostu ciąg obiektów określonej klasy posiadający swoją nazwę i przechowywany w spójnym fragmencie pamięci operacyjnej. Liczba elementów tego ciągu zwana jest rozmiarem tablicy i *musi być znana już podczas kompilacji programu.*

Przykład zastosowania tablicy prezentuje wydruk 4.2. Zawiera on dwie funkcje: `fibo1` i `fibo3`, które stanowią rozwiązanie zadania 5 (s. 84).

Wydruk 4.2. Przykład użycia tablicy

```
4  int fibo1(int n)
5  {
       if (n <= 2)
           return 1;
       return fibo1(n-1) + fibo1(n-2);
   }
10
   const int fibo_max = 50;
   int fibo_tab[fibo_max] = {0};

   int fibo3(int n)
15 {
       if (n <= 2 or n >= ::fibo_max)
           return 1;

       if (::fibo_tab[n] == 0)
20         ::fibo_tab[n] = fibo3(n-1) + fibo3(n-2);
       return ::fibo_tab[n];
   }
```

Pierwsza z tych funkcji, `fibo1`, stanowi rozwiązanie czysto rekurencyjne. Wyróżnia się elegancją, jednak nie grzeszy efektywnością: wyznaczenie 46 kolejnych liczb Fibonacciego zajmuje jej na moim komputerze 2 minuty i 38 sekund. Druga z tych funkcji, `fibo3`, również korzysta z rekurencji, jednak wyznaczenie 46 kolejnych liczb Fibonacciego zajmuje jej zaledwie jedną milionową sekundy. Jak to jest możliwe?

[1] Tj. dostosowaną bardziej do możliwości komputera niż potrzeb człowieka.

Otóż funkcja `fibo3` stara się nie powtarzać raz przeprowadzonych obliczeń. Dlatego po pierwszym obliczeniu swojej wartości dla danego argumentu zapisuje odpowiadającą mu wartość w specjalnej tablicy. Przy każdym wywołaniu funkcja najpierw sprawdza w tablicy, czy nie zapisała już w niej poszukiwanej wartości. Jeśli tak, po prostu zwraca ją. Jeżeli nie – wykorzystuje rekurencję do znalezienia swojej wartości dla bieżącego argumentu, zapisuje ją w tablicy i dopiero wtedy zwraca jako swoją wartość.

Prześledźmy, jak powyższy scenariusz został zrealizowany na wydruku 4.2. W wierszu 10 zdefiniowano stałą symboliczną `fibo_max`, która określa rozmiar tablicy przechowującej wartości funkcji (por. reguła 2.7 na s. 44). Samą tablicę tworzy się w wierszu 12, stosując instrukcję o ogólnej postaci

```
TYP_ELEMENTU NAZWA_TABLICY [ROZMIAR_TABLICY];
```

Dodatkowo wraz ze zdefiniowaniem tablicy wszystkim jej elementom nadano wartość początkową `0` (ten element języka zostanie szczegółowo opisany w punkcie 4.2.2). Funkcja `fibo3` najpierw sprawdza, czy jej argument mieści się w rozmiarze tablicy. Jeśli nie – zwraca `1`. Jeśli argument jest poprawny, funkcja sprawdza, czy odpowiadająca mu liczba Fibonacciego nie została wcześniej zapisana w globalnej tablicy `fibo_tab`. Wykorzystano tu fakt, że żadna liczba Fibonacciego nie ma wartości `0`. Jak widać, dostęp do n-tego elementu tablicy zapewnia operator `[]`. Oznacza to, że nie ma żadnej różnicy w używaniu tablic i wektorów.

Reguła 4.5. *Pierwszy element tablicy n-elementowej ma indeks* `0`*, a ostatni ma indeks* `n-1`*. Użycie indeksu spoza zakresu* `0...n-1` *oznacza poważny błąd, który może doprowadzić nawet do padu programu.*

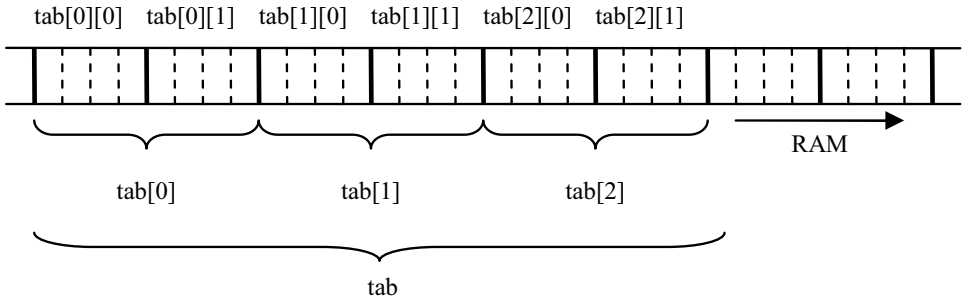
Nie ma chyba programisty, który choć raz na własnej skórze nie odczuł doniosłości powyższej reguły.

4.2.1. Tablice wielowymiarowe

Elementami tablic mogą być inne tablice. Oto prosty przykład:

```
int tab[3][2];      // definicja tablicy dwuwymiarowej 3 na 2
tab[1][1] = 100;    // dostęp do elementu 1,1
```

W pierwszym wierszu tego kodu definiuje się tablicę trzech obiektów, z których każdy jest tablicą dwóch liczb typu `int`. W sumie zdefiniowano więc dwuwymiarową tablicę 3×2 liczb typu `int`, co ilustruje rysunek 4.2. Z kolei w drugim wierszu elementowi $\texttt{tab}_{11}$ przypisano wartość `100`.



Rysunek 4.2. Sposób umieszczenia w pamięci operacyjnej tablicy `int tab[3][2]`

4.2.2. Inicjalizacja tablic

W wierszu 12 programu z wydruku 4.2 zastosowano specjalną konstrukcję służącą do nadawania wartości początkowej tablicy:

```
int fibo_tab[fibo_max] = {0};
```

Jak widzimy, inicjalizacja tablicy wygląda podobnie do inicjalizacji zwykłej zmiennej, tj. rozpoczyna się znakiem =. Główna różnica polega na tym, że tablicę można (a często nawet trzeba) inicjalizować kilkoma wartościami. Wartości te podaje się po znaku = jako ujętą w klamry listę, w której poszczególne pozycje oddzielone są od siebie przecinkiem.

Podczas tworzenia kodu inicjalizacji tablicy kompilator stosuje następującą regułę:

Reguła 4.6. *Jeżeli lista inicjalizacyjna tablicy zawiera mniej elementów niż rozmiar tablicy, to kompilator uzupełni ją zerami.*

To właśnie ta reguła (a nie wpisanie `0` w klamrach) gwarantuje, że wszystkie elementy tablicy `fibo_tab` będą początkowo miały wartość `0`.

W kontekście inicjalizacji warto jeszcze zapamiętać następującą regułę:

Reguła 4.7. *Wszystkie zmienne, obiekty i tablice globalne domyślnie inicjalizowane są zerami.*

Wynika stąd, że tak naprawdę inicjalizacja tablic globalnych zerami jest zbędna. Mimo to w programie z wydruku 4.2 zastosowałem jawną inicjalizację, ponieważ zwiększa to czytelność kodu – od razu informuje bowiem ewentualnego czytelnika, że inicjalizacja tablicy `fibo_tab` zerami jest istotnym składnikiem kodu.

Reguła 4.8. *Unikaj stosowania niejawnych inicjatorów.*

Oto kilka przykładów zastosowania inicjatorów tablicy:

```
int tab1[3] = {0};                  // 0, 0, 0
int tab2[3] = {1};                  // 1, 0, 0
int tab22a[2][2] = {0, 1, 2, 3};    // tab22a[1][1] == 3
int tab22b[2][2] = {{0, 1}, {2, 3}}; // to samo co powyżej
double tab4[ ] = {1.0, 10.0, 100.0};
```

Na szczególną uwagę zasługują puste nawiasy kwadratowe w ostatniej z powyższych instrukcji. Jeżeli wraz z definicją tablicy podajemy jej listę inicjalizacyjną, w definicji tablicy możemy pominąć jej rozmiar. W tym przypadku kompilator sam „dopisze” wymaganą wartość na podstawie liczby elementów listy inicjalizacyjnej.

4.2.3. Zastosowanie operatora `sizeof` do tablic

Jak już wiemy, operator `sizeof` zwraca liczbę bajtów zajmowanych przez dany obiekt w pamięci operacyjnej. Jeżeli jako argumentu tego operatora użyjemy nazwy tablicy, zwróci on liczbę bajtów zajmowanych przez tablicę. Właściwość ta ma zastosowanie do wyznaczania rozmiaru tablicy w przypadku, gdy jej rozmiar zdefiniowano niejawnie poprzez użycie inicjatora tablicy:

```
double tab4[] = {1.0, 10.0, 100.0};
const int rozmiar_tablicy = sizeof(tab4)/sizeof(tab4[0]);
```

W powyższym kodzie zmienna `rozmiar_tablicy` będzie miała poprawną wartość nawet po zmianie liczby elementów tablicy lub ich typu.

4.2.4. Tablice a wskaźniki

W językach C i C++ obowiązuje następująca reguła:

Reguła 4.9. *Nazwa tablicy może być traktowana jak wskaźnik na jej pierwszy element.*

Własność tę demonstruje wydruk 4.3.

Wydruk 4.3. Odpowiedniość między nazwą tablicy a wskaźnikiem na jej pierwszy element

```
3 int main()
  {
5     int tab[4] = {1, 2, 3, 4};
      std::cout << tab << "\t" << &tab[0] << "\n";
      int* p = tab;
      std::cout << p << "\t" << *p << "\t" << p[1] << "\n";
      std::cout << sizeof(p) << "\t" << sizeof(tab) << "\n";
10 }
```

```
0x22ff60    0x22ff60
0x22ff60    1    2
4    16
```

W wierszu 5 zdefiniowano tablicę `tab` i nadano jej kolejnym elementom wartości początkowe `1`, `2`, `3` i `4`. W wierszu 6 „wysłano" nazwę tablicy na standardowe wyjście, w wyniku czego na ekranie pojawił się napis `0x22ff60`. Notacja szesnastkowa (por. s. 40) oznacza, że na ekranie faktycznie wyświetlono wartość adresu (a nie np. kolejnych elementów tablicy). W tej samej instrukcji operatorem `&` pobrano adres pierwszego elementu tablicy i również wyświetlono go na ekranie. Wynik potwierdza regułę 4.9.

W wierszu 7 adres pierwszego elementu tablicy przypisano wskaźnikowi `p`. Wiersz 8 dowodzi, że wskaźnika tego można używać tak samo jak nazwy tablicy (np. stosować do niego operator indeksowania).

Spostrzeżenie 4.1. *Wskaźniki różnią się od nazw tablic sposobem definiowania, nie różnią się zaś sposobem używania. Wskaźników można używać jak nazw tablic, a nazw tablic jak wskaźniki.*

Wiersz 9 dowodzi, że kompilator odróżnia jednak wskaźniki od nazw tablic. Nazwa tablicy zawsze wskazuje na to samo miejsce, w którym na pewno znajduje się tablica, dlatego `sizeof(tab)` zwraca liczbę bajtów zajmowanych przez tablicę. Wskaźnik może wskazywać na cokolwiek, dlatego `sizeof(p)` zwraca liczbę bajtów zajmowanych przez wskaźniów.

4.2.5. Tablice wskaźników i wskaźniki na tablice

W języku C++ można definiować zarówno tablice wskaźników, jak i wskaźniki na tablice. Składnia definicji takich obiektów wydaje się na pierwszy rzut oka bardzo skomplikowana, ale można ją ogarnąć dzięki następującej regule, stanowiącej uogólnienie zasady podanej na s. 88:

Reguła 4.10. *Odczytywanie definicji wskaźnika rozpoczynamy od nazwy zmiennej, po czym odczytujemy definicję w prawo do napotkania prawego nawiasu okrągłego lub końca definicji; następnie kontynuujemy odczytywanie deklaracji w kierunku przeciwnym, poczynając od nazwy zmiennej, aż do napotkania otwierającego nawiasu okrągłego lub początku definicji. Procedurę tę powtarzamy rekurencyjnie do wyczerpania wszystkich nawiasów.*

Oto kilka przykładów użycia tej reguły:

```
int const* p[5];
int (*q) [5];
char const *(* const (*pp)[])[10];
void (*f)(double&);
```

Pierwszą z powyższych instrukcji odczytujemy następująco (strzałki informują o kierunku odczytywania definicji): „p → jest tablicą pięciu ← wskaźników na stałe obiekty typu int". Drugą odczytujemy tak: „q ← jest wskaźnikiem → na tablicę pięciu ← liczb typu int". Trzecia deklaracja jest nieco trudniejsza: „pp jest ← wskaźnikiem → na tablicę ← stałych wskaźników → na tablicę dziesięciu ← wskaźników na stałe znaki". Ostatnią definicję można odczytać następująco: „f jest ← wskaźnikiem → na funkcję przyjmującą jeden argument typu double przez referencję ← i nie zwracającą żadnej wartości".

4.2.6. Tablice jako argumenty funkcji

Tablice nie są do funkcji przekazywane ani przez wartość (byłaby to karygodna rozrzutność), ani przez referencję (bo mechanizmu tego nie ma w C), lecz przez wskaźnik. Przykład przekazania tablicy do funkcji przedstawia wydruk 4.4. Zawiera on m.in. definicję funkcji suma, której zadaniem jest obliczenie sumy elementów tablicy. Funkcja ta jest więc odpowiednikiem funkcji obliczającej sumę elementów wektora (por. wydruk 3.6 na s. 69).

Wydruk 4.4. Przekazywanie tablic do funkcji

```
3  double suma(double tab[], int rozmiar_tablicy)
   {
5      double wynik = 0.0;
       for (int i = 0; i < rozmiar_tablicy; ++i)
           wynik += tab[i];
       return wynik;
   }
10
   int main()
   {
       const int rozmiar = 5000;
       double v[rozmiar];
15     for (int i = 0; i < rozmiar; ++i)
           v[i] = static_cast<double>(i) / rozmiar;

       std::cout << suma(v, rozmiar) << "\n";
   }
```

W wierszu 3 rozpoczyna się definicja funkcji `suma`. Ponieważ tablice są obiektami niskopoziomowymi, nie zawierają informacji o swoim rozmiarze. Dlatego funkcja `suma` musi pobrać dwa argumenty: nazwę tablicy (czyli adres jej pierwszego elementu) i jej rozmiar. W deklaracjach argumentów tablicowych funkcji z zasady nie podaje się ich rozmiaru, lecz używa się pustej pary nawiasów kwadratowych. Faktycznymi argumentami tych funkcji mogą być tablice o dowolnym rozmiarze.

Funkcję `suma` wywołuje się w wierszu 18. Oczywiście jako argumentu funkcji używa się nazwy tablicy (czyli adresu jej pierwszego elementu). Dlatego wiersz 18 równie dobrze mógłby być zapisany następująco:

```
std:: cout << suma(&v[0], rozmiar) << "\n";
```

Analogicznie prototyp funkcji `suma` mógłby wyglądać następująco:

```
double suma(double *tab, int rozmiar_tablicy)
```

Byłoby to jednak dość mylące – użycie „zwykłego" wskaźnika sugerowałoby, że funkcja przyjmuje przez wskaźnik tylko jeden obiekt, a nie całą tablicę.

Jak widać, przy przekazywaniu tablic do funkcji w pełni wykorzystuje się odpowiedniość między nawami tablic a wskaźnikami na ich pierwszy element.

4.2.7. Teksty literalne i tablice znaków

Mimo że niejednokrotnie używałem już w tym podręczniku literałów tekstowych (czyli ciągów znaków ujętych w cudzysłów, np. `"koniec programu\n"`), to aż dotąd unikałem odpowiedzi na pytanie, jaki jest ich typ. Okazuje się, że są to *stałe tablice znaków* zakończone specjalnym znakiem `'\0'` o wartości `0`. Np. typem literału tekstowego `"koniec"` jest `const char[7]`, a składa się on z elementów `'k'`, `'o'`, `'n'`, `'i'`, `'e'`, `'c'`, `'\0'`. Dlatego też wartością wyrażenia `sizeof("koniec")` jest `7`. Znak `'\0'`, zwany też *bajtem zerowym*, dopisywany jest automatycznie na końcu tablicy przez kompilator.

Literały tekstowe mogą służyć do uproszczonej inicjalizacji tablic znaków. Np. zamiast

```
char tab[] = {'A', 'l', 'a', ' ', 'm', 'a', ' ', 'k', 'o', 't', 'a', '\0'};
```

można napisać po prostu

```
char tab[] = "Ala ma kota";
```

Podręcznikowy przykład zastosowania tablic znaków w stylu języka C przedstawia wydruk 4.5. Znajdująca się w nim funkcja `kopiuj` służy do kopiowania tekstu przekazywanego w drugim argumencie do obszaru wskazywanego przez pierwszy argument. Kopiowanie kontynuowane jest tak długo, aż zostanie skopiowany bajt zerowy.

```
tekst 'Ala ma kota' ma dlugosc 11
```

Mimo swojej pozornej elegancji (związanej z wyjątkowo zwięzłym zapisem) funkcja ta obnaża dwa poważne problemy związane ze stosowaniem tekstów w stylu C. Po pierwsze, funkcja `kopiuj` nie ma pojęcia, czy i ile miejsca zarezerwowano pod adresem `dokad`. Funkcja nie ma też żadnych informacji na temat rzeczywistej długości kopiowanej tablicy znaków. Obie te właściwości są potencjalnym źródłem bardzo poważnych błędów. Zaprawdę korzystanie z tekstów w stylu C wymaga od programisty dużej wprawy i samodyscypliny.

Wydruk 4.5. Przykład funkcji operującej na tablicach tekstowych w stylu C

```
4 void kopiuj(char* dokad, const char* skad)
5 {
      while(*dokad++ = *skad++)
          continue;
  }

10 int main()
  {
      char bufor[200];
      kopiuj (bufor, "Ala ma kota");
      std::cout << "tekst '" << bufor << "' ma dlugosc " << strlen(bufor) << "\n";
15 }
```

Reguła 4.11. *W miarę możliwości należy posługiwać się napisami w stylu języka C++, czyli obiektami typu* `std::string`*; z języka C warto zapożyczyć wyłącznie literały tekstowe.*

4.2.8. Porównanie tablic i wektorów

Porównanie tablic (w stylu języka C) z wektorami (w stylu języka C++) zdecydowanie wypada na korzyść tych drugich. Jedyną zaletą tablic jest to, że w funkcjach traktowane są jak zwykłe zmienne lokalne, w związku z czym pamięć dla nich rezerwowana jest już w czasie kompilacji programu. Innymi słowy, samo utworzenie tablicy nie kosztuje programu dosłownie nic (poza pamięcią operacyjną). Niestety, wektory biblioteki C++ tworzone są dopiero podczas wykonywania programu, dlatego samo ich utworzenie jest stosunkowo kosztowne. Wynika stąd, że wewnątrz intensywnie wykorzystywanej pętli `for` lepiej jest definiować tablice niż wektory. Można jednak ominąć ten problem, definiując wektor przed pętlą.

A oto główne zalety wektorów w stosunku do tablic:

- Rozmiar wektorów nie musi być znany podczas kompilacji.
- Wektory przechowują informację o swoim rozmiarze.
- Wektory mogą zmieniać swój rozmiar podczas wykonywania się programu.
- Dostęp do dowolnego elementu wektora jest równie szybki, jak dostęp do elementu tablicy.
- Dane przechowywane w wektorach nie obciążają obszaru stosu funkcji – dlatego nawet lokalne wektory mogą mieć gigantyczne rozmiary.
- Bez wprowadzania dodatkowego kodu do programu można spowodować, by podczas wykonania programu była sprawdzana poprawność użycia każdego indeksu wektora (por. spostrzeżenie 11.8 na s. 220).

Reguła 4.12. *Tablic w stylu języka C używaj tylko wtedy, gdy naprawdę musisz. Niemal zawsze lepszym pomysłem jest użycie wektorów C++.*

4.3. Pamięć wolna (sterta)

Wszystkie obiekty, z którymi mieliśmy dotąd do czynienia, charakteryzowały się tym, że programista wybierał dla nich nazwę, ale miejsce w pamięci operacyjnej rezerwował dla nich kompilator w jednym z dwóch bloków pamięci – albo w obszarze zmiennych globalnych, albo na stosie funkcji. W C++ istnieje też metoda tworzenia obiektów „na żądanie", tj. tylko w określonych sytuacjach, których wystąpienia nie można przewidzieć podczas kompilacji programu. Obiekty tworzone na żądanie zwane są *obiektami dynamicznymi*, blok pamięci, w którym przechowywane są takie obiekty, zwany jest *pamięcią wolną* lub *stertą*, a proces tworzenia obiektów dynamicznych na stercie zwany jest *dynamiczną alokacją pamięci*.

Spostrzeżenie 4.2. *Słowo „dynamiczny" oznacza „taki, którego pełna charakterystyka ustalana jest dopiero podczas wykonywania się programu"*[2].

Podstawowe sposoby tworzenia obiektów dynamicznych ilustruje wydruk 4.6:

Wydruk 4.6. Tworzenie obiektów dynamicznych

```
5   int* p = new int;
    *p = 7;
    double* q = new double (3.14);
    std::cout << "*p = " << *p << ", *q = " << *q << "\n";
    delete p;
10  delete q;
```

```
*p = 7, *q = 3.14
```

W wierszu 5 za pomocą operatora `new` tworzona jest na stercie nowa zmienna typu `int`. Zmienna ta *nie ma swojej nazwy*, a dostęp do niej możliwy jest wyłącznie poprzez wskaźniki. W szczególności operator `new` zwraca jako swoją wartość właśnie adres zarezerwowany w pamięci operacyjnej dla nowo utworzonego obiektu. Na wydruku 4.6 adres ten jest natychmiast zapisywany w zmiennej wskaźnikowej `p`.

Spostrzeżenie 4.3. *Obliczenie wyrażenia*

new NAZWA_KLASY

powoduje zarezerwowanie na stercie pamięci dla obiektu klasy `NAZWA_KLASY`*, a wartością tego wyrażenia jest adres tego obszaru pamięci.*

W wierszu 6 nowo utworzonej zmiennej dynamicznej przypisywana jest wartość `7`. Oczywiście czyni się to za pośrednictwem zmiennej wskaźnikowej i operatora wyłuskania.

Ustalenie wartości początkowej obiektu powinno być – w miarę możliwości – wykonywane już w momencie jego tworzenia. Niestety, metoda użyta w wierszach 5 i 6 nie spełnia tego warunku. Na szczęście C++ umożliwia jednoczesne utworzenie obiektu dynamicznego i nadanie mu wartości początkowej – w tym celu wartość początkową umieszcza się w nawiasach okrągłych za nazwą typu. Metodę tę ilustruje wiersz 7.

[2] Antonimem słowa „dynamiczny" jest (w tym kontekście) słowo „*statyczny*". Np. „dynamiczna kontrola typów" to kontrola poprawności użycia argumentów funkcji wykonywana podczas wykonywania programu; „statyczna kontrola typów" wykonywana jest przez kompilator w fazie translacji kodu źródłowego na kod maszynowy, czyli przed uruchomieniem programu.

W przypadku używania obiektów dynamicznych programista zobowiązany jest nie tylko do ich jawnego tworzenia, ale i do ich jawnego zniszczenia. Do zwalniania pamięci służy specjalny operator `delete`. Przykład jego użycia demonstrują wiersze 8 i 9. Po zwolnieniu pamięci operatorem `delete` nie wolno już się do niej odwoływać.

Reguła 4.13. *Obiekty utworzone dynamicznie na stercie powinny być z niej jawnie usunięte operatorem* `delete`.

Tworzenie dynamicznych obiektów reprezentujących pojedyncze obiekty typu `int` czy `double` nie jest specjalnie interesujące. Jednym z podstawowych zastosowań operatora `new` jest tworzenie dynamicznych tablic. Przykład przedstawia wydruk 4.7.

Wydruk 4.7. Tworzenie i usuwanie tablic dynamicznych

```
5     std::cout << "podaj rozmiar tablicy: ";
      int n;
      std::cin >> n;
      int * const p = new int [n];
      for (int i = 0; i < n; ++i)
10        p[i] = i;
// ...
      delete [] p;
```

W wierszach 5–7 program wczytuje z klawiatury dowolną liczbę `n` klasy `int`. Następnie w wierszu 8 tworzy na stercie tablicę `n` liczb typu `int`, a jej adres zapisuje w zmiennej wskaźnikowej `p`. Rozmiar tablicy podawany jest w nawiasach kwadratowych po nazwie klasy i nie musi być znany podczas kompilacji.

Spostrzeżenie 4.4. *Obliczenie wyrażenia*

new NAZWA_KLASY [ROZMIAR]

powoduje zarezerwowanie na stercie bloku pamięci dla tablicy zawierającej `ROZMIAR` *obiektów klasy* `NAZWA_KLAY`*, a wartością wyrażenia jest adres tego obszaru pamięci.*

Zgodnie ze spostrzeżeniem 4.1 (s. 92) wskaźnik przechowujący adres początku tablicy można traktować jak jej nazwę – wykorzystano to w wierszu 10.

Wiersz 12 ukazuje sposób usunięcia niepotrzebnej już tablicy ze sterty. Zwolniona w ten sposób pamięć może być następnie wykorzystywana w innych fragmentach programu lub przez inne programy.

Posługiwanie się pamięcią wolną to skomplikowany temat godzien osobnego opracowania. Oto kilka dodatkowych wskazówek ułatwiających to zadanie.

- Operator `new` i operator `new []` to dwa różne operatory. Podobnie `delete` i `delete []` to dwa różne operatory. Pamięć pobraną operatorem `new` należy zwalniać wyłącznie operatorem `delete`, a pamięć pobraną operatorem `new []` należy zwalniać operatorem `delete []`.
- Argumentem operatorów `delete` może być wyłącznie adres zwrócony wcześniej przez odpowiedni operator `new`.
- Argumentem operatorów `delete` nie może być adres obszaru pamięci zwolnionego już operatorem `delete`.

- Argumentem operatorów `delete` może być adres zerowy (nie powoduje to żadnych konsekwencji).
- Może się zdarzyć, że operator `new` nie będzie mógł znaleźć miejsca na stercie. W tym przypadku w starszych implementacjach C++ operator ten zwracał adres zerowy; w nowych implementacjach operator `new` zgłasza wyjątek `std::bad_alloc` (wyjątki zostaną omówione w rozdziale 13.).
- Podczas alokacji tablic obiektów nie ma możliwości przekazania im inicjatorów (tj. tablice dynamiczne wymagają osobnej inicjalizacji).

Najważniejsza jest jednak następująca reguła:

Reguła 4.14. *Bezpośrednie korzystanie z pamięci wolnej jest trudniejsze, niż się to wydaje, i rzadko kiedy naprawdę potrzebne. Zanim użyje się operatorów* `new` *i* `delete`*, warto się zastanowić, czy tego samego efektu nie osiągnie się za pomocą klas biblioteki standardowej, np.* `std::vector<T>` *czy* `std::list<T>`*.*

4.4. Q & A

Czy to prawda, że tablice bezpośrednio tworzone operatorem `new` działają szybciej niż standardowe wektory `std::vector`?
Jeżeli kompilator pracuje w trybie optymalizacji prędkości, to nie ma żadnej różnicy. Tak naprawdę klasa `std::vector` to bardzo efektywna, wysokopoziomowa „obudowa" tablic tworzonych operatorem `new`.
Czy to prawda, że wskaźniki są szybsze od operatora indeksowania?
Nieprawda. Zamiana kodu źródłowego wykorzystującego tablice na kod źródłowy wykorzystujący wskaźniki z reguły prowadzi do jego zaciemnienia, natomiast uzyskany kod maszynowy jest praktycznie taki sam.
Jaka jest różnica między `int const* p` i `const int * p`?
Żadna. Pierwsza z tych form lepiej odpowiada regule 4.10, jednak bardziej popularna jest druga z nich.
Zupełnie nie rozumiem programu z wydruku 4.5!
To normalne – przecież ten program napisano nie w C++, tylko w C! Nieco poważniej: proszę spróbować rozwiązać problem 1.

4.5. Pytania przeglądowe

1. Co to są wskaźniki?
2. Do czego służy operator wyłuskania?
3. W jaki sposób można określić adres, pod którym znajduje się dany obiekt?
4. Jaka jest różnica między wskaźnikami na stałą a wskaźnikami stałymi?
5. Jaki jest związek między nazwami tablic w stylu języka C a wskaźnikami?
6. Dlaczego należy unikać stosowania wskaźników?
7. W jaki sposób definiuje się wartości początkowe elementów tablic?
8. Jaki jest związek tekstów literalnych (np. `"Ala ma kota"`) z tablicami?
9. Co to jest bajt zerowy?
10. Jakie są zalety wektorów C++ w stosunku do tablic w stylu języka C?

11. Jakie są zalety tablic w stylu C w stosunku do wektorów C++?
12. Co to jest pamięć wolna?
13. Co w języku C++ oznacza przymiotnik „dynamiczny”?
14. Do czego służy i co zwraca operator `new`?
15. Do czego służy operator `delete`?
16. Jaka jest różnica między `delete` i `delete []`?
17. Jaki jest efekt wykonania instrukcji `delete 0;`?

4.6. Problemy

1. Spójrz na wydruk 4.5 i odpowiedz na następujące pytania:
   - Który z operatorów użytych w wyrażeniu warunkowym pętli `while` ma najwyższy, a który najniższy priorytet.
   - Jak działa i jaką wartość zwraca przyrostkowy operator `++`?
   - Do czego służy jednoargumentowy operator `*`?
   - Jaką wartość zwraca operator przypisania?
   - Jak interpretowane jest wyrażenie warunkowe w pętli `while`, jeżeli zamiast wyrażenia logicznego (o wartości `true` lub `false`) pojawia się w nim wyrażenie arytmetyczne?
2. Zadeklaruj:
   - Wskaźnik na tablicę stu liczb typu `double`.
   - Dwuwymiarową tablicę 5×5 wskaźników na `char`.
   - Tablicę (o nieokreślonej długości) wskaźników na `char`.
   - Standardowy wektor wskaźników `void*`.
   - Czteroelementową tablicę wektorów o elementach typu `double`.
   - Wskaźnik na funkcję zwracającą jako wartość liczbę typu `double` i przyjmującą dwa argumenty: tablicę (w stylu C) liczb typu `double` oraz liczbę typu `int`.
   - Dowolną funkcję przyjmującą przez referencję wskaźnik na `int`.
3. Znajdź takie `N`, dla którego wykonanie pętli

```
for (unsigned n = 1; n <= N; ++n)
{
    int* p = new int[n];
    delete [] p;
}
```

   trwa około jednej sekundy. Na tej podstawie porównaj koszt (=czas) wykonania pojedynczej pary instrukcji `new` / `delete` z kosztem pojedynczego dodawania. (Wskazówka: koszt dodawania można oszacować podobnie w innej pętli `for`).
4. Program

```
int main()
{
    double tab[10];
    double x;
    std:: cout << &x << "\t" << &tab[-1] << "\n";
}
```

produkuje w *moim* komputerze następujący wynik:

```
0x22ff18      0x22ff18
```

Skomentuj i wyjaśnij to zjawisko. Czy występuje ono także w Twoim komputerze? Jakie konsekwencje może mieć niepoprawne indeksowanie tablic?

5. Wyjaśnij, w jaki sposób w poniższym programie wykorzystano indolencję programisty do włamania się do systemu?

```
int main()
{
    char haslo[8];   // tu będzie przechowywane hasło
    strcpy(haslo, "Ta.jnE!"); // kopiuje drugi argument w miejsce pierwszego
    for ( ; ; )
    {
        std::cout << "podaj haslo: ";
        char tmp[8];     // zmienna tymczasowa na wczytanie hasła
        std::cin >> tmp;
        if (strcmp(tmp, haslo) == 0) // czy teksty są takie same?
            break;
        std::cout << "przykro mi, haslo jest niepoprawne!\n";
    }
    std::cout << "witaj w systemie!\n";
}
```

```
podaj haslo: 123456781234567
przykro mi, haslo jest niepoprawne!
podaj haslo: 1234567
witaj w systemie!
```

6. Działanie programów z zadań 4 i 5 zależy od użytego kompilatora i platformy, na której są uruchamiane. Wyjaśnij, dlaczego. Jeżeli w Twoim komputerze dają one inne wyniki niż w moim, zmodyfikuj je tak, by osiągnąć ten sam efekt.