Wskaźniki

wskaźniki / adresy

Wskaźnik → jest zmienną, która zawiera adres (wskazanie) początku dowolnego obszaru w pamięci komputera,

(np. może być to adres obszaru danych lub adres kodu programu)

Ogólna postać definicji wskaźnika:

typ_danych ∗ identyfikator wskaźnika ;

Najczęściej używane są wskaźniki „zdefiniowane” zawierające adres innej zmiennej. Taki wskaźnik zawiera informację o:

adresie zmiennej w pamięci komputera

typie danych przechowywanych w tej zmiennej

Przykłady definicji:

int ∗ wskaznik; // wskaźnik na zmienną całkowitą

double ∗ wsk_liczby; // wskaźnik na zmienną rzeczywistą

char ∗ wsk_znaku; // wskaźnik na pojedynczy znak

char ∗ tekst; // wskaźnik na początek łańcucha znaków

(na pierwszy znak tego łańcucha)

Można również korzystać ze wskaźników „niezdefiniowanych” (anonimowych).

Taki wskaźnik zawiera tylko informację o adresie obszaru pamięci (bez określenia typu wskazywanych danych). Definicja takiego wskaźnika ma postać:

void ∗ identyfikator wskaźnika ;

jest to wskaźnik na „dowolny” ciąg bajtów danych.

Ze wskaźnikami i adresami związane są dwa operatory:

operator referencji & zwracający adres zmiennej podanej po prawej stronie tego operatora.

operator dereferencji ∗ identyfikujący obszar wskazywany przez wskaźnik podany po prawej stronie tego operatora.

Przykład 1.

Zmienna wskazująca i wskazywana

Napisac program prezentujący, dla zmiennej typu calkowitego, zaleznosc między zmienną wskazującą a wskazywaną.

```
#include <stdio.h>
#include <conio.h>
void main ()
{
int n=100, *wsk_n=&n, k;
clrscr ();
printf ("Nasza liczba to %d\n",n);
printf (”A wskaznik do niej to %d\n", *wsk_n);
printf (”Podaj nowa liczbe:");
scanf ("%d",&n);
printf ("\nTeraz liczba wynosi %d\n",n);
printf ("A wskaznik do niej to %d\n",*wsk_n);
*wsk_n=222;
printf ("A teraz podstawilismy za wskaznik %d\n", *wsk_n);
printf ("i mamy liczbe %d\n",n);
printf ("Podaj nowa liczbe:");
scanf ("%d", &k) ;
*wsk_n=k,
printf ("A teraz podstawiamy inna liczbe za wskaznik i"
"mamy liczbe %d\n" ,n);
printf ("A wskaznik do niej wynosi %d\n", *wsk_n);
getch ();
}
```

W tym programie tak naprawdę chodzi o prezentację zaleznosci między zmienną wskazującą a wskazywaną. Jest to tutaj nowinka programistyczna i nalezy sohie dokladnie wyjasnic, co to w ogóle jest. Dotąd mielismy po prostu zmienną i to nam wystarczalo.

Terazniejszy nowy sposób postrzegania nam się przyda przy zmiennych tablicowych, ale o tym w następnym przykladzie. Tutaj widzimy zmienną n, ktora jest dokladnie taka, jak zmienne uzywane do tej pory, oraz zmiennq *wsk_n (w dodatku z gwiazdką *) pod którą w procesie inicjacji podstawiamy &n (czyli adres zmiennej n). Otóż to jest wlasnie ta magiczna zmienna wskazująca, potocznie zwana wskaźnikiem. Fizycznie zmienna wsk_n jest adresem do naszej zmiennej, czyli informacją o tym, gdzie w pamięci komputera ona jest zapisana. Natomiast *wsk_n jest już odwołaniem się do tego adresu, czyli wlasciwie mozemy go prawie że utożsamiać ze zmienną -i to wlasnie prezentuje ten program. Jak widzimy, & jest równoważna adresowi zmiennej.

Dzialanie tego programu jest następujące:

Najpierw inicjujemy zmienną n o wartości 100, oraz odpowiadający jej wskaźnik *wsk_n, pod który podstawiamy adres naszej zmiennej, oraz zmienną k, która będzie nam potrzebna później.

Następnie drukujemy na ekranie wartość zmiennej n oraz wartość zmiennej wskazującej *wsk_n, podczas uruchomienia programu widać, że są rowne. Następnie wczytujemy z klawiatury nową wartość zmiennej n, i znów wyświetlamy na ekran zmienną i wskaźnik do niej, po czym podstawiamy programowo pod zmienną wskazującą 222, i powtórnie drukujemy zmienną i wskaźnik do niej. Ostatnią tego typu operacją jest wczytanie z klawiatury wartości innej zmiennej k, podstawienie jej pod wskaźnik, a następnie wydrukowanie na ekranie wartości zmiennej n i wskaźnika. Przy uruchomieniu programu za każdym razem wartość zmiennej wskazującej i wskazywanej okaże się równa -pozwala nam to uwierzyć, ze wszystko jedno, którą z nich się posługujemy.

Rzecz jasna, dla zmiennych innych typów prostych ten program dalby identyczne wyniki - można dla przykładu zamienic int na float.

Przykład 2.

Wczytanie elementów tablicy i wypisanie w odwrotnej kolejności -równoważność miedzy tablicą a wskaźnikiem dla tablic jednowymiarowych.

Napisac program wczytujący 10-elemenlową tablicę liczb całkowitych z klawiatury i wypisujący ją na ekran w odwrotnej kolejności.

```
#include <conio.h>

#include <stdio.h>
```

```
void main()
(
int tab[10], a, i;
clrscr();
printf("Podawaj elementy tablicy\n");
for (i=0;i<10;i++)
{
printf("Podaj %d element tablicy: ",i);
scanf ("%d", &a) ;
*(tab+i)=a;
}
printf("\nElementy w odwrotnej kolejnosci:\n");
for(i=9; i>=0; i--)
printf("tab[%d]=%d\n", i ,*(tab+i));
getch();
}
```

W tym programie prezentujemy, w jaki sposób korzystać ze wskaźnika do tablicy. Język C utożsamia nazwę tablicy ze wskaznikiem na nią, tak więc *tab będzie oznaczać odwołanie się do pierwszego elementu tablicy (tak samo jak tab [0]), zaś *(tab+i) jest odwołaniem się do elementu o numerze i (rownoważnie do tab [i]).

Program działa w ten sposób, że pobieramy 10 elementow tablicy wykorzystując pętlę for, której Iicznik i dodawany jest do wskaźnika na tablicę, po czym, przy pomocy takiej samej pętli (tyle że ze zmniejszającym się licznikiem), wypisujemy elementy tablicy na ekran, znowu wykorzystując wskaźnik na tablicę a nie bezpośrednie odwołanie do jej pola.

Analizując działanie programu krok po kroku:

Najpierw definiujemy zmienną tablicową 10-elementową tab oraz dwie zmienne typu int; i będzie licznikiem pętli, zaś a to zmienna pomocnicza do wczytywania liczb do tablicy. Po wyczyszczeniu ekranu i informacji dla użytkownika rozpoczyna działanie pierwsza pętla for, zwiększająca licznik i od 0 do 9, w każdym obrocie pętli pytamy o wartość elementu, wczytujemy go do zmiennej a, zaś tę wartość wstawiamy do tablicy, czyli w miejsce

wskazane przez * (tab+i). Teraz poinformujemy o tym, co za chwilę pokaże się na ekranie, po czym zaczyna działanie druga pętla for. Teraz licznik i zmniejsza się od 9 do 0, czyli przeglądamy tablicę "od końca", za każdym razem wypisując na ekranie numer elementu i jego wartość. Program kończy funkcja getch( ) , czekająca na klawisz.

Wskaźniki i pamięć

Program napisany w języku C po skompilowaniu wykorzystuje trzy rodzaje pamięci:

Statyczna/globalna

W tym rodzaju pamięci alokowane są zmienne deklarowane statycznie. Zmienne globalne także korzystają z tej pamięci. Są one alokowane w pamięci od chwili rozpoczęcia działania programu aż do jego zamknięcia. Wszystkie funkcje mają dostęp do zmiennych globalnych. Zasięg zmiennych statycznych jest ograniczony do funkcji, w której zostały one zdefiniowane.

Automatyczna

Zmienne te są deklarowane wewnątrz funkcji, a więc są tworzone w chwili wywołania funkcji. Ich zasięg jest ograniczony do funkcji. Okres istnienia tych zmiennych jest ograniczony do czasu wykonywania funkcji.

Dynamiczna

Pamięć jest alokowana na stercie i może zostać zwolniona, gdy będzie to konieczne. Wskaźnik odnosi się do alokowanej pamięci. Zasięg zmiennych jest ograniczony przez wskaźniki odnoszące się do tej pamięci. Pamięć ta istnieje do momentu jej zwolnienia.

W tabeli podsumowane są wiadomości dotyczące zasięgu i okresu istnienia zmiennych stosowanych w poszczególnych obszarach pamięci.

	Zasięg	Okres istnienia
Globalna	cały plik	cały okres działania aplikacji
Statyczna	funkcja, w której została zadeklarowana	cały okres działania aplikacji
Automatyczna (lokalna)	funkcja, w której została zadeklarowana	czas wykonywania funkcji
Dynamiczna	określony przez wskaźniki odnoszące się do tej pamięci	do momentu zwolnienia pamięci

Bliższe zapoznanie się z tymi rodzajami pamięci pozwoli Ci na lepsze zrozumienie funkcjonowania wskaźników. W większości przypadków wskaźniki są stosowane do wykonywania operacji na danych przechowywanych w pamięci. Zinterpretowanie tego, jak pamięć jest partycjonowana i organizowana, pomoże w wyjaśnieniu działań wykonywanych na pamięci przez wskaźniki.

Zmienna będąca wskaźnikiem zawiera adres pamięci, pod którym znajduje się inna zmienna, obiekt lub funkcja. Obiekt jest alokowany w pamięci za pomocą funkcji alokującej, takiej jak np. funkcja malloc. Zwykle deklaruje się typ wskaźnika, który zależy od tego, na co dany wskaźnik wskazuje.

Np. możemy zadeklarować wskaźnik na obiekt typu char. Obiektem może być liczba całkowita, znak, łańcuch, struktura lub dowolny inny typ danych spotykany w języku C. Wskaźnik nie zawiera niczego, co by informowało o tym, na jaki typ danych wskazuje. Wskaźnik zawiera tylko adres danych.

Dlaczego warto opanować wskaźniki

Wskaźniki można stosować do:

 tworzenia szybkiego i wydajnego kodu,

 rozwiązywania w prosty sposób różnego typu problemów,

 obsługi dynamicznej alokacji pamięci,

 tworzenia zwięzłych wyrażeń,

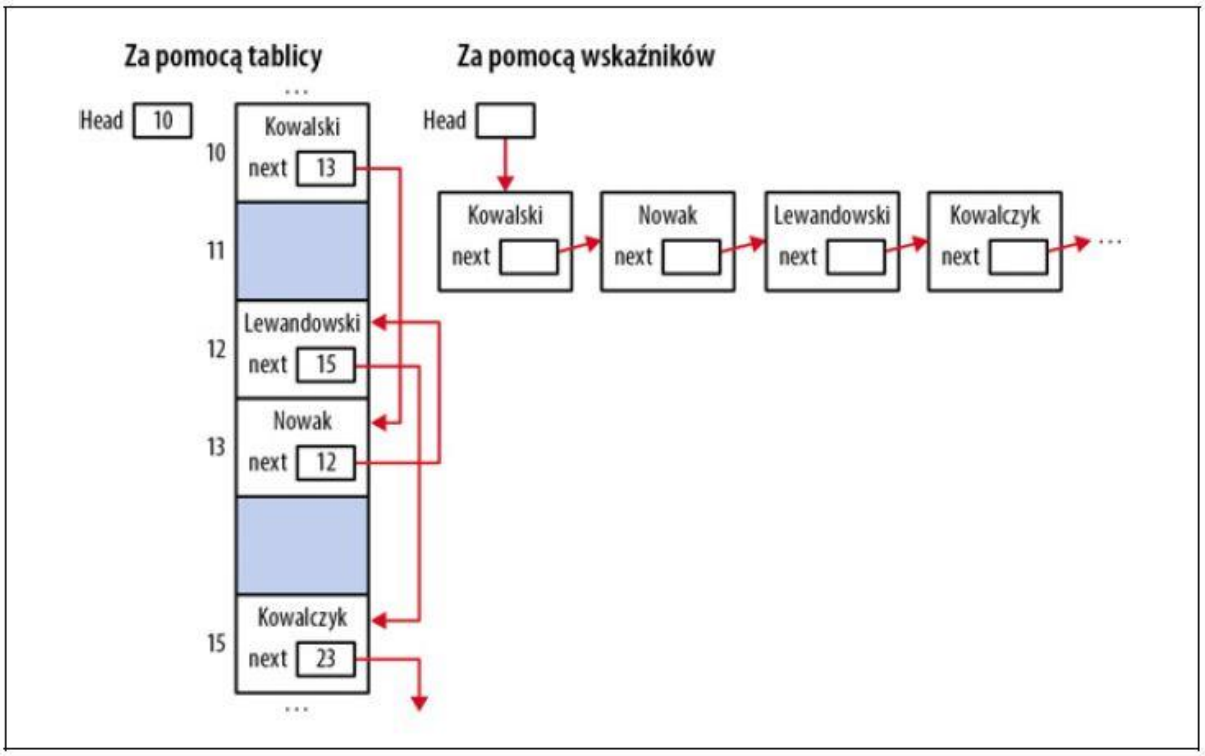
 przekazywania struktur danych bez ponoszenia kosztów w postaci narzutu,

 ochrony danych przekazywanych do funkcji jako parametry.

Logika działania wskaźników jest bliska zasadzie funkcjonowania komputera, a więc możliwe jest dzięki nim tworzenie szybszego i bardziej wydajnego kodu. To znaczy, że kompilator jest w stanie sprawniej przełożyć operacje na kod maszynowy. Korzystając ze wskaźników, tworzymy mniej narzutu niż w przypadku korzystania z innych operatorów. Przy użyciu wskaźników możliwa jest o wiele łatwiejsza implementacja wielu struktur danych. Np. lista powiązana może być obsługiwana za pomocą zarówno tablic, jak i wskaźników, ale stosując wskaźniki, można łatwiej odwoływać się bezpośrednio do następnego lub wcześniejszego powiązania.

Wykonanie tej samej operacji przy użyciu tablic wymaga korzystania z indeksów tablic, co nie jest tak intuicyjne i wygodne jak stosowanie wskaźników.

Rysunek poniżej obrazuje korzystanie z listy powiązanych elementów (listy pracowników) przy użyciu wskaźników i tablic. Z lewej strony rysunku pokazano operacje przeprowadzane za pomocą tablicy. Zmienna head (z ang. głowa) informuje o tym, że pierwszy element listy znajduje się pod indeksem tablicy o numerze 10. Każdy element tablicy zawiera strukturę reprezentującą danego pracownika. Pole *next* (z ang. następny), będące elementem struktury, przechowuje indeks, pod którym znajduje się tablica zawierająca dane następnego pracownika. Elementy zacieniowane symbolizują niewykorzystane elementy tablicy.



Prawa strona rysunku przedstawia tę samą operację przeprowadzaną przy użyciu wskaźników. Zmienna head przechowuje wskaźnik do węzła zawierającego dane pierwszego pracownika. Każdy węzeł przechowuje dane pracownika, a także wskaźnik do następnego węzła powiązanego z listą. Reprezentacja wykonana za pomocą wskaźników jest nie tylko bardziej czytelna, ale także bardziej elastyczna. Zwykle przed stworzeniem tablicy musimy określić jej rozmiar, co narzuca nam ograniczenie liczby elementów przechowywanych przez tablicę. Reprezentacja wykonana przy użyciu wskaźników nie narzuca takiego ograniczenia. W razie potrzeby nowy węzeł można dynamicznie alokować.

Wskaźniki w języku C są stosowane do obsługi dynamicznej alokacji pamięci.

Funkcja malloc jest stosowana do dynamicznego alokowania pamięci, a funkcja free jest używana do jej zwalniania. Dynamiczna alokacja pamięci pozwala na tworzenie struktur danych i tablic o zmiennym rozmiarze. Takie struktury to np. listy powiązane i kolejki. Jedynie nowszy standard języka C — C11 — obsługuje tablice o zmiennym rozmiarze. Zwarte wyrażenia mogą zawierać wiele informacji, ale jednocześnie mogą być trudne do

odczytania. Z tego powodu zapis wskaźników nie jest zrozumiały dla wielu programistów. Zwarty zapis powinien odpowiadać na konkretne potrzeby. Nie powinien być niepotrzebnie zagmatwany. W poniższej przykładowej sekwencji kodu trzeci znak drugiego elementu names (litera „w”) jest wyświetlany za pomocą dwóch różnych funkcji printf. Na razie takie zastosowanie wskaźników może wydawać się niejasne, jednakże zostanie ono wytłumaczone w podrozdziale „Wyłuskiwanie wskaźnika za pomocą operatora adresowania pośredniego”. Obie funkcje printf dają ten sam rezultat — wyświetlają literę w. Prostszym działaniem wydaje się jednak stosowanie notacji tablicowej.

```
char *names[] = {"Kowalski","Nowak","Kowalczyk"};

printf("%c\n",*(*(names+1)+2));

printf("%c\n",names[1][2]);
```

Wskaźniki są potężnym narzędziem służącym do tworzenia aplikacji oraz usprawniania ich działania. Jednakże korzystając ze wskaźników, możemy natrafić na liczne problemy, takie jak:

- próby uzyskania dostępu do danych znajdujących się poza granicami struktury (może mieć to miejsce w przypadku odczytu danych z tablicy);
- odwoływanie się do zmiennych automatycznych, gdy te zmienne już nie istnieją;
- odwoływanie się do pamięci alokowanej na stercie, gdy ta już została wcześniej zwolniona;
- wyłuskiwanie wskaźnika przed alokowaniem go w pamięci.

Składnia i semantyka wskaźników są jasno określone w specyfikacji języka C (*http://bit.ly/173cDxJ*). Pomimo tego można napotkać sytuacje, w których specyfikacja dokładnie nie określa zachowania wskaźnika. W takich przypadkach zachowanie wskaźnika jest:

Zdefiniowane przez implementację

Niektóre przypadki są zdefiniowane w dokumentacji. Przykładem zachowania zdefiniowanego przez implementację jest propagacja najbardziej znaczącego bitu podczas operacji prawostronnej zamiany elementów typu integer.

Nieokreślone

Niektóre implementacje są ustalone, ale nieudokumentowane. Przykładem tego może być ilość pamięci alokowanej przez funkcję malloc z argumentem zerowym. Listę tego typu zachowań można znaleźć w CERT Secure Coding, w załączniku DD (*http://bit.ly/YOFY8s*).

Niezdefiniowane

W takich przypadkach nie istnieją żadne nałożone wymagania, a więc może wyniknąć dosłownie wszystko. Takim przykładem jest dealokacja wartości wskaźnika za pomocą funkcji free. Listę tego typu przypadków można znaleźć w CERT Secure Coding w załączniku CC (*http://bit.ly/16msOVK*).

Niektóre zachowania są czasami określone miejscowo. Można je zwykle znaleźć w dokumentacji kompilatora. Tolerancja wynikająca z istnienia zachowań określonych miejscowo pozwala na generowanie bardziej wydajnego kodu.

Deklarowanie wskaźników

Deklaracja zmiennej będącej wskaźnikiem składa się z następujących po sobie elementów: typu danych, gwiazdki, nazwy wskaźnika. W poniższym przykładzie zadeklarowano obiekt typu integer, a także wskaźnik na element typu integer:

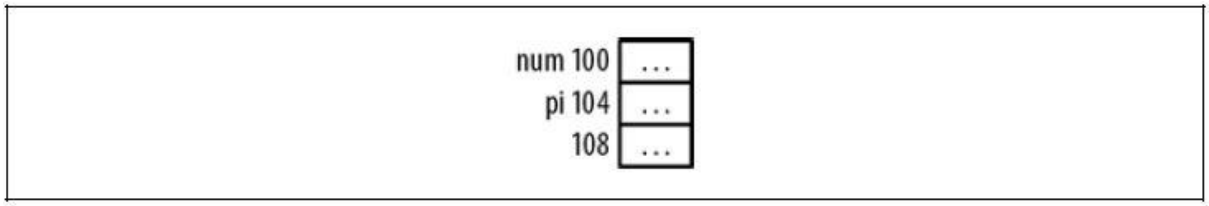
```
int num;

int *pi;
```

Stosowanie w zapisie znaku spacji nie ma tutaj znaczenia. Poniższe przykłady są równoznaczne z zapisem umieszczonym powyżej:

```
int* pi;

int * pi;

int *pi;

int*pi;
```

Gwiazdka informuje o tym, że dana zmienna jest wskaźnikiem. Symbol ten jest bardzo często używany. Korzysta się z niego również podczas manipulowania wskaźnikiem i wyłuskiwania go.

Rysunek poniżej wizualizuje sposób alokowania pamięci dla powyższych deklaracji. Komórki pamięci są przedstawione za pomocą trzech prostokątów. Numery znajdujące się po lewej stronie odpowiadają adresom zmiennych. Adres numer 100 został tu zastosowany w celu uczynienia rysunku wyraźniejszym. Zwykle nie znamy dokładnych adresów wskaźników ani jakichkolwiek innych zmiennych. W większości sytuacji taki dokładny adres nie interesuje nas jako programistów. Wielokropki symbolizują pamięć niezainicjowaną.



Wskaźniki na niezainicjowaną pamięć mogą być problematyczne. Gdy poddamy taki wskaźnik dereferencji, prawdopodobnie jego zawartość nie będzie określała poprawnego adresu. W przypadku, gdy będzie wskazywać na poprawny adres, może on nie zawierać poprawnych danych. Niepoprawnym adresem nazywamy adres, do którego dany program nie ma praw dostępu. Zaistnienie takiego adresu spowoduje na większości platform zakończenie działania programu. Może to prowadzić do licznych, poważnych problemów.

Zmienne num oraz pi znajdują się odpowiednio pod adresami 100 i 104. Zakładamy, że obie zmienne zajmują po 4 bajty każda. Rozmiary te mogą być różne w zależności od konfiguracji systemu.

Jeżeli nie zaznaczono inaczej, przyjmujemy w przedstawianych przykładach, że wszystkie obiekty typu integer zajmują po cztery bajty. W celu wyjaśnienia zasad działania wskaźników będziemy korzystać z adresów pamięci, takich jak np. 100. W dużym stopniu uprości to przykłady. Kiedy samodzielnie wykonasz zaprezentowane przykłady, otrzymasz zupełnie inne adresy. Adresy te mogą być różne w kolejnych uruchomieniach tego samego programu.

Warto pamiętać o tym, że:

 Wskaźnik pi powinien w końcu być przypisany do adresu zmiennej typu całkowitoliczbowego (integer).

 Przedstawione zmienne nie zostały zainicjowane, a więc zawierają bezużyteczne dane.

 Implementacja wskaźnika nie zawiera w swojej istocie niczego, co by mogło sugerować typ danych, na jakie wskazuje wskaźnik, oraz informować o poprawności wskazywanych danych. Jednakże, jeżeli określiliśmy typ wskaźnika, kompilator będzie sygnalizować sytuacje, w których wskaźnik nie będzie stosowany prawidłowo.

Poprzez dane bezużyteczne należy rozumieć takie elementy, które po alokowaniu pamięci mogą zawierać dowolne wartości.

Pamięć nie jest czyszczona po alokacji. Wcześniej mogły być w niej już zapisane jakieś dane. Jeżeli dany obszar pamięci wcześniej zawierał liczbę zmiennoprzecinkową, to interpretacja jej jako liczby całkowitej nie ma sensu. Nawet jeżeli była tam zapisana liczba całkowita, to prawdopodobnie nie będzie nam ona do niczego potrzebna. Dlatego dane zawarte w takiej pamięci są bezużyteczne.

Wskaźnik może być stosowany bez uprzedniego zainicjowania, jednakże może on nie działać prawidłowo do momentu inicjalizacji.

Interpretowanie deklaracji

Aby zrozumieć działanie wskaźników, warto przyjrzeć się ich deklaracjom. Należy je odczytywać od końca. Co prawda nie omówiliśmy jeszcze wskaźników na stałe, jednakże przyjrzyjmy się poniższej deklaracji.

const int *pci;

Odczytanie deklaracji od końca pozwoli Ci na jej stopniowe odszyfrowanie.

1. `pci` jest zmienną	`const int *pci;`
2. `pci` jest zmienną będącą wskaźnikiem	`const int *pci;`
3. `pci` jest zmienną będącą wskaźnikiem na element typu integer (liczba całkowita)	`const int *pci;`
4. `pci` jest zmienną będącą wskaźnikiem na stałą liczby całkowitej (typu integer)	`const int *pci;`

Według wielu programistów interpretowanie deklaracji „od końca” jest łatwiejsze.

Pracując ze złożonymi wskaźnikami, rysuj ich schematy. Takie schematy zostaną przedstawione przy wielu omawianych przykładach.

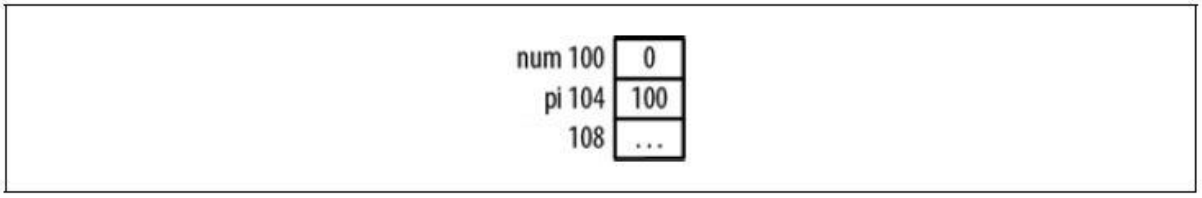
Operator adresu

Operator adresu & zwróci adres argumentu wyrażenia. Stosując adres zmiennej num, możesz zainicjować wskaźnik pi:

int num;

pi = #

Zmiennej num przypisano wartość zero, a zmienna pi ma wskazywać na adres zmiennej num.



Już podczas deklaracji zmiennych możesz zainicjować pi, aby wskazywała na adres num:

int num;

int *pi = #

Jednakże zastosowanie poniższych deklaracji w większości kompilatorów spowoduje wyświetlenie informacji o błędzie składni:

num = 0;

pi = num;

Wyświetli się komunikat błędu o następującej treści:

error: invalid conversion from 'int' to 'int*'

Zmienna pi jest wskaźnikiem na obiekt typu integer, a num jest zmienną typu integer. Komunikat o błędzie informuje nas, że nie możemy dokonać konwersji.

Przypisanie elementu typu integer do wskaźnika zwykle powoduje wyświetlenie przez kompilator ostrzeżenia lub komunikatu o błędzie.

Wskaźniki różnią się od zmiennych typu integer. Co prawda obydwa te elementy mogą być przechowywane w pamięci przy użyciu takiej samej liczby bajtów, jednakże są pomiędzy nimi znaczące różnice. Istnieje możliwość rzutowania zmiennej typu integer na wskaźnik na zmienną typu integer:

pi = (int *)num;

Zastosowanie powyższej instrukcji nie spowoduje wyświetlenia komunikatu o błędzie składni, jednakże wykonywany program może ulec anormalnemu zakończeniu podczas próby dereferencji wartości o adresie zero.

W większości systemów operacyjnych nie zawsze można wykorzystywać adres zerowy.

Dobrą praktyką stosowaną w programowaniu jest jak najszybsze inicjowanie wskaźnika, co ilustruje poniższy przykład:

```
int num;

int *pi;

pi = &num;
```

Wyświetlanie wartości wskaźników

W praktyce bardzo rzadko spotkasz zmienne posiadające adresy takie jak 100 i 104. Adres zmiennej można wyświetlić za pomocą następujących instrukcji:

```
int num = 0;

int *pi = &num;

printf("Adres num: %d Wartosc: %d\n",&num, num);

printf("Adres pi: %d Wartosc: %d\n",&pi, pi);
```

Po wykonaniu powyższych instrukcji uzyskasz dane wyjściowe podobne do poniższych. W tym przykładzie podaliśmy prawdziwe adresy. Adresy uzyskane przez Ciebie będą prawdopodobnie inne.

Adres num: 4520836 Wartosc: 0

Adres pi: 4520824 Wartosc: 4520836

Korzystając z funkcji printf podczas pracy ze wskaźnikami, możesz stosować inne przydatne specyfikatory pola.

Specyfikator	Funkcja specyfikatora
%x	wyświetla wartość w postaci liczby w systemie szesnastkowym
%o	wyświetla wartość w postaci liczby w systemie ósemkowym
%p	wyświetla wartość właściwą dla implementacji, zwykle jest to liczba w postaci szesnastkowej

Poniższe przykłady ilustrują zastosowanie tych specyfikatorów:

printf("Adres pi: %d Wartosc: %d\n",&pi, pi);

printf("Adres pi: %x Wartosc: %x\n",&pi, pi);

printf("Adres pi: %o Wartosc: %o\n",&pi, pi);

printf("Adres pi: %p Wartosc: %p\n",&pi, pi);

Powyższy ciąg instrukcji spowoduje wyświetlenie adresu i zawartości pi. W tym przypadku pi przechowuje adres num.

Adres pi: 4520824 Wartosc: 4520836

Adres pi: 44fb78 Wartosc: 44fb84

Adres pi: 21175570 Wartosc: 21175604

Adres pi: 0044FB78 Wartosc: 0044FB84

Specyfikator %p różni się od specyfikatora %x. Zwykle wyświetla liczbę w systemie szesnastkowym, stosując wielkie litery. O ile nie zaznaczono inaczej, specyfikator %p będzie stosowany do wyświetlania adresów. Konsekwentne wyświetlanie wartości wskaźników na różnych platformach jest zadaniem trudnym. Jednym ze sposobów na to jest rzutowanie wskaźnika jako wskaźnik na void, a następnie wyświetlenie go za pomocą specyfikatora %p:

printf("Wartosc pi: %p\n", (void*)pi);

Aby prezentowane przykłady były bardziej zrozumiałe, będziemy stosować specyfikator %p bez rzutowania adresu na wskaźnik na void.

Wyłuskiwanie wskaźnika za pomocą operatora adresowania pośredniego

Operator adresowania pośredniego — * — zwraca wartość, na którą wskazuje zmienna wskaźnika. Taką operację nazywa się wyłuskaniem (dereferencją) wskaźnika. W poniższym przykładzie zadeklarowano i zainicjowano zmienne num i wskaźnik pi:

int num = 5;

int *pi = #

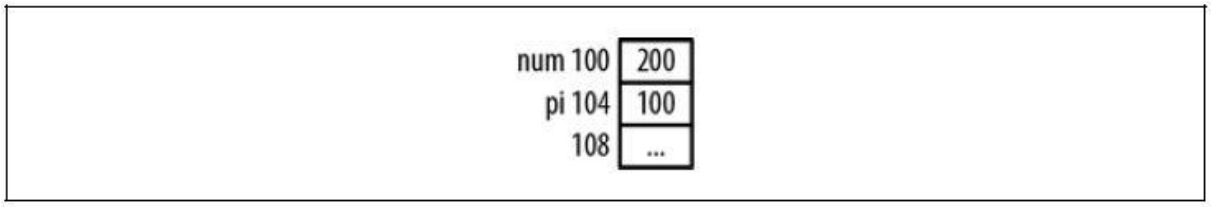
Zastosujmy operator adresowania pośredniego, aby wyświetlić 5 — wartość przechowywaną przez num:

printf("%p\n",*pi); *// wyświetla 5*

Rezultat dereferencji możemy również wykorzystać w roli **lvalue** (wartości lewostronnej). Termin ten odnosi się do argumentu znajdującego się po lewej stronie operatora przypisania. Wszystkie wartości lewostronne muszą być modyfikowalne w celu przeprowadzenia operacji przypisania.

W poniższym przykładzie przypiszesz wartość 200 do zmiennej typu integer wskazywanej przez pi. W związku z tym, że wskaźnik wskazuje na zmienną num, wartość 200 zostanie przypisana do tej zmiennej.

```
*pi = 200;

printf("%d\n",num); // wyświetla 200
```



Wskaźniki na funkcje

Wskaźnik może być zadeklarowany tak, aby wskazywał na funkcję. Zapis takiej deklaracji jest nieco skomplikowany. Poniżej przedstawiono sposób deklaracji wskaźnika na funkcję. Nie przekazujemy żadnych argumentów do funkcji, a funkcja niczego nie zwraca. Wskaźnik nazywamy **foo**:

```
void (*foo)();
```

Pojęcie wartości null

Zagadnienia związane z wartością null są ciekawe, aczkolwiek często mylone. Pomyłka może nastąpić w wyniku tego, że często mamy do czynienia z różnymi, choć podobnymi do siebie pojęciami, takimi jak:

- brak wartości,
- stała będąca wskaźnikiem zerowym,
- makro NULL,
- znak NUL w ASCII,

 pusty łańcuch znakowy,

 porównanie do wartości null.

Przypisanie wartości NULL wskaźnikowi skutkuje tym, że wskaźnik nie będzie niczego wskazywał. Pojęcie null (braku wartości) odnosi się do tego, że wskaźnik może przechowywać określoną wartość, która nie jest równa innemu wskaźnikowi. Wskaźnik pusty nie wskazuje adresu pamięci. Dwa puste wskaźniki są zawsze równe. Pustym można uczynić wskaźnik dowolnego typu (np. wskaźnik na znak, wskaźnik na zmienną typu integer), jednakże w praktyce jest to rzadko stosowany zabieg.

Pojęcie braku wartości jest pewną abstrakcją obsługiwaną za pomocą stałej wskaźnika pustego. Stała ta może, ale nie musi, być równa zeru. Programista języka C nie musi się przejmować jej wewnętrzną reprezentacją.

Makro NULL jest zerową stałą typu integer rzutowaną na wskaźnik na void. W wielu bibliotekach jest ona zdefiniowana w następujący sposób:

#define NULL ((void *)0)

To jest właśnie to, co zwykle nazywamy pustym wskaźnikiem. Jego definicję możesz znaleźć w różnych plikach nagłówkowych, takich jak: *stddef.h,*

stdlib.h, i *stdio.h*.

Jeżeli kompilator stosuje niezerowy wzorzec do reprezentacji wartości zerowej, to taki kompilator musi zapewnić to, że wszystkie wskaźniki, w których kontekście zastosowano 0 i NULL, będą traktowane jako puste. Właściwa wewnętrzna reprezentacja braku wartości jest definiowana przez implementację.

Symbole NULL i 0 są stosowane na poziomie języka tylko w celu utworzenia pustego wskaźnika.

Znak NUL w ASCII jest bajtem zawierającym same zera. Jednakże jest to coś zupełnie innego niż pusty wskaźnik. Łańcuch w języku C jest zapisywany jako ciąg znaków zakończonych wartością zerową. Pusty łańcuch nie zawiera żadnych znaków. Pusta instrukcja jest to instrukcja składająca się z samego średnika.

Jak się później przekonasz, pusty wskaźnik bardzo się przydaje do implementacji różnych struktur danych, takich jak np. listy powiązane, gdzie jest on stosowany do oznaczania końca listy.

Jeżeli naszym zamiarem jest przypisanie wartości zerowej wskaźnikowi pi, możemy to zrobić w następujący sposób:

pi = NULL;

NB. Pusty wskaźnik to nie to samo co wskaźnik niezainicjowany. Wskaźnik niezainicjowany może zawierać dowolną wartość. Z kolei pusty wskaźnik nie wskazuje żadnego miejsca w pamięci.

Co ciekawe, możemy przypisać wskaźnikowi wartość zerową, ale nie możemy mu przypisać żadnej innej wartości typu integer. Przyjrzyj się następującym operacjom przypisania:

```
pi = 0;

pi = NULL;

pi = 100; // spowoduje powstanie błędu składni

pi = num; // spowoduje powstanie błędu składni
```

Wskaźnik może być zastosowany jako samodzielny argument wyrażenia logicznego. Sprawdźmy na przykład, czy w wyniku zastosowania poniższego kodu wskaźnik będzie pusty:

```
if(pi) {

// wskaźnik nie jest pusty

} else {

// wskaźnik jest pusty

}
```

NB. Każde z dwóch zastosowanych wyrażeń jest prawidłowe, jednakże takie ich stosowanie jest zbędne. Bardziej czytelne, aczkolwiek niekonieczne, jest bezpośrednie porównanie z NULL.

Jeżeli w tym kontekście wskaźnikowi pi przypisano wartość NULL, będzie ona interpretowana jako zero binarne. Instrukcja else zostanie wykonana, jeżeli pi będzie zawierać NULL, ponieważ w języku C zero jest binarną reprezentacją fałszu.

```
if(pi == NULL) ...

if(pi != NULL) ...
```

NB. Nie powinno się dokonywać dereferencji pustych wskaźników, ponieważ nie zawierają one prawidłowego adresu. Próba wykonania takiej operacji będzie skutkować zakończeniem działania programu.

Przypisywać wartość zerową czy nie?

Co jest lepsze podczas pracy ze wskaźnikami? Przypisywanie im wartości 0 czy NULL? Każdy wybór jest dobry. Niektórzy programiści wolą stosować NULL, ponieważ przypomina im to o tym, że pracują ze wskaźnikami. Inni uważają, że nie jest to konieczne, ponieważ zero jest po prostu ukryte.

Nie powinno się jednakże stosować NULL w kontekście innym niż wskaźniki. Nie zawsze da to pożądany efekt. Z pewnością będzie problematyczne, gdy zostanie zastosowane zamiast znaku ASCII NUL. Znak ten w języku C nie jest definiowany przez żaden standardowy plik nagłówkowy. Jest on ekwiwalentem łańcucha znakowego \0, który, jako wartość dziesiętna, oznacza zero. Znaczenie zera zmienia się w zależności od kontekstu, w jakim zostało ono użyte. W jednym kontekście może oznaczać liczbę całkowitą, w innym pusty wskaźnik. Przeanalizuj poniższy przykład:

int num;

int *pi = 0; *// zero odnosi się do pustego wskaźnika*

pi = #

*pi = 0; *// zero odnosi się do elementu będącego liczbą całkowitą*

Przyzwyczailiśmy się do operatorów pełniących wiele funkcji. Takim operatorem jest na przykład gwiazdka. Jest ona stosowana do deklarowania wskaźników, dereferencji wskaźników, a także jest operatorem mnożenia.

Zero jest również elementem pełniącym wiele funkcji. Może być to dla Ciebie kłopotliwe, zwłaszcza jeżeli nie jesteś przyzwyczajony do tego, że argumenty operacji mogą pełnić wiele funkcji.

Wskaźniki na void

Wskaźnik na void jest wskaźnikiem ogólnego stosowania. Jest on przeznaczony do przechowywania odniesień do danych dowolnego typu. Oto przykładowy wskaźnik na void:

void *pv;

Przedstawiony wskaźnik posiada dwie interesujące właściwości:

 Wskaźnik na void ma taką samą reprezentację i organizację pamięci jak wskaźnik na char.

 Wskaźnik na void nigdy nie będzie równy innemu wskaźnikowi. Jednakże dwa wskaźniki na void, do których przypisano wartość NULL, będą sobie równe.

Każdy wskaźnik może zostać przypisany do wskaźnika na void. Później taki wskaźnik można z powrotem rzutować na jego początkowy typ. Po takiej operacji wartość wskaźnika będzie równa wartości wskaźnika przed zmianami. Taką operację pokazano poniżej. Wskaźnik int jest przypisywany do wskaźnika na void, a następnie wraca do swojej pierwotnej postaci:

```
int num;

int *pi = &num;

printf("Wartosc pi: %p\n", pi);

void* pv = pi;

pi = (int*) pv;

printf("Wartosc pi: %p\n", pi);
```

Adresy wyświetlone w wyniku działania tego programu będą identyczne:

Value of pi: 100

Value of pi: 100

Wskaźniki na void są stosowane przy wskaźnikach na dane, a nie wskaźnikach na funkcje.

NB. Bądź ostrożny, gdy stosujesz wskaźniki na void. Jeżeli przeprowadzisz operację rzutowania dowolnego wskaźnika na void, nic nie będzie zabezpieczać przed ewentualnym rzutowaniem go na inny typ wskaźnika.

Operator sizeof może być stosowany ze wskaźnikami na void, jednakże nie można go stosować w następujący sposób:

```
size_t size = sizeof(void*); // niedozwolona operacja

size_t size = sizeof(void); // niedozwolona operacja
```

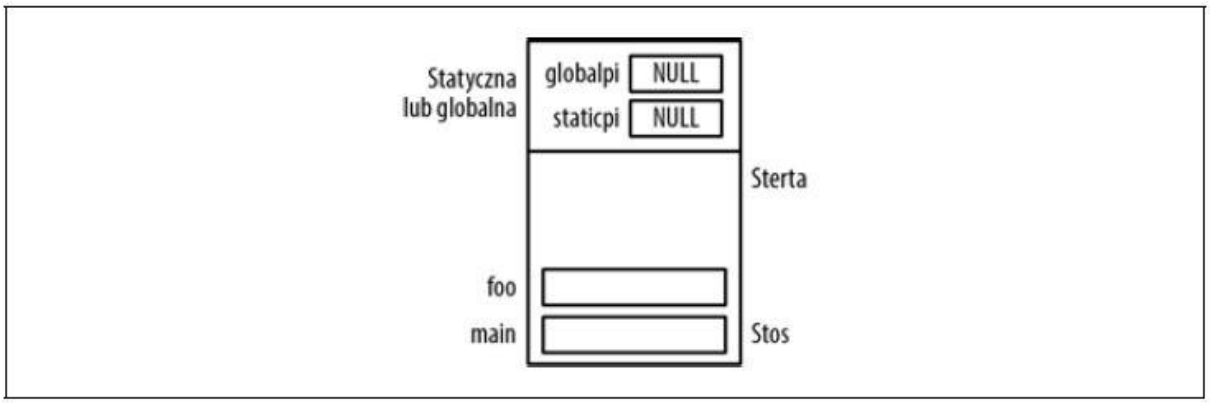
Typ size_t jest typem danych stosowanym do rozmiarów.

Wskaźniki globalne i statyczne

W chwili uruchomienia programu wskaźnik jest inicjowany wartością NULL (jeżeli jest zadeklarowany jako globalny lub statyczny). Poniżej przedstawiono przykłady wskaźników globalnego i statycznego.

```
int *globalpi;

void foo() {

static int *staticpi;

...

}

int main() {

...

}
```

Na rysunku przedstawiono ułożenie tych wskaźników w pamięci. Ramki stosu są odkładane na stos, a sterta jest wykorzystywana do dynamicznej alokacji pamięci. Przestrzeń pamięci ponad stosem jest wypełniana zmiennymi globalnymi i statycznymi. Jest to tylko diagram ideowy. Zmienne globalne i statyczne są często umieszczane w segmencie danych oddzielonym od segmentu, w którym znajdują się stos i sterta.



Stosowanie operatora sizeof ze wskaźnikami

Operator sizeof może być stosowany do określenia rozmiaru wskaźnika. Poniższy kod wyświetla rozmiar wskaźnika na obiekt typu char:

```
printf("Rozmiar *char: %d\n",sizeof(char*));
```

Stosując ten kod, otrzymamy następujące dane wyjściowe:

Rozmiar *char: 4

NB. Gdy chcesz wyświetlić rozmiar wskaźnika, zawsze korzystaj z operatora sizeof.

Operatory wskaźników

Istnieje kilka operatorów, których można używać podczas pracy ze wskaźnikami. Wcześniej omówiliśmy operatory wyłuskiwania i uzyskiwania adresu. Teraz omówimy porównywanie wskaźników oraz działania arytmetyczne przeprowadzane przy użyciu wskaźników. Tabela prezentuje operatory wskaźników.

Operator	Nazwa	Funkcja
*		deklaracja wskaźnika
*	operator dereferencji	wyłuskiwanie wskaźnika
–>	operator odwołania	uzyskiwanie dostępu do półstruktur wskazywanych przez wskaźnik
+	dodawanie	inkrementacja wskaźnika
–	odejmowanie	dekrementacja wskaźnika
==, !=	równość, nierówność	porównywanie dwóch wskaźników
>, >=, <, <=	większy od, większy lub równy, mniejszy od, mniejszy lub równy	porównywanie dwóch wskaźników
(typ danych)	rzutowanie	zmiana typu wskaźnika

Arytmetyka wskaźnikowa

Na wskaźnikach na dane można przeprowadzać między innymi następujące operacje arytmetyczne:

 dodanie liczby całkowitej do wskaźnika,

 odjęcie liczby całkowitej od wskaźnika,

 odjęcie od siebie dwóch wskaźników,

 porównywanie wskaźników.

Nie zawsze możliwe jest przeprowadzenie tych operacji na wskaźnikach na funkcje.

Dodawanie liczby całkowitej do wskaźnika

Operacja ta jest przydatna i często stosowana. Podczas dodawania liczby całkowitej do wskaźnika dodawaną wartością jest wynik mnożenia liczby całkowitej przez liczbę bajtów podstawowego typu danych.

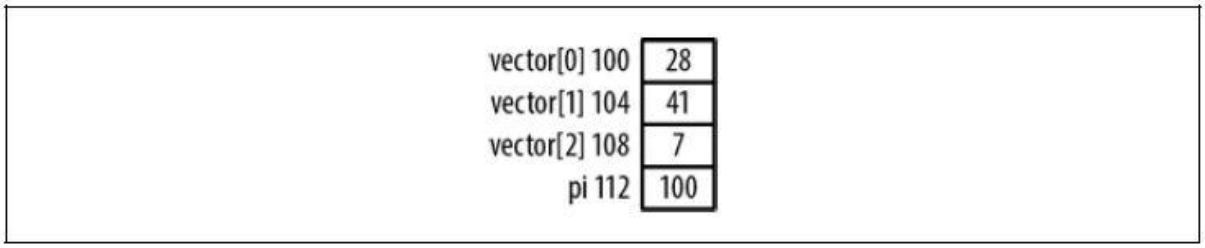
Rozmiar podstawowego typu danych jest różny w różnych systemach. Tabela poniżej prezentuje często spotykane rozmiary.

Typ danych	Rozmiar wyrażony w bajtach
byte	1
char	1
short	2
int	4
long	8
float	4
double	8

W poniższym przykładzie zaprezentowano efekty dodawania liczby całkowitej do wskaźnika. W tym celu zastosowano tablicę wypełnioną liczbami całkowitymi. Z każdą jedynką dodaną do pi liczba cztery zostaje dodana do adresu. Poniższy rysunek pokazuje alokację tych zmiennych. Aby umożliwić przeprowadzenie operacji arytmetycznych, zadeklarowano wskaźniki z typem danych. Znajomość rozmiarów poszczególnych typów danych pozwala na automatyczne dobranie wartości wskaźnika:

```
int vector[] = {28, 41, 7};

int *pi = vector; // pi: 100

printf("%d\n",*pi); // wyświetli 28

pi += 1; // pi: 104

printf("%d\n",*pi); // wyświetli 41

pi += 1; // pi: 108

printf("%d\n",*pi); // wyświetli 7
```

NB. Gdy zastosujesz samą nazwę tablicy, zwrócony zostanie Ci adres tablicy. Ten adres jest także adresem pierwszego elementu tablicy.



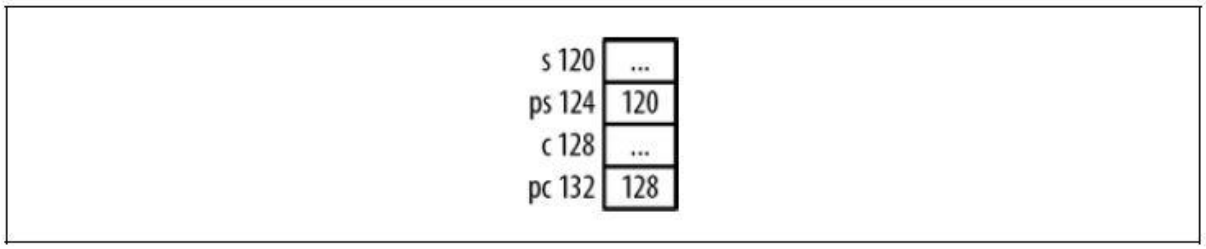
W poniższej sekwencji dodamy do wskaźnika liczbę trzy. Zmienna pi będzie zawierać adres 112 — adres pi:

```
pi = vector;

pi += 3;
```

Wskaźnik wskazuje na siebie. Taka operacja nie jest bardzo przydatna, aczkolwiek pokazuje, że musisz być ostrożny, stosując działania arytmetyczne na wskaźnikach. Istnieje niebezpieczeństwo próby dostępu do pamięci poza obszarem tablicy. Należy tego unikać. Nie ma żadnej gwarancji na to, że zostanie nam zwrócona poprawna zmienna. Bardzo łatwo jest wyliczyć nieważny lub bezużyteczny adres. Poniższa deklaracja zostanie użyta w celu przedstawienia operacji dodawania wykonanej najpierw z danymi typu short, a później z danymi typu char:

```
short s;

short *ps = &s;

char c;

char *pc = &c;
```

Załóżmy, że pamięć została alokowana w sposób przedstawiony na poniższym rysunku. Wszystkie zastosowane tu adresy mieszczą się w granicy czterech bajtów. Realne adresy mogą być przyporządkowane w innych granicach i w innej kolejności.



Poniższa sekwencja kodu dodaje jeden do każdego wskaźnika, a następnie wyświetla jego zawartość:

```
printf("Zawartość ps przed: %d\n",ps);

ps = ps + 1;

printf("Zawartość ps po: %d\n",ps);

printf("Zawartość pc przed: %d\n",pc);

pc = pc + 1;

printf("Zawartość pc po: %d\n",pc);
```

Po uruchomieniu program powinien zwrócić dane podobne do poniższych:

```
Zawartosc ps przed: 120

Zawartosc ps po: 122

Zawartosc pc przed: 128

Zawartosc pc po: 129
```

Wskaźnik ps jest inkrementowany o dwa, ponieważ rozmiar obiektu typu short to dwa bajty. Wskaźnik pc jest inkrementowany o jeden, ponieważ jego rozmiar wynosi jeden bajt. Podobnie jak w poprzednim przykładzie, uzyskane adresy mogą nie zawierać przydatnych danych.

Wskaźniki na void i dodawanie

Rozszerzenia większości kompilatorów pozwalają na wykonywanie działań arytmetycznych na wskaźnikach na void. Zakładamy, że rozmiar wskaźnika na void wynosi cztery. Próba dodania jedynki do wskaźnika na void może zakończyć się wyświetleniem komunikatu o błędzie składni. W poniższym fragmencie kodu zadeklarowano wskaźnik i dodano do niego jeden:

```
int num = 5;
```

```
void *pv = &num;

printf("%p\n",pv);

pv = pv+1; //ostrzeżenie o nieprawidłowej składni
```

Wyświetlone zostanie ostrzeżenie o następującej treści:

warning: pointer of type 'void *' used in arithmetic [-Wpointerarith]

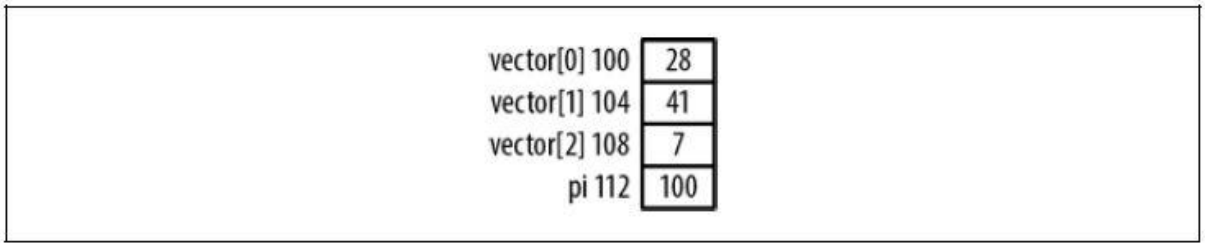
Kompilator wyświetlił ostrzeżenie, ponieważ w języku C operacje arytmetyczne nie są standardowo przeprowadzane na wskaźnikach na void. Wynik będący adresem przechowywanym przez pv będzie jednakże inkrementowany o cztery bajty.

Odejmowanie liczby całkowitej od wskaźnika

Liczby całkowite można odejmować od wskaźnika w taki sam sposób jak dodawać. Iloczyn rozmiaru typu danych i wartości danych obiektu typu integer jest odejmowany od adresu. Poniżej pokazano przykład ilustrujący odejmowanie liczby całkowitej od wskaźnika. W przykładzie zastosowano tabelę wypełnioną elementami typu integer. Pamięć zarezerwowaną dla tych zmiennych pokazano na rysunku.

```
int vector[] = {28, 41, 7};

int *pi = vector + 2; // pi: 108

printf("%d\n",*pi); // wyświetli 7

pi--; // pi: 104

printf("%d\n",*pi); // wyświetli 41

pi--; // pi: 100

printf("%d\n",*pi); // wyświetli 28
```

Za każdym razem, gdy od pi odjęto jeden, od adresu odjęto cztery.

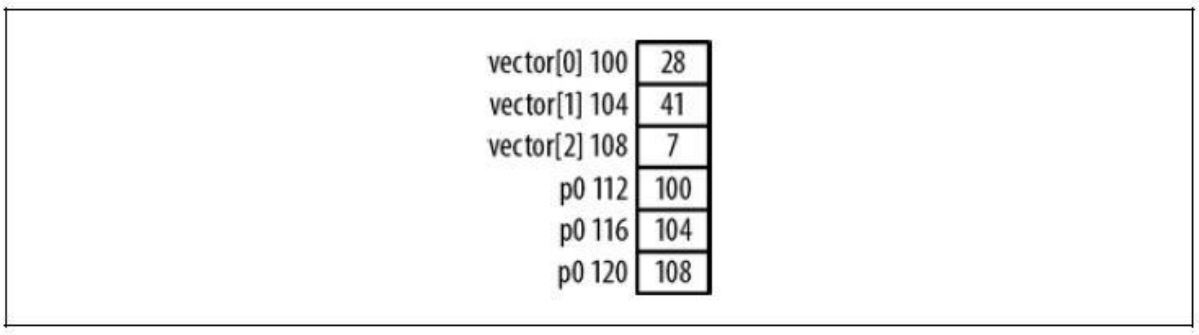


Odejmowanie wskaźników

Po odjęciu jednego wskaźnika od drugiego otrzymamy różnicę ich adresów. Jest to przydatne w zasadzie tylko do określania kolejności elementów w tablicy. Różnica pomiędzy wskaźnikami jest liczbą „jednostek", o jakie się różnią. Znak różnicy zależy od kolejności argumentów. Jest to zgodne z mechanizmem dodawania wskaźników, gdzie do rozmiaru typu danych wskaźnika dodawano liczbę. Zastosujmy „jednostki" w roli argumentów. W poniższym przykładzie deklarujemy tablicę i wskaźniki do jej elementów. Następnie obliczamy ich różnicę.

```
int vector[] = {28, 41, 7};

int *p0 = vector;

int *p1 = vector+1;

int *p2 = vector+2;

printf("p2-p0: %d\n",p2-p0); // p2-p0: 2

printf("p2-p1: %d\n",p2-p1); // p2-p1: 1

printf("p0-p1: %d\n",p0-p1); // p0-p1: -1
```

W pierwszej instrukcji printf obliczyliśmy, że różnica pomiędzy położeniem ostatniego i pierwszego elementu tablicy wynosi 2. Oznacza to, że ich indeksy różnią się o 2. W ostatniej instrukcji printf otrzymaliśmy wynik –1. Oznacza to, że p0 znajduje się bezpośrednio przed elementem, na który wskazuje p1. Na rysunku przedstawiono alokację pamięci.



Typ ptrdiff_t jest przenośnym sposobem wyrażania różnicy pomiędzy dwoma wskaźnikami. W poprzednim przykładzie wynik odejmowania dwóch wskaźników został zwrócony jako typ ptrdiff_t. Stosowanie tego typu ułatwia pracę z odejmowaniem wskaźników, ponieważ wskaźniki mogą różnić się pod względem rozmiaru.

Nie myl tej techniki z zastosowaniem operatora wyłuskiwania do odejmowania dwóch liczb. W poniższym przykładzie zastosujemy dwa wskaźniki w celu określenia różnicy pomiędzy wartościami przechowywanymi przez pierwszy i drugi element tablicy:

printf("*p0-*p1: %d\n",*p0-*p1); *// *p0-*p1: -13*

Porównywanie wskaźników

Wskaźniki można porównywać za pomocą standardowych operatorów porównania. Zwykle takie porównywanie nie jest nam do niczego przydatne. Jednakże porównywanie wskaźników na elementy tablicy może nam pozwolić określić względną kolejność elementów tablicy. Aby pokazać porównywanie wskaźników, skorzystamy z poprzedniego przykładu (patrz „Odejmowanie wskaźników"). Zastosujemy kilka operatorów porównania, które wyświetlą wyniki w postaci 1 (prawda), 0 (fałsz):

int vector[] = {28, 41, 7};

int *p0 = vector;

int *p1 = vector+1;

int *p2 = vector+2;

printf("p2>p0: %d\n",p2>p0); *// p2>p0: 1*

printf("p2<p0: %d\n",p2<p0); *// p2<p0: 0*

printf("p0>p1: %d\n",p0>p1); *// p0>p1: 0*

Zastosowania wskaźników

Wskaźniki mogą być stosowane na wiele sposobów. Tutaj omówimy dwa zagadnienia:

 wielopoziomowe adresowanie pośrednie,

 wskaźniki na stałe.

Wielopoziomowe adresowanie pośrednie

Wskaźniki mogą być stosowane do wielopoziomowego adresowania pośredniego. Dość często spotyka się zmienne zadeklarowane jako wskaźnik na wskaźnik. Czasami są one nazywane **podwójnymi wskaźnikami**. Dobrym przykładem tego jest sytuacja, gdy argumenty programu są przekazywane do funkcji main za pomocą tradycyjnych parametrów argc i argv.

W poniższym przykładzie użyto trzech tablic. Pierwszą tablicą jest tablica łańcuchów znaków zastosowana do przechowywania listy tytułów książek:

```
char *titles[] = {"Opowieść o dwóch miastach",

"Komu bije dzwon","Don Kichot",

"Odyseja","Moby Dick","Hamlet",

"Podróże Guliwera"};
```

Dwie pozostałe tablice mają być listami „najlepszych książek” oraz książek anglojęzycznych autorów. Zamiast przechowywać kopie tytułów, będą one przechowywały adresy tytułów umieszczonych w tablicy titles. Obie tablice będą musiały być zadeklarowane jako wskaźnik na wskaźnik na char. Elementy tych tablic będą przechowywały adresy elementów z tablicy titles. Dzięki temu tytuły nie będą musiały być duplikowane w pamięci.

Będą zawarte tylko w jednym miejscu. Gdybyś chciał zmienić tytuł książki, będziesz musiał to zrobić tylko w jednym miejscu.

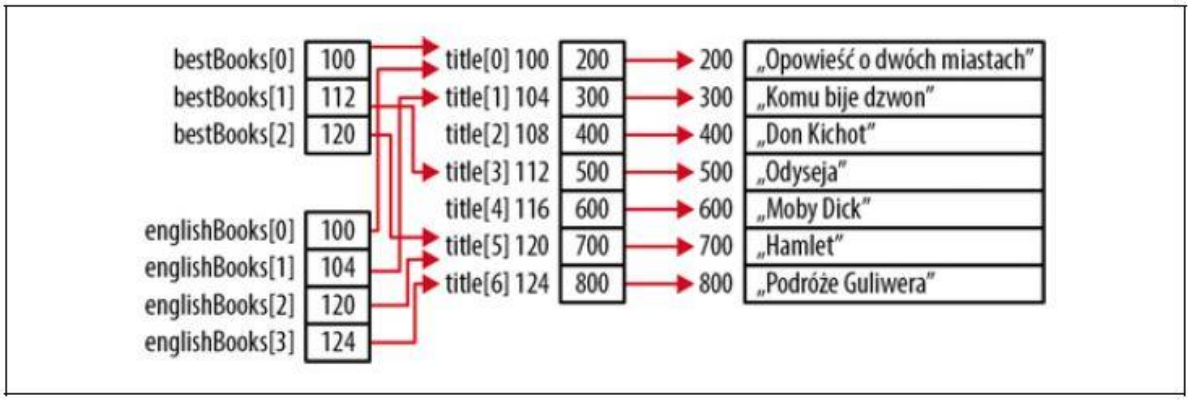
Poniżej przedstawiono deklaracje omówionych tablic. Każdy element tablicy zawiera wskaźnik, który wskazuje na drugi wskaźnik na char.

```
char **bestBooks[3];

char **englishBooks[4];
```

Obydwie tablice zostają zainicjowane, a jeden z ich elementów jest wyświetlony na ekranie. Podczas operacji przypisania wartość z prawej strony jest obliczana dzięki zastosowaniu w pierwszej kolejności operatora indeksu. Następnie pobierany jest adres operatora. Np. druga linia kodu w poniższym przykładzie przypisuje adres czwartego elementu tablicy titles do drugiego elementu tablicy bestBooks:

```
bestBooks[0] = &titles[0];

bestBooks[1] = &titles[3];

bestBooks[2] = &titles[5];

englishBooks[0] = &titles[0];

englishBooks[1] = &titles[1];

englishBooks[2] = &titles[5];

englishBooks[3] = &titles[6];

printf("%s\n",*englishBooks[1]); // Komu bije dzwon
```

Rysunek pokazuje alokacje pamięci dla powyższego przykładu.



Stosowanie wielopoziomowego adresowania pośredniego poszerza elastyczność pisanego kodu, a także poszerza wachlarz możliwości tworzenia kodu. Bez tej techniki przeprowadzenie niektórych operacji byłoby dość trudne. Zmiana adresu tytułu w przytoczonym przykładzie wymaga jedynie modyfikacji tablicy title. Nie musimy modyfikować pozostałych tablic. Nie ma określonego ograniczenia co do liczby poziomów adresowania pośredniego. Oczywiście stosowanie zbyt wielu poziomów tego adresowania może okazać się dla Ciebie kłopotliwe i trudne.

Stałe i wskaźniki

Możliwość zestawienia słowa klucza const ze wskaźnikami jest bardzo złożoną i przydatną cechą języka C. Możliwość ta daje programiście zestaw zabezpieczeń przydatnych podczas rozwiązywania różnych problemów. Wskaźnik na funkcję jest szczególnie przydatnym elementem języka C.

Wskaźniki na stałą

Wskaźniki można zdefiniować tak, aby wskazywały na stałą. Oznacza to, że wskaźnik nie może być zastosowany do modyfikowania wartości, do której się odnosi. W poniższym przykładzie zadeklarowano liczbę całkowitą (integer), a także stałą typu integer. Następnie zadeklarowano wskaźnik na liczbę całkowitą, a także wskaźnik na stałą typu integer, po czym zainicjowano je odpowiednimi liczbami:

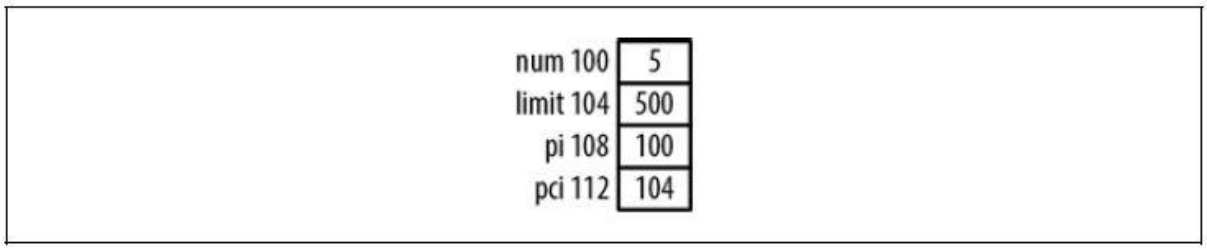
```
int num = 5;

const int limit = 500;
```

int *pi; *// wskaźnik do liczby całkowitej*

const int *pci; *// wskaźnik na stałą typu integer*

pi = #

pci = &limit;



Poniższa sekwencja kodu wyświetla adresy i wartości zmiennych:

printf(" num - Adres: %p wartosc: %d\n",&num, num);

printf("limit - Adres: %p wartosc: %d\n",&limit, limit);

printf(" pi - Adres: %p wartosc: %p\n",&pi, pi);

printf(" pci - Adres: %p wartosc: %p\n",&pci, pci);

Program po uruchomieniu zwróci wartości zbliżone do poniższych:

num - Address: 100 value: 5

limit - Address: 104 value: 500

pi - Address: 108 value: 100

pci - Address: 112 value: 104

Jeżeli chcesz odczytać wartość elementu typu integer, możesz stosować wyłuskiwanie wskaźnika na stałą. Jak pokazano poniżej, odczyt jest w pełni dozwoloną, a wręcz niezbędną operacją:

printf("%d\n", *pci);

Nie możemy dokonać wyłuskania wskaźnika na stałą w celu modyfikacji wskazywanego obiektu, ale możemy dokonać modyfikacji wskaźnika. Wartość wskaźnika nie jest stałą. Wskaźnik można zmienić w celu wskazania na inną stałą typu integer bądź na obiekt typu

integer. Czynność ta jest łatwa. Deklaracja po prostu ogranicza nasze możliwości modyfikowania wskazywanej zmiennej. Dopuszczalne jest więc następujące przypisanie:

pci = #

Możemy dokonać dereferencji pci w celu jej odczytania, jednakże nie możemy dokonać dereferencji w celu jej modyfikacji.

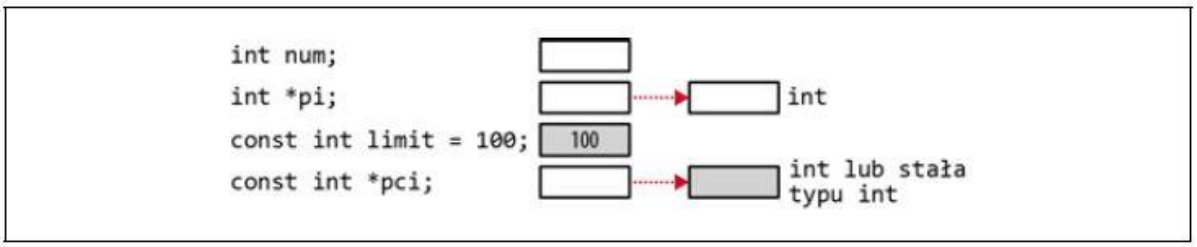
Przeanalizuj poniższe przypisanie:

*pci = 200;

Spowoduje ono wyświetlenie następującego komunikatu o błędzie składni:

'pci' : you cannot assign to a variable that is const

Wskaźnik „myśli”, że wskazuje stałą typu integer, a więc zezwala na jej modyfikację. Możemy modyfikować num przy użyciu nazwy, ale nie możemy w tym celu korzystać z pci. Ponizszy rysunek ilustruje pojęcie wskaźnika na stałą. Puste pola symbolizują zmienne, które można modyfikować. Zacieniowane pola symbolizują zmienne, których nie można modyfikować. Zacieniowane pole wskazywane przez pci nie może zostać zmienione za pomocą pci. Linie zakończone strzałkami ilustrują to, że wskaźnik może wskazywać na ten typ danych. W poprzednim przykładzie pci wskazywał na limit.



Deklaracja pci jako wskaźnika na stałą typu integer oznacza, że:

 pci może być przypisane do wskazywania na inne stałe typu integer;

 pci może być przypisane do wskazywania na inne obiekty typu integer niebędące stałymi;

 pci może być wyłuskiwane w celu odczytu;

 pci nie może być wyłuskiwane w celu zmiany tego, na co wskazuje.

NB. Kolejność określenia typu i słowa klucza const nie musi być zachowana. Poniższe zapisy są sobie równoważne:

const int *pci;

int const *pci;

Wskaźniki typu constant stosowane do obiektów niebędących stałymi

Istnieje możliwość zadeklarowania wskaźnika na stałą na obiekt niebędący stałą. Gdy dokonamy takiej deklaracji, nie będziemy mogli zmienić wskaźnika, ale uzyskamy możliwość modyfikacji wskazywanych danych. Przykładem takiego wskaźnika jest:

int num;

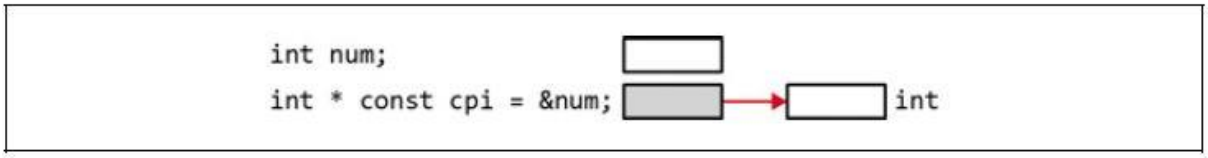
int *const cpi = #

Po takiej deklaracji:

 cpi musi być zainicjowane zmienną niebędącą stałą;

 cpi nie może być modyfikowane;

 dane, na które wskazuje cpi, mogą być modyfikowane.



Możliwe jest wyłuskanie cpi i przypisanie nowej wartości do jakiegokolwiek elementu wskazywanego przez cpi. Można tego dokonać na dwa sposoby:

*cpi = limit;

*cpi = 25;

Jeżeli jednakże spróbujemy zainicjować cpi stałą limit, tak jak pokazano to poniżej, zostanie wyświetlony komunikat z ostrzeżeniem.

const int limit = 500;

int *const cpi = &limit;

Ostrzeżenie będzie miało następującą treść:

warning: initialization discards qualifiers from pointer target type

Inicjacja odrzuca kwalifikatory z docelowego typu wskaźnika. Jeżeli cpi odsyłałoby do stałej limit, stała ta mogłaby być modyfikowana, co nie jest pożądane. W większości przypadków chcemy, aby stałe pozostawały stałymi.

Gdy już przypiszemy adres do cpi, nie możemy przypisać cpi żadnej nowej wartości:

int num;

int age;

int *const cpi = #

cpi = &age;

Na ekranie zostanie wyświetlony komunikat błędu informujący Cię o tym, że nie możesz przypisać niczego do zmiennej, która jest stałą:

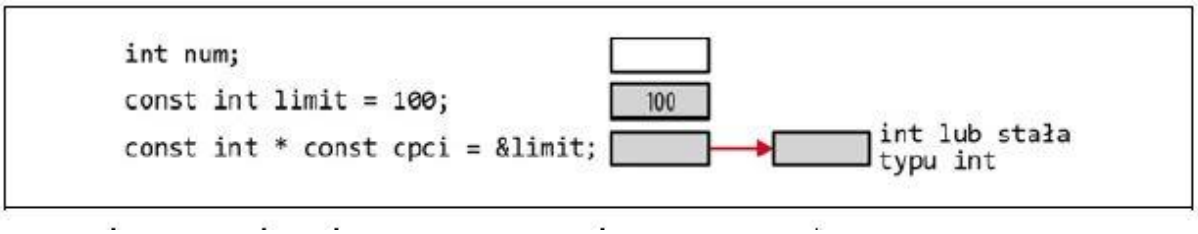
'cpi' : you cannot assign to a variable that is const

Wskaźniki typu constant stosowane do stałych

Rzadko stosuje się wskaźniki typu constant do obsługi stałych. W takim przypadku nie można modyfikować zarówno wskaźnika, jak i wskazywanych danych. Poniżej znajduje się przykład wskaźnika typu constant na stałą typu integer:

const int * const cpci = &limit;

Wskaźnik typu constant wskazujący stałą został przedstawiony na rysunku:



Stosując wskaźniki na stałe, nie musisz przypisywać adresu stałej do cpci. Możliwe jest za to num, co pokazano poniżej:

int num;

const int * const cpci = #

Po zadeklarowaniu wskaźnika musisz go zainicjować. Jeżeli tego nie zrobisz, kompilator wyświetli komunikat informujący o błędzie składni, co pokazuje poniższy przykład:

const int * const cpci;

Komunikat błędu poinformuje Cię, że musisz zainicjować wskaźnik:

'cpci' : const object must be initialized if not extern

Stosując omawiane wskaźniki, nie możesz:

 modyfikować wskaźnika,

 modyfikować danych wskazywanych przez wskaźnik.

Próba przypisania cpci nowego adresu będzie skutkowała wyświetleniem komunikatu o błędzie składni:

cpci = #

Następujący komunikat błędu poinformuje Cię, że nie możesz przypisywać do zmiennej typu const:

'cpci' : you cannot assign to a variable that is const

Błąd składni powstanie również, jeżeli spróbujesz wyłuskać wskaźnik w celu przypisania nowej wartości:

*cpci = 25;

Wyświetlony komunikat błędu poinformuje Cię, że nie możesz przypisać wartości do stałej. Wyrażenie musi być modyfikowalną wartością lewostronną. Komunikat będzie mieć treść:

'cpci' : you cannot assign to a variable that is const

expression must be a modifiable lvalue

Rzadko stosuje się wskaźniki typu constant do wskazywania na stałe.

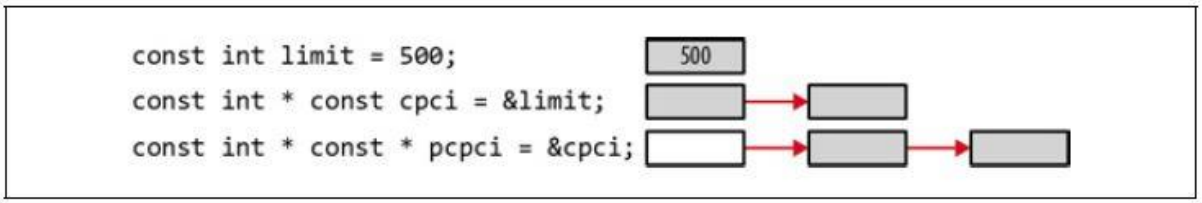
Wskaźniki na (wskaźniki typu constant na stałe)

Wskaźniki na stałą mogą być także stosowane do wielopoziomowego adresowania pośredniego. W poniższym przykładzie przypisujemy wskaźnik do opisanego wcześniej wskaźnika cpci. Czytanie złożonych deklaracji „od prawej do lewej” ułatwia zrozumienie ich treści.

const int * const cpci = &limit;

const int * const * pcpci;

Rysunek przedstawia schemat wskaźnika na wskaźnik na stałą.



Poniższy przykład ilustruje zastosowanie takich wskaźników. W wyniku działania poniższego kodu liczba 500 powinna zostać wyświetlona dwukrotnie.

```
printf("%d\n",*cpci);

pcpci = &cpci;

printf("%d\n",**pcpci);
```

Poniższa tabela podsumowuje wiadomości na temat opisanych czterech typów wskaźników.

Typ wskaźnika	Czy wskaźnik jest modyfikowalny?	Czy wskazywane dane są modyfikowalne?
wskaźnik na obiekt niebędący stałą	tak	tak
wskaźnik na stałą	tak	nie
wskaźnik będący stałą na obiekt niebędący stałą	nie	tak
stała będąca wskaźnikiem na stałą	nie	nie