

*Kompendium wiedzy
o wskaźnikach w języku C!*

Wskaźniki w języku C

Przewodnik



HELION

O'REILLY®

Richard Reese

Tytuł oryginału: Understanding and Using C Pointers

Tłumaczenie: Konrad Matuk

ISBN: 978-83-246-8289-8

© 2014 Helion S.A.

Authorized Polish translation of the English edition of Understanding and Using C Pointers
ISBN 9781449344184 © 2013 Richard Reese, Ph.D.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/wskazc>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Przedmowa	9
 1. Wstęp	 15
Wskaźniki i pamięć	16
Dlaczego warto opanować wskaźniki	17
Deklarowanie wskaźników	20
Interpretowanie deklaracji	22
Operator adresu	23
Wyświetlanie wartości wskaźników	24
Wyłuskiwanie wskaźnika za pomocą operatora adresowania pośredniego	26
Wskaźniki na funkcje	27
Pojęcie wartości null	27
Rodzaje wskaźników i ich rozmiary	32
Modele pamięci	32
Predefiniowane typy związane ze wskaźnikami	33
Operatory wskaźników	37
Arytmetyka wskaźnikowa	37
Porównywanie wskaźników	42
Zastosowania wskaźników	42
Wielopoziomowe adresowanie pośrednie	43
Stałe i wskaźniki	44
Podsumowanie	50
 2. C i dynamiczne zarządzanie pamięcią	 51
Dynamiczna alokacja pamięci	52
Wycieki pamięci	55
Funkcje dynamicznego alokowania pamięci	57
Stosowanie funkcji malloc	58
Stosowanie funkcji calloc	62

Stosowanie funkcji realloc	63
Funkcja alloca i tablice o zmiennej długości	66
Dealokacja pamięci przy użyciu funkcji free	66
Przypisywanie wartości NULL do zwalnianego wskaźnika	68
Podwójne uwalnianie	68
Stera i pamięć systemowa	70
Zwalnianie pamięci po zakończeniu działania programu	70
Wiszące wskaźniki	71
Przykłady wiszących wskaźników	71
Rozwiązywanie problemu wiszących wskaźników	74
Stosowanie wersji testowej do wykrywania wycieków pamięci	74
Techniki dynamicznej alokacji pamięci	75
Sprzątanie pamięci w języku C	76
Inicjowanie przy pozyskaniu zasobu (RAII)	76
Korzystanie z procedury obsługi wyjątków	77
Podsumowanie	78
3. Wskaźniki i funkcje	79
Stera i stos programu	80
Stos programu	80
Organizacja ramki stosu	81
Przekazywanie i zwracanie za pomocą wskaźnika	84
Stosowanie wskaźników do przekazywania danych	84
Przekazywanie danych poprzez wartość	85
Przekazywanie wskaźnika do stałej	86
Zwracanie wskaźnika	87
Wskaźniki do danych lokalnych	89
Przekazywanie pustych wskaźników	91
Przekazywanie wskaźnika do wskaźnika	91
Wskaźniki na funkcję	95
Deklarowanie wskaźników na funkcję	96
Stosowanie wskaźników na funkcję	97
Przekazywanie wskaźników na funkcję	99
Zwracanie wskaźników na funkcję	99
Stosowanie tablic wskaźników na funkcję	100
Porównywanie wskaźników na funkcję	101
Rzutowanie wskaźników na funkcję	102
Podsumowanie	103

4. Wskaźniki i tablice	105
Tablice	106
Tablice jednowymiarowe	107
Tablice dwuwymiarowe	108
Tablice wielowymiarowe	109
Notacja wskaźnikowa i tablice	109
Różnice pomiędzy tablicami a wskaźnikami	112
Stosowanie funkcji malloc do tworzenia tablic jednowymiarowych	113
Stosowanie funkcji realloc do zmiany rozmiaru tablicy	114
Przekazywanie tablicy jednowymiarowej	118
Stosowanie notacji tablicowej	118
Stosowanie notacji wskaźnikowej	119
Stosowanie jednowymiarowych tablic wskaźników	120
Wskaźniki i tablice wielowymiarowe	122
Przekazywanie tablicy wielowymiarowej	125
Dynamiczna alokacja tablicy dwuwymiarowej	128
Alokowanie pamięci o potencjalnie nieciągłym obszarze	129
Alokacja pamięci o ciągłym obszarze	129
Tablice postrzępione i wskaźniki	131
Podsumowanie	135
 5. Wskaźniki i łańcuchy	 137
Podstawowe wiadomości na temat wskaźników	138
Deklaracja łańcucha	139
Pula literałów łańcuchowych	139
Inicjalizacja łańcucha	141
Standardowe operacje wykonywane na łańcuchach	145
Porównywanie łańcuchów	145
Kopiowanie łańcuchów	147
Łączenie łańcuchów	149
Przekazywanie łańcuchów	153
Przekazywanie prostego łańcucha	153
Przekazywanie wskaźnika na stałą typu char	155
Przekazywanie wskaźnika wymagającego inicjalizacji	155
Przekazywanie argumentów do aplikacji	157

Zwracanie łańcuchów	158
Zwracanie adresu literału	158
Zwracanie adresu pamięci adresowanej dynamicznie	160
Wskaźniki na funkcje i łańcuchy	162
Podsumowanie	165
6. Wskaźniki i struktury	167
Wstęp	168
Alokacja struktury w pamięci	169
Zagadnienia związane z dealokacją struktury	170
Unikanie narzutu wynikającego ze stosowania funkcji malloc i free	174
Stosowanie wskaźników do obsługi struktur danych	176
Jednostronna lista powiązana	177
Stosowanie wskaźników do obsługi kolejek	185
Stosowanie wskaźników do obsługi stosu	188
Stosowanie wskaźników do obsługi drzewa	190
Podsumowanie	194
7. Problemy z zabezpieczeniami	
 i niewłaściwe stosowanie wskaźników	195
Deklaracja i inicjalizacja wskaźników	197
Niewłaściwa deklaracja wskaźnika	197
Niepowodzenie inicjalizacji wskaźnika przed użyciem	198
Rozwiązywanie problemów	
z niezainicjalizowanymi wskaźnikami	198
Problemy wynikające ze stosowania wskaźników	199
Wykrywanie wartości zerowej	200
Niewłaściwe stosowanie operatora wyłuskiwania	201
Wiszące wskaźniki	201
Uzyskiwanie dostępu do pamięci	
znajdującej się poza granicami tablicy	202
Błędne obliczenie rozmiaru tablicy	203
Niewłaściwe stosowanie operatora sizeof	203
Zawsze dopasowuj do siebie typy wskaźników	204
Wskaźniki ograniczone	205
Problemy z zabezpieczeniami związane z łańcuchami	206
Arytmetyka wskaźnikowa i struktury	207
Problemy związane ze wskaźnikami na funkcję	209

Problemy związane z dealokacją pamięci	211
Dublowanie funkcji free	211
Czyszczenie danych wrażliwych	211
Stosowanie narzędzi analizy statycznej	212
Podsumowanie	213
8. Pozostałe techniki	215
Rzutowanie wskaźników	216
Uzyskiwanie dostępu do adresu specjalnego przeznaczenia	217
Uzyskiwanie dostępu do portu	219
Uzyskiwanie dostępu do pamięci przy użyciu DMA	220
Określanie porządku bajtów danej maszyny	220
Aliasing wskaźników i słowo kluczowe restrict	221
Stosowanie unii do reprezentacji wartości na różne sposoby	223
Strict aliasing	225
Stosowanie słowa kluczowego restrict	226
Wątki i wskaźniki	227
Współdzielenie wskaźników przez wątki	228
Stosowanie wskaźników	
na funkcję do obsługi wywołań zwrotnych	231
Techniki obiektowe	233
Tworzenie i stosowanie wskaźników nieprzeźroczystych	233
Polimorfizm w języku C	237
Podsumowanie	242
Skorowidz	243

Wstęp

Dobre opanowanie wskaźników i umiejętność ich efektywnego stosowania odróżniają zaawansowanego programistę od nowicjusza. Wskaźniki są ważnym elementem języka C, zapewniającym mu elastyczność. Umożliwiają obsługę dynamicznej alokacji pamięci. Ich zapis jest związany z zapisem tablic. Wskaźniki na funkcję dają ogromne możliwości kontroli wykonywania programu.

Wskaźniki od dawna są główną przeszkodą w nauce języka C. W założeniu wskaźnik jest zmienną przechowującą adres pamięci. Ten prosty element zaczyna wydawać się skomplikowany, gdy zaczynamy stosować operatory wskaźników lub gdy próbujemy zrozumieć ich skomplikowaną notację. Takie problemy nie muszą grozić każdemu. Jeżeli dobrze opanujesz proste podstawy wskaźników, wtedy ich zaawansowane zastosowania wcale nie są trudne do opanowania.

Kluczem do opanowania wskaźników jest zrozumienie tego, jak program napisany w języku C zarządza pamięcią — wskaźniki zawierają adresy komórek pamięci. Jeżeli nie zrozumiesz tego, jak zorganizowana jest pamięć, trudno będzie Ci pojąć, jak działają wskaźniki. W celu ułatwienia nauki książka zawiera rysunki ilustrujące organizację pamięci. Umieszczono je wszędzie tam, gdzie mogą okazać się pomocne w wyjaśnianiu działania wskaźników. Gdy już zrozumiesz organizację pamięci, zgłębianie zagadnień związanych ze wskaźnikami stanie się o wiele łatwiejsze.

Poniższy rozdział zawiera wstępne informacje na temat wskaźników, ich operatorów oraz tego, jak wskaźniki oddziałują na pamięć. Pierwsza część rozdziału opisuje deklaracje wskaźników, podstawowe operatory wskaźników, a także pojęcie wartości typu `null`. Istnieje wiele rodzajów wartości typu `null` obsługiwanych przez język C. Dokładne przeanalizowanie tego zagadnienia może okazać się przydatne.

Drugi podrozdział opisuje różnorakie modele pamięci, które z pewnością napotkasz, pracując w języku C. Model stosowany przez dany kompilator lub system operacyjny wpływa na działanie wskaźników. Ponadto w tym podrozdziale spotkasz się z analizą różnych predefiniowanych typów związanych ze wskaźnikami i modelami pamięci.

W kolejnym podrozdziale opisano bardziej szczegółowo operatory wskaźników. Dowiesz się wielu przydatnych rzeczy na temat arytmetyki wskaźników oraz ich porównywania. Ostatni podrozdział opisuje problematykę związaną ze stałymi i wskaźnikami. Liczne kombinacje deklaracji oferują wiele interesujących, a zarazem przydatnych możliwości.

Niezależnie od tego, czy jesteś początkującym, czy zaawansowanym programistą języka C, niniejsza książka pomoże Ci zrozumieć wskaźniki i wypełnić braki w wiedzy. Zaawansowany programista może czytać tę publikację selektywnie, wybierając tylko te tematy, które go interesują. Początkujący programista powinien ostrożnie zapoznać się z całością książki.

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. Problematyka ta została szerzej opisana w rozdziale 2.

Tabela 1.1 podsumowuje wiadomości dotyczące zasięgu i okresu istnienia zmiennych stosowanych w poszczególnych obszarach pamięci.

Tabela 1.1. Zasięg i okres istnienia

	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

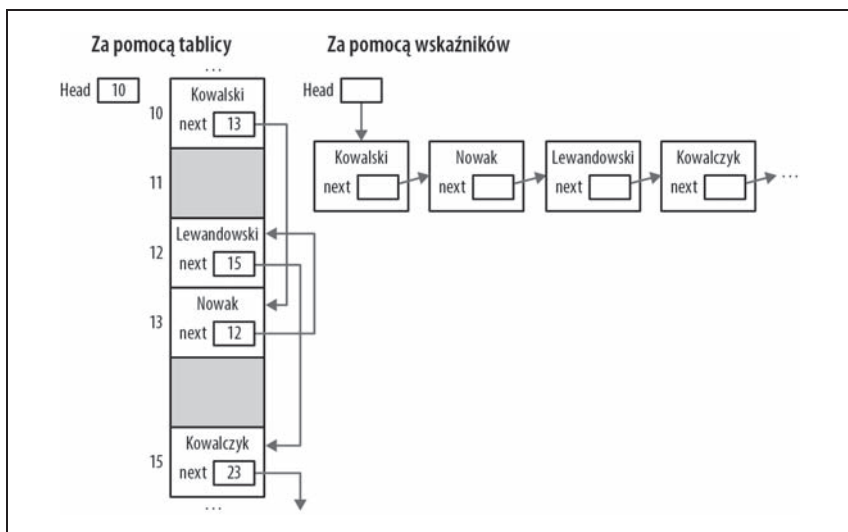
Wskaźniki można stosować do:

- 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 1.1 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.



Rysunek 1.1. Porównanie reprezentacji listy powiązanej za pomocą tablicy i wskaźników

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 „Wyluskiwanie wskaźnika za pomocą operatora adresowania pośredniego”. Obie funkcje `printf` dają ten sam rezultat — wyświetlają literę `w`. Prostszy 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 stacku, gdy ta już została wcześniej zwolniona;
- wyłuskiwanie wskaźnika przed alokowaniem go w pamięci.

Szczegółowe omówienie tych problemów znajdziesz w rozdziale 7.

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:

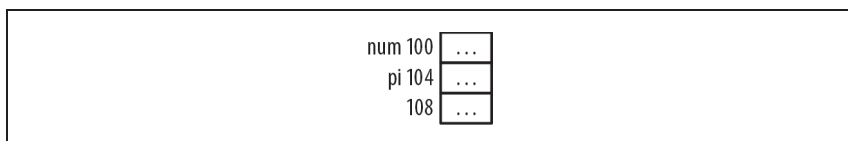
```
int* pi;  
int * pi;  
int *pi;  
int*pi;
```



Stosowanie znaku spacji jest indywidualną sprawą użytkownika.

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 1.2 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.



Rysunek 1.2. Schemat pamięci

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. Jak opisano w rozdziale 7., 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. Zagadnienie to opisano

szerzej w podrozdziale „Rodzaje wskaźników i ich rozmiary”. 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 (patrz rysunek 1.3).

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

Rysunek 1.3. Odczytywanie deklaracji

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 = &num;
```

Zmiennej `num` przypisano wartość zero, a zmienna `pi` ma wskazywać na adres zmiennej `num`. Ilustruje to poniższy rysunek.

num	100	0
pi	104	100
	108	...

Rysunek 1.4. Przyporządkowanie pamięci

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

```
int num;  
int *pi = &num;
```

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. Problematykę tę opisano szerzej w podrozdziale „Pojęcie braku wartości”.



Dobłą 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. Przedstawiono je w tabeli 1.2.

Tabela 1.2. 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);
```

Wskaźniki na `void` szerzej omówiono w podrozdziale „Wskaźniki na `void`”. Aby prezentowane przykłady były bardziej zrozumiałe, będziemy stosować specyfikator `%p` bez rzutowania adresu na wskaźnik na `void`.

Pamięć wirtualna i wskaźniki

W rzeczywistości wyświetlanie adresów wskaźników jest jeszcze bardziej złożone. Adresy wskaźników wyświetlane przez **wirtualny system operacyjny** prawdopodobnie nie będą ich prawdziwymi, fizycznymi adresami. Wirtualny system operacyjny pozwala na umieszczenie programu w fizycznej pamięci komputera po uprzednim podzieleniu go na kilka części. Aplikacja jest dzielona na strony (ramki). Strony odzwierciedlają obszary głównej pamięci komputera. Strony aplikacji są alokowane w różnych, niekoniecznie sąsiadujących ze sobą obszarach pamięci. Ponadto strony programu wcale nie muszą znajdować się jednocześnie w pamięci. Jeżeli system operacyjny potrzebuje pamięci obecnie zajmowanej przez ramkę, ramka może zostać przeniesiona do pamięci pomocniczej. Gdy ramka znów będzie potrzebna, system operacyjny może ją przenieść z powrotem do pamięci, nawet pod inny adres. Ta funkcja systemu operacyjnego sprawia, że zarządzanie pamięcią jest procesem bardzo elastycznym.

Każdy program zakłada potencjalną możliwość uzyskania dostępu do całej pamięci dostępnej fizycznie w komputerze. W rzeczywistości jest nieco inaczej. Program korzysta z adresów wirtualnych. Gdy zachodzi potrzeba, system operacyjny mapuje adresy wirtualne do pamięci fizycznej.

Oznacza to, że podczas wykonywania programu kod i dane przechowywane przez stronę mogą znajdować się w różnych fizycznych lokacjach. Wirtualne adresy aplikacji nie zmieniają się. Są to właśnie te adresy, które analizowaliśmy, przyglądając się zawartości wskaźników. System operacyjny mapuje adresy wirtualne na adresy rzeczywiste w sposób transparentny.

Jest to rzecz w pełni obsługiwana przez system operacyjny. Programista nie ma nad tym procesem kontroli, a także nie musi się nim przejmować. Zrozumienie tego zagadnienia pomoże w wyjaśnieniu adresów zwracanych przez program działający w wirtualnym systemie operacyjnym.

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 = &num;
```

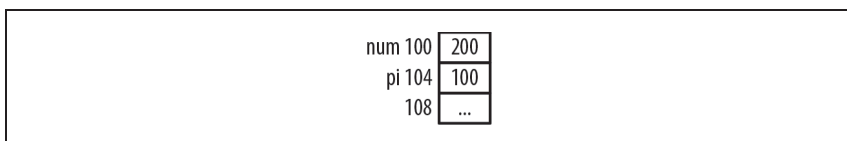
Zastosujmy operator adresowania pośredniego, aby wyświetlić 5 — wartość przechowywaną przez `num`:

```
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. Rysunek 1.5 ilustruje wpływ tej operacji na stan pamięci.

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



Rysunek 1.5. Przydzielenie pamięci podczas korzystania z operatora dereferencji

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)();
```

Wskaźniki na funkcję są obszernym tematem poruszonym w rozdziale 3.

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;
```



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  
}
```



Każde z dwóch zastosowanych wyrażen 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) ...
```



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 pożądaný 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 = &num;
*pi = 0; // zero odnosi się do elementu będącego liczbą całkowitą
```

Przyzwyczajiliśmy 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. W podrozdziale „Polimorfizm w języku C” ponownie przyjrzymy się zastosowaniu wskaźników na void w odniesieniu do zachowań polimorficznych.



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
```

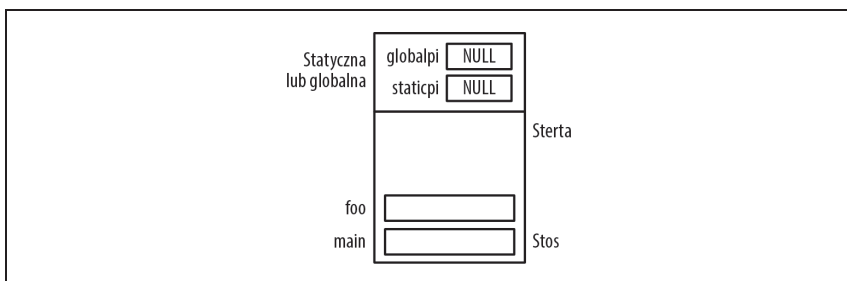
Typ size_t jest typem danych stosowanym do rozmiarów. Omówiono go w podrozdziale „Predefiniowane typy związane ze wskaźnikami”.

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 1.6 przedstawiono ułożenie tych wskaźników w pamięci. Ramki stosu są odkładane na stos, a sarta 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 oddzielnym od segmentu, w którym znajdują się stos i sarta. Sarta i stos programu zostaną szczegółowo omówione w rozdziale „Sarta i stos programu”.



Rysunek 1.6. Wskaźniki globalne i statyczne — alokacja pamięci

Rodzaje wskaźników i ich rozmiary

Rozmiar wskaźnika jest problemem, którym programista zaczyna się martwić, gdy musi zadbać o przenośność i kompatybilność aplikacji. W większości nowoczesnych platform rozmiar wskaźnika na dane jest zwykle identyczny, niezależny od typu wskaźnika. Wskaźnik do znaku ma taki sam rozmiar jak wskaźnik do struktury. Dzieje się tak nawet pomimo tego, że standard języka C nie określa tego, że rozmiary wskaźników mają być identyczne. Jednakże wskaźnik na funkcję może się różnić pod względem rozmiaru od wskaźnika na dane.

Rozmiar wskaźnika zależy od maszyny oraz kompilatora. Na przykład w nowych wersjach systemu Windows wskaźniki mają rozmiar 32 lub 64 bitów. W systemie DOS oraz Windows 3.1 wskaźniki miały długość 16 lub 32 bitów.

Modele pamięci

Wprowadzenie maszyn 64-bitowych sprawiło, że różnice w ilości pamięci alokowanej dla poszczególnych typów danych stały się zauważalne. Różne komputery i kompilatory oferują różne opcje alokowania danych. Poniżej zaprezentowano popularne notacje służące do opisu różnych modeli danych:

I In L Ln LL LLn P Pn

Każda wielka litera symbolizuje dane typu integer (I), dane typu long (L) lub wskaźnik (P). Małe litery symbolizują liczbę bitów alokowanych dla danego typu danych. Tabela 1.3 ilustruje to dokładniej. Podano w niej rozmiar w bitach.

Tabela 1.3. Komputery i modele pamięci

Typ danych (w języku C)	LP64	ILP64	LLP64	ILP32	LP32
char	8	8	8	8	8
short	16	16	16	16	16
_int32		32			
int	32	64	32	32	16
long	64	64	32	32	32
long long			64		
wskaźnik	64	64	64	32	32

Model zależy od systemu i kompilatora. System operacyjny może obsługiwać więcej niż jeden model. W takim przypadku możliwe jest zarządzanie przy użyciu odpowiednich opcji kompilatora.

Predefiniowane typy związane ze wskaźnikami

Istnieją cztery predefiniowane typy, które są często używane ze wskaźnikami:

`size_t`

Żapewnia rozmiarom bezpieczny typ.

`ptrdiff_t`

Obsługuje arytmetykę wskaźników.

`intptr_t` i `uintptr_t`

Stosowane do przechowywania adresów wskaźników.

W poniższych podrozdziałach przedstawiono zastosowanie każdego z wymienionych wyżej typów, z wyjątkiem `ptrdiff_t`, który zostanie omówiony w podrozdziale „Odejmnowanie wskaźników”.

Typ `size_t`

Typ `size_t` reprezentuje maksymalny rozmiar dowolnego obiektu istniejącego w języku C. Do reprezentacji stosowana jest liczba całkowita bez znaku. Rozmiar nie może być liczbą ujemną. Typ `size_t` służy do deklarowania rozmiaru zgodnie z dostępnym adresowalnym obszarem pamięci. Typ `size_t` jest stosowany jako typ zwracany dla operatora `sizeof`. Ponadto jest on argumentem dla wielu funkcji, takich jak np. `malloc` i `strlen`.



Dobłą praktyką jest stosowanie `size_t` podczas deklaracji zmiennych określających rozmiary liczb, łańcuchów znaków oraz tablic. Typ ten powinien być stosowany do liczników pętli, indeksowania tablic, a czasami także do arytmetyki wskaźnikowej.

Deklaracja `size_t` jest określana przez implementację. Można ją odnaleźć w wielu standardowych nagłówkach, takich jak `stdio.h` i `stdlib.h`. Zwykle jest ona definiowana w następujący sposób:

```
#ifndef __SIZE_T
#define __SIZE_T
typedef unsigned int size_t;
#endif
```

Dyrektywa `define` zapewnia to, że deklaracja będzie wyłącznie jednokrotna. Rzeczywisty rozmiar będzie zależał od implementacji. W przypadku systemów 32-bitowych `size_t` będzie miał zwykle długość 32 bitów, a w przypadku systemów 64-bitowych długość ta wyniesie 64 bity. Zwykle najwyższą wartością `size_t` jest `SIZE_MAX`.



Typ `size_t` może być zwykle stosowany do przechowywania wskaźnika. Nie można jednakże zakładać, że `size_t` będzie takiego samego rozmiaru jak wskaźnik. W kolejnym podrozdziale dowiesz się, że lepiej w tym celu skorzystać z `intptr_t`.

Musisz być ostrożny podczas wyświetlania wartości zdefiniowanych jako `size_t`. Są to wartości bez znaku (typu `unsigned`), a więc jeżeli wybierzesz nieprawidłowy specyfikator formatu, wyświetlisz nieprawidłowe dane. Zalecany specyfikatorem formatu jest `%zu`, jednakże nie zawsze jest on dostępny. Dopuszcza się również stosowanie specyfikatorów `%u` lub `%lu`.

Przyjrzyj się poniższemu przykładowi. Najpierw zadeklarowano w nim zmienną jako `size_t`, a następnie wyświetlono ją przy użyciu dwóch różnych specyfikatorów:

```
size_t sizet = -5;
printf("%d\n", sizet);
printf("%zu\n", sizet);
```

Zmienna typu `size_t` powinna być używana do dodatnich liczb całkowitych. Stosowanie wartości ujemnej może prowadzić do problemów. Gdy przypiszemy liczbę ujemną do `size_t`, a następnie ją wyświetlimy, korzystając ze specyfikatorów `%d` i `%zu`, otrzymamy następujące wyniki:

```
-5
4294967291
```

Korzystając ze specyfikatora `%d`, interpretujemy `size_t` jako liczbę całkowitą ze znakiem. Wyświetlono `-5`, ponieważ `size_t` przechowuje `-5`. Specyfikator `%zu` interpretuje zawartość `size_t` jako liczbę całkowitą bez znaku. Podczas interpretowania liczby `-5` jako liczby całkowitej bez znaku najbardziej znaczący bit tej liczby jest zmieniany na jedynkę, co jest interpretowane jako liczba 2 podniesiona do wysokiej potęgi. Z tego powodu po użyciu specyfikatora `%zu` wyświetlona została tak duża wartość.

Liczba dodatnia przy użyciu obu specyfikatorów zostanie wyświetlona poprawnie:

```
size_t = 5;
printf("%d\n",size_t); // Wyświetli 5
printf("%zu\n",size_t); // Wyświetli 5
```

Zmiennej typu `size_t` przypisuj tylko liczby dodatnie.

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
```



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

Wskaźniki na funkcje mogą mieć różne rozmiary. Zwykle ten rozmiar jest określony dla danej kombinacji systemu operacyjnego i kompilatora. Wiele kompilatorów obsługuje zarówno aplikacje 32-bitowe, jak i 64-bitowe. Możliwa jest więc sytuacja, w której ten sam program, skompilowany w inaczej skonfigurowanym kompilatorze, będzie zawierał wskaźniki o różnych rozmiarach.

W architekturze harwardzkiej kod programu i dane są przechowywane w fizycznie oddzielnych pamięciach. Przykładem układu o takiej architekturze jest mikrokontroler Intel MCS-51 (8051). Co prawda Intel zaprzestał produkcji tego układu, ale spotykanych i nadal produkowanych jest wiele układów binarnie z nim kompatybilnych. Kompilator Small Device C Compiler (w skrócie SDCC) obsługuje ten rodzaj mikroukładów. Procesor ten obsługuje wskaźniki o rozmiarze od 1 do 4 bajtów. Zależnie od potrzeby rozmiar wskaźnika musi zostać zdefiniowany, ponieważ w tym środowisku wskaźniki nie mają stałego rozmiaru.

Typy `intptr_t` i `uintptr_t`

Typy `intptr_t` i `uintptr_t` stosuje się do przechowywania adresów wskaźników. Dzięki nim można wygodnie i bezpiecznie deklarować wskaźniki. Mają one rozmiary identyczne z podstawowym rozmiarem wskaźnika w danym systemie. Można je stosować do konwersji wskaźnika na jego reprezentację w postaci obiektu typu `integer`.

Typ `uintptr_t` jest bezznakową wersją `intptr_t`. Typ `intptr_t` jest preferowany w przypadku większości operacji. Typ `uintptr_t` nie jest tak elastyczny jak `intptr_t`. Poniższy przykład pokazuje sposób użycia `intptr_t`:

```
int num;  
intptr_t *pi = &num;
```

Jeżeli spróbujesz przypisać (tak jak pokazano poniżej) adres obiektu typu `integer` do wskaźnika typu `uintptr_t`, kompilator wyświetli informację o błędzie składni:

```
uintptr_t *pu = &num;
```

Treść komunikatu o błędzie konwersji będzie następująca:

```
error: invalid conversion from 'int*' to  
      'uintptr_t* {aka unsigned int*}' [-fpermissive]
```

Aby uniknąć komunikatu o błędzie, musisz dokonać przypisania za pomocą rzutowania:

```
intptr_t *pi = &num;  
uintptr_t *pu = (uintptr_t*)&num;
```

Nie możesz stosować `uintptr_t` z innymi typami danych bez rzutowania:

```
char c;  
uintptr_t *pc = (uintptr_t*)&c;
```

Powinieneś stosować omówione właśnie typy, jeżeli jesteś przezorny i zależy Ci na kompatybilności. Nie będziemy ich jednakże stosować w wyjaśnieniach, ponieważ mogłoby to niepotrzebnie zagmatwać.



Unikaj rzutowania wskaźnika na obiekt typu `integer`. Jeżeli wskaźniki są 64-bitowe, a obiekty typu `integer` tylko 4-bitowe, nastąpi utrata danych.



Wczesne procesory firmy Intel posiadały 16-bitową architekturę segmentową. Stosowano tam wskaźniki bliskie i dalekie. Wskaźniki te nie są stosowane w nowoczesnych komputerach o architekturze pamięci wirtualnej. Omawiane wskaźniki stanowiły rozszerzenie standardu języka C. Służyły one do obsługi segmentowej architektury wczesnych procesorów firmy Intel. Wskaźniki bliskie mogły jednorazowo zaadresować około 64 KB pamięci. Wskaźniki dalekie mogły adresować do 1 MB pamięci, ale były wolniejsze od wskaźników bliskich. Wskaźniki ogromne były znormalizowanymi wskaźnikami dalekimi. Ich adresy wykorzystywały najwyższy dostępny segment.

Operatory wskaźników

Istnieje kilka operatorów, których można używać podczas pracy ze wskaźnikami. Wcześniej omówiliśmy operatory wyluskiwania i uzyskiwania adresu. W tym podrozdziale omówimy porównywanie wskaźników oraz działania arytmetyczne przeprowadzane przy użyciu wskaźników. Tabela 1.4 prezentuje operatory wskaźników.

Tabela 1.4. Operatory wskaźników

Operator	Nazwa	Funkcja
*		deklaracja wskaźnika
*	operator dereferencji	wyluskiwanie wskaźnika
->	operator odwołania	uzyskiwanie dostępu do półstruktur wskazywanych przez wskaźnik
+	dodawanie	inkrementacja wskaźnika
-	odejmowanie	dekrementacja wskaźnika
==, !=	równość, nierówność	porównywanie dwóch wskaźników
>, >=, <, <=	wiekszy 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,
- odejęcie 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. Szerzej problem ten omówiono w podrozdziale „Modele pamięci”. Tabela 1.5 prezentuje często spotykane rozmiary. O ile nie zaznaczono inaczej, wartości te będą używane w przedstawionych w tej książce przykładach.

Tabela 1.5. Typy danych i ich 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 jedyneką dodaną do `pi` liczba cztery zostaje dodana do adresu. Rysunek 1.7 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
```



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

vector[0]	100	28
vector[1]	104	41
vector[2]	108	7
pi	112	100

Rysunek 1.7. Vector — alokacja pamięci

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;
```

Żałujemy, że pamięć została alokowana w sposób przedstawiony na rysunku 1.8. Wszystkie zastosowane tu adresy mieszczą się w granicy czterech bajtów. Realne adresy mogą być przyporządkowane w innych granicach i w innej kolejności.

s	120	...
ps	124	120
c	128	...
pc	132	128

Rysunek 1.8. Wskaźniki na `short` i `char`

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, używane 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 1.7.

```
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 1.9 przedstawiono alokację pamięci.

vector[0]	100	28
vector[1]	104	41
vector[2]	108	7
p0	112	100
p0	116	104
p0	120	108

Rysunek 1.9. Odejmowanie dwóch wskaźników

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 przykładu, na którym pracowaliśmy w podrozdziale „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. W tym podrozdziale 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`. Zagadnienie to zostanie dokładniej omówione w rozdziale 8.

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",
                 "Podróż 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.

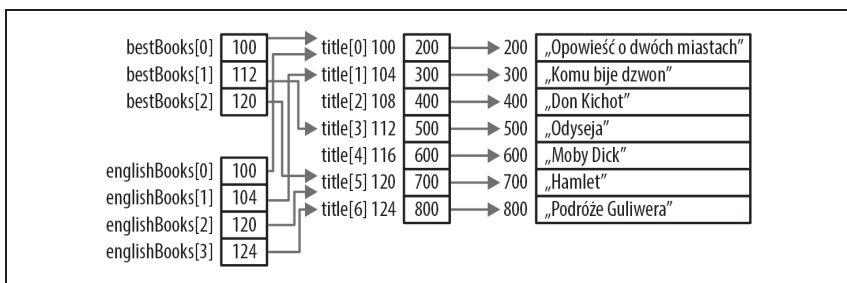
Poniżej przedstawiono deklaracje omówionych tablic. Każdy element tablicy zawiera wskaźnik, który wskazuje na drugi wskaźnik na `char`.

```
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 1.10 pokazuje alokację pamięci dla powyższego przykładu.



Rysunek 1.10. Wskaźniki do wskaźników

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. W rozdziałach 3. i 5. dowiesz się, jak takie wskaźniki mogą chronić parametry funkcji przed modyfikacją.

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 = &num;
pci = &limit;
```

Sytuację tę ilustruje rysunek 1.11.

num 100	5
limit 104	500
pi 108	100
pci 112	104

Rysunek 1.11. Wskaźnik na stałą typu integer

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 dozwolony, 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 = &num;
```

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

Przeanalizuj poniższe przypisanie:

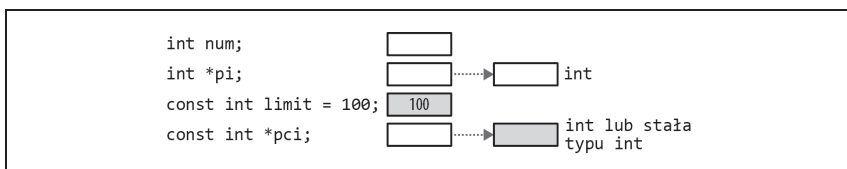
```
*pci = 200;
```

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.

Rysunek 1.12 ilustruje pojęcie wskaźnika na stałą. Puste pola symbolizują zmienne, które można modyfikować. Zaciemnione pola symbolizują zmienne, których nie można modyfikować. Zaciemnione 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.



Rysunek 1.12. Wskaźnik na stałą

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.



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

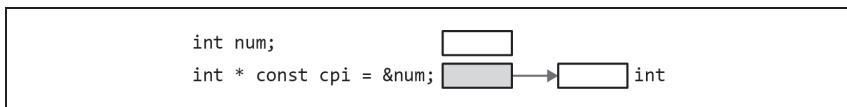
Istnieje możliwość zadeklarowania wskaźnika na stałą na obiekt niebędący stałą. Gdy dokonamy takiej deklaracji, nie będziemy mogli zmienić wskaźnika, ale uzyskamy możliwość modyfikacji wskazywanych danych. Przykładem takiego wskaźnika jest:

```
int num;
int *const cpi = &num;
```

Po takiej deklaracji:

- cpi musi być zainicjowane zmienną niebędącą stałą;
- cpi nie może być modyfikowane;
- dane, na które wskazuje cpi, mogą być modyfikowane.

Na rysunku 1.13 przedstawiono wskaźnik tego typu.



Rysunek 1.13. Wskaźniki typu *constant* stosowane do obiektów niebędących stałymi

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 = &num;
cpi = &age;
```

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

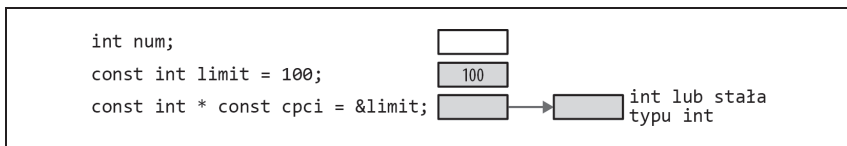
```
'cpi' : you cannot assign to a variable that is const
```

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 1.14.



Rysunek 1.14. Wskaźnik typu constant wskazujący na stałą

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 = &num;
```

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 = &num;
```

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;
```

```
'cpci' : you cannot assign to a variable that is const
expression must be a modifiable lvalue
```

Wskaźniki na (wskaźniki typu constant na stałe)

```
const int * const cpci = &limit;
const int * const * pcpci;
```

```
const int limit = 500;
const int * const cpci = &limit;
const int * const * pcpci = &cpci;
```

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);
```

Tabela 1.6. Typy 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

Podsumowanie

Omówiliśmy podstawowe wiadomości dotyczące wskaźników, takie jak deklarowanie wskaźników i ich stosowanie w typowych sytuacjach. Wyjaśniliśmy także pojęcie znaku `null` i jego różne warianty. Przyjrzeliliśmy się też wielu operatorom wskaźników.

Dowiedziałeś się, że wskaźniki mogą mieć różne rozmiary, w zależności od modelu pamięci obsługiwanego przez docelowy system i kompilator. Dowiedziałeś się także, jak ze wskaźnikami można stosować słowo klucz `const`.

Po opanowaniu tych podstaw jesteś przygotowany na odkrywanie kolejnych obszarów, w których przydaje się stosowanie wskaźników. W kolejnych rozdziałach opiszemy przekazywanie wskaźników do funkcji, obsługę struktur danych, a także dowiesz się, jak stosować wskaźniki do obsługi dynamicznej alokacji pamięci. Ponadto przeczytasz, jak zabezpieczać aplikację za pomocą wskaźników.

Skorowidz

A

- adres
 - buforu, 115
 - elementu tablicy, 121, 126
 - literału
 - zwracanie, 158
 - literału łańcuchowego
 - nieprawidłowe przypisanie, 142
 - kopiowanie do wskaźnika, 143
 - łańcucha
 - przypisanie, 147
 - zwracanie, 158
 - niepoprawny, 21
 - operator, 23
 - pamięci, 22
 - adresowanej
 - dynamicznie, 160
 - utrata, 56
 - powrotny, 81, 82
 - specjalnego przeznaczenia, 217
 - tablicy, 110
 - vector, 92
 - wirtualny, 26
 - wskaźnika, 21
 - wyświetlanie, 25
 - zerowy, 24, 218
 - zmiennej, 24
 - num, 23
- adresowanie
 - pamięci graficznej, 218
 - pośrednie drugiego poziomu, 121
 - zerowej lokacji pamięci, 217
- algorytmy wzajemnego wykluczenia, 227
- aliasing, 69
 - wskaźników, 221
- alokacja
 - ciągła, 128
 - łańcuchów, 144
 - nieciągła, 129
 - sposób, 129
 - stron aplikacji, 26
 - tablicy, 106, 107
 - zmiennych struktury, 237
- alokacja pamięci
 - a czyszczenie pamięci, 22
 - argumenty argc i argv, 158
 - dla funkcji
 - getline, 116
 - stringLength, 154
 - dla łańcucha, 64, 138
 - znaków, 55
 - dla obiektu typu integer, 53
 - dla zmiennych
 - automatycznych, 51
 - dynamicznej, 58
 - na ramce stosu funkcji, 66
 - nadzorowanie, 55
 - niepowodzenie, 60
 - niewłaściwe indeksy tablicowe, 202
 - o ciągłym obszarze, 129
 - o potencjalnie nieciągłym obszarze, 129
 - określenie ilości, 61
 - polimorfizm i dziedziczenie, 237
 - porządek bajtów, 221
 - pula literalów łańcuchowych, 140
 - rzutowanie zmiennej typu integer, 216
 - struktura person, 171
 - zainicjalizowana, 172
- struktury, 169, 240
- tablica, 90
 - dwuwymiarowa, 123, 132
 - na element typu char, 66
 - postrzępiona, 134
 - trójwymiarowa, 128
- uzyskiwanie dostępu do portu, 220
- vector, 39
- wskaźniki
 - dzikie, 198
 - globalne i statyczne, 32
 - na person, 173
 - niedopasowane typy, 204
- wycieki pamięci, 93
- zwracanie łańcucha alokowanego dynamicznie, 161
- alokator
 - Hoard, 75
 - ogólnego przeznaczenia, 75
- analiza statyczna narzędzia, 212
- anulowanie dereferencji, 111
- aplikacja
 - kontrola procesu wykonywania, 209
 - nieprawidłowe zakończenie, 70

- aplikacja
 - nieprzewidywalne zachowanie, 195
 - przekazywanie parametrów, 157
 - przeniesienia kontroli w kodzie, 200
 - rozmiar wskaźnika, 32
- architektura harwardzka, 35
- argument
 - ujemny, 58
 - przekazywanie do aplikacji, 157
 - rozmiar łańcucha, 203
- arytmetyka wskaźnikowa, 37
 - i struktury, 207
 - typ danych, 34
- ASCII
 - znak NUL, 28
- ASLR, 196
- ataki
 - denial of service, 211
 - format string attack, 207
 - nadpisanie adresu wskaźnika na funkcję, 210
 - nadpisywanie złośliwym kodem, 196
 - naruszenie ochrony pamięci, 200
 - return-to-libc, 196

B

- big-endian, 216, 220
- binarne drzewo poszukiwań, 190
- blok
 - finally, 77
 - try, 77
- instrukcji w języku C, 83
- błąd
 - inicjalizacji, 144
 - kompilacji
 - modyfikacja łańcucha, 140

- składni
 - dereferencja pci, 46
 - dodawanie, 40
 - inicjacja wskaźnika, 48
 - konwersja, 24, 36
 - kopiowanie łańcuchów, 152
 - niedopasowanie parametrów, 87
 - porównanie łańcuchów, 147
 - przekazanie ardesu literału całkowitego, 87
 - przypisanie cpci nowego adresu, 48
 - przypisanie nowej wartości, 48
- błędne odwołanie do pamięci, 60
- Boehm-Weiser Collector, 76
- Bounded Model Checking, 205
- brak wartości, 28
- bufor, 75, 114
 - kopiowanie łańcuchów, 150
 - przekazywanie, 155
 - przepelnienie, 199
 - czynniki, 200
 - informacja o błędzie, 206
 - wewnątrz przestrzeni adresowej, 200
- rozmiar, 115
- przekroczenie, 116

C

- CBMC, 205
- CERT, 195
- CERT Secure Coding, 20
- cpci, 48
- cpi, 47
- czyszczenie danych wrażliwych, 211

D

- dane
 - bezużyteczne, 22
 - little-endian, 204
 - modyfikowalne, 84
 - o rozmiarze bloku, 54
 - obudowywanie, 233
 - przekazywanie
 - przez wartość, 85
 - wskaźniki, 84
 - struktura, 54
 - o zmiennej liczbie elementów, 52
 - typu
 - char, 39
 - integer, 32
 - long, 32
 - short, 39
 - wrażliwe, 211
 - zapisywanie na porcie, 219
- dealokacja
 - pamięci, 66
 - automatyczna, 76
 - inicjowanie przy pozyskaniu zasobu, 76
 - procedura obsługi wyjątków, 77
 - wartości wskaźnika, 20
- definicje
 - makr, 197
 - typów, 197
 - typu struktury, 168
- deklaracja
 - pmatrix, 123
 - size_t, 34
- deklarowanie
 - rozmiaru obiektu, 33
 - struktury, 233
 - wskaźników, 20
 - na funkcję, 27, 79, 96
 - nazwa definicji typu, 98
 - zmiennych, 197
- DEP, 196
- dereferencja, 26
- DMA, 220

- dmalloc, 75
- dostęp
 - do adresu
 - specjalnego
 - przeznaczenia, 217
 - zerowego, 218
 - do obiektu odniesienia, 84
 - do pamięci
 - bezpośredni, 220
 - DMA, 220
 - poza granicami
 - tablicy, 202
 - ulotnej, 220
 - zwolnionej, 54
 - do portu, 219
 - do wewnętrznej
 - struktury danych, 237
- drzewo, 176, 190
 - binarne, 176, 190
 - korzeń, 190, 191
 - logiczna organizacja, 192
 - metody przeglądania, 194
 - obsługa wyrażań
 - arytmetycznych, 194
 - posortowana liczba
 - elementów, 194
 - przeglądanie, 192
 - puste, 191
 - TreeNode, 192
 - węzły
 - dodawanie, 190
 - zarządzanie elementami, 190
- dynamiczna alokacja
 - pamięci, 19, 52
 - dla tablicy, 106
 - funkcje, 57
 - kroki, 52
 - łańcuchy, 142
 - sterta, 81
 - tablicy dwuwymiarowej, 128
 - techniki, 75
- dyrektywa
 - define, 34
- dziedziczenie, 237
 - struktury, 237

E

- element
 - head, 177
 - names, 19
 - tail, 177
 - typu integer, 20

F

- FILO, 188
- format string attack, 207
- funkcja
 - a ramka stosu, 81
 - a stos programu, 79
 - add, 99
 - addHead, 179, 180
 - zastosowanie, 180, 181
 - addNode, 234
 - addTail, 179, 181
 - alloca, 66
 - allocateArray, 89
 - puste wskaźniki, 91
 - alokująca, 17
 - assert
 - inicjalizacja
 - wskaźników, 198
 - umiejscowienie, 199
 - blanks, 160
 - calloc, 58
 - stosowanie, 62
 - cfree, 63
 - compare, 163, 182
 - compareIgnoreCase, 163
 - compute, 99
 - deallocatePerson, 173, 175
 - delete, 179, 182
 - dequeue, 186
 - displayLinkedList, 179, 184
 - DMA, 220
 - dotProduct, 229
 - evaluate, 100
 - evaluateArray, 101
 - factorial, 231
 - format, 156
 - fptrDisplay, 238

- fptrSet, 238
- free, 19, 52, 58, 66
 - dublowanie, 211
 - narzut pamięci po
 - strukturach, 174
 - podwójne
 - uwalnianie, 69
 - stan programu, 72
 - tworzenie własnej, 93
 - wywoływanie, 54
 - zastąpienie, 74
- getArea, 242
- getLinkedListInstance, 234
- getNode, 179, 182
- getPerson, 174
- getRectangleInstance, 240
- gets
 - przepelenienie
 - buforu, 206
- getShapeInstance, 239
- getSystemStatus, 209
- initializeList, 179
 - kolejki, 185
 - zastosowanie, 180
- initializePerson, 173
- initializeQueue, 185
- initializeStack, 188
- insertNode, 190
- drzewo, 192
- konsekwencje, 10
- main, 43, 53, 157
- malloc, 10, 17, 33, 52, 58
 - alokacja pamięci
 - ciągłej, 130
 - alokacja pamięci
 - zwracanej, 88
 - maksymalny rozmiar
 - pamięci, 61
 - narzut pamięci po
 - strukturach, 174
 - określanie długości
 - łańcucha, 143
 - przekazywanie
 - wskaźnika do
 - wskaźnika, 93
 - stosowanie, 58

- ul>
- funkcja
 - malloc
 - stosowanie złego rozmiaru, 61
 - tworzenie tablic jednowymiarowych, 113
 - unikanie narzutu struktur, 174
 - wskaźniki globalne i statyczne, 62
 - wykrywanie wartości zerowej, 200
 - malloca, 66
 - memset, 63, 218
 - notacja wskaźnikowa, 119
 - obliczanie silni, 231
 - obsługa list
 - powiązanych, 179
 - porównująca, 177
 - printf, 19, 25, 82
 - przepełnienie buforu, 200, 206
 - przypisanie do wskaźnika na funkcję, 210
 - realloc, 58
 - działanie, 64
 - obsługa tablic, 107
 - pamięć zajmowana przez wskaźnik, 117
 - przetrzykiwanie pamięci, 116
 - przykład działania, 117
 - stosowanie, 63
 - zmiana rozmiaru tablicy, 114
 - rekursywna, 193
 - removeLinked
 - ListInstance, 234
 - removeNode, 234, 235
 - replace, 203
 - returnPerson, 175
 - saferFree, 94
 - scanf_s, 206
 - select, 99
 - Sleep, 232
 - snprintf, 156
 - sort, 163
 - specyfikatora, 25
 - sprawdzająca poprawność wskaźnika, 205
 - sptrGet, 238
 - square, 97
 - przypisanie adresu do wskaźnika, 97
 - umiejscowienie, 98
 - srtlcat, 206
 - startThread, 232
 - strcat
 - łączenie łańcuchów, 149
 - strcat_s, 206
 - strcmp
 - porównywanie łańcuchów, 145
 - strcpy, 141
 - kopiowanie łańcuchów, 147
 - przepełnienie buforu, 203
 - strcpy_s, 206
 - stringToLower, 163
 - strncpy, 206
 - strlen, 33, 143
 - strncat
 - błędy, 206
 - strncpy
 - błędy, 206
 - substracy, 99
 - subtract, 99
 - swap, 84
 - stos programu, 85
 - trim, 117
 - uwaniania listy powiązanej, 235
 - wielowątkowa, 228
 - wscanf_s, 206
 - wyświetlanie zawartości tablicy trójwymiarowej, 127
 - z zewnątrz
 - implementacja struktury danych, 233
 - zamień, 84
 - zarządzanie pamięcią dynamiczną, 57
 - zwracanie łańcucha, 159
 - wskaźnika, 84, 97
 - zwrotna, 220
- G**
- GCC
 - modyfikacja literału łańcucha, 140
 - opcje strict aliasing, 222
- H**
- head, 177
- homogeniczność elementów, 106
- I**
- iloczyn skalarny, 228
- implementacja
 - kolejek i stosów, 176
 - wskaźnika, 22
- indeks, 106
 - nieprawidłowy, 107
 - podwójny, 127
 - pojedynczy, 107
 - znakowy, 100
- indeksowanie tablic
 - typ danych, 34
- inicjowanie
 - przy pozyskaniu zasobu, 76
- instrukcja
 - blokowa
 - problemy, 73
 - printf, 41, 126
 - pusta, 28
- Intel MCS-51, 35
- interpretowanie deklaracji, 22

J

język C
 łańcuch, 28
 struktura, 10
 zabezpieczenia aplikacji,
 195
 język C++
 alokacja i dealokacja
 zasobów, 76

K

kod
 błędu POSIX, 63
 korzystanie z
 przykładów, 14
 kolejki, 19, 52, 176, 185
 dodanie elementu, 185
 FIFO, 185
 implementacja, 185
 instrukcja definiująca
 typ, 185
 jednoelementowa, 186
 oparta na liście
 powiązanej, 185
 operacja
 odłączania, 185
 dołączania, 185
 pusta, 186
 uwalnianie węzła, 187
 wieloelementowa, 186
 kolejność rzędowo-
 kolumnowa, 123
 kolumny, 106
 kompilator GCC, 75
 komunikat
 0x0, 93
 kontrola
 toku wykonywania
 programu, 79
 konwencje typograficzne, 13
 konwersja
 typów wskaźników, 223
 wskaźników, 36, 102
 korzeń, 190

L

libc, 196
 liczba
 zmiennoprzecinkowa,
 224
 liczniki pętli
 typ danych, 34
 lista
 alokowanych struktur,
 174
 inicjalizująca, 132
 powiązana, 18, 19, 52, 176
 alokacja pamięci dla
 nowego elementu,
 181
 alokacja pamięci dla
 węzła, 180
 cykliczna, 178
 dwukierunkowa, 178
 identyfikacja
 elementu, 182
 implementacja, 234
 inicjalizacja, 179, 180
 jednostronna, 177
 kasowanie
 elementów, 182
 nowy węzeł, 181
 porównywanie, 182
 przeglądanie, 184
 reprezentacja za
 pomocą tablicy, 18
 reprezentacja za
 pomocą
 wskaźników, 19
 schemat budowy, 178
 stos i sterta, 184
 struktura obsługi, 179
 struktura Node, 179
 usunięcia węzła, 188
 utrzymanie
 integralności, 186
 uwolnienie, 236
 wprowadzanie
 danych do
 elementów, 179

wskaźniki
 nieprzeźroczyste,
 233
 zwalnianie
 elementów, 189

struktur
 alokowanie pamięci,
 175

liście, 190

literal

łańcuchowy, 139
 łączenie łańcuchów,
 152
 niebędący stałą, 140
 przechowywanie w
 pamięci, 139
 statyczny, 159
 złożony, 132
 tablica postrzępiona,
 134
 znakowy, 139
 modyfikacja, 140
 little-endian, 216, 220
 losowy rozkład przestrzeni
 adresowej, 196
 lvalue, 27, 112

Ł

łańcuchy, 138
 alokacja, 137
 alokowane
 na stercie, 144
 w pamięci globalnej,
 144
 bajtów, 138
 deklaracja, 139
 informacje o
 podzespołach, 155
 inicjalizacja, 141, 143
 standardowe wejście,
 144
 jako wskaźniki na stałe
 typu char, 146
 kopiowanie, 147, 151
 lokalizacja, 144
 łączenie, 149
 łańcuchy źródłowe, 152
 popelaniane błędy, 152

- łańcuchy
 - porównywanie, 145, 163
 - przechowywanie, 138
 - przekazywanie, 137, 153
 - proste, 153
 - puste, 28
 - sortowanie w kolejności alfabetycznej, 146
 - standardowe operacje, 145
 - statyczne, 144
 - szerokich znaków, 138
 - umiejscowienie funkcji, 138
 - umieszczanie w pulach
 - wyłączenie, 140
 - w języku C, 137
 - zabezpieczenie przed modyfikacją, 137
 - zapisywanie w tablicy, 147
 - zwracanie, 137, 158

M

- macierz, 106
- magazyn
 - danych lokalnych, 81
 - parametrów, 81
- makro
 - NULL, 28
 - RAIL_VARIABLE, 76
 - kod, 77
- makroinstrukcja
 - safeFree, 94
- malloc, 75
- mapowanie
 - adresów wirtualnych na rzeczywiste, 26
- pamięci
 - notacja tablicy
 - dwuwymiarowej, 131
- menedżer sterty, 54
 - dodatkowa pamięć, 55
- funkcja
 - free, 70
 - realloc, 65

- nadzorowanie alokacji
 - pamięci, 55
 - podwójne uwalnianie, 69
 - źródło pamięci, 75
- metoda
 - inorder, 193
 - postorder, 193
 - preorder, 193
 - staticFormat, 160
- model konceptualny
 - stosu i sterty, 80
- modele
 - danych, 32
 - pamięci, 10, 32
- muteksy, 227
 - kod inicjalizujący, 230
 - ochrona zmiennej, 229
 - zablokowanie, 229

N

- naruszenie ochrony
 - pamięci, 71, 200
- nazwa
 - tablicy, 39
 - struktur, 168
- nieautoryzowany dostęp, 200
- niepoprawny adres, 21
- nieużytek, 76
- notacja
 - kropkowa, 168
 - tablicowa, 105, 109, 111
 - przekazywanie tablicy wielowymiarowej, 125
 - stosowanie, 118
 - tablica wskaźników, 120
 - wewnątrz funkcji, 119
 - tablicy dwuwymiarowej, 124
- vector, 112
- wskaźnikowa, 105, 109, 111
 - dostęp do elementu tablicy, 124

- i tablice, 109
- przekazywanie tablicy wielowymiarowej, 125
 - stosowanie, 119
- tablica wskaźników, 121
- NUL
 - a NULL, 138

O

- obiekt
 - alokowanie w pamięci, 17
 - person, 168
 - replacemen, 203
 - typu integer, 20
 - wartość lewostronna, 112
- obsługa
 - struktur danych, 176
 - tablic o zmiennej długości, 52
 - wątków, 228
 - wyjatków, 77
 - wywołań zwrotnych, 231
- odczytywanie deklaracji, 23
- odejmowanie wskaźników, 41
- odmowa usługi, 211
- odzyskiwanie pamięci, 70
- opaque pointer, 215, 233
- opcja
 - fno-strict-aliasing, 222
 - fstrict-aliasing, 222
 - Wall, 212
 - Wstrict-aliasing, 222
- opcode, 99
- OpenBSD, 75
- operacje
 - dodawanie liczby całkowitej, 38
 - odejmowanie liczby całkowitej, 40
 - przypisania, 29
- operator
 - adresowania, 110
 - pośredniego, 26
 - adresu, 23

- dereferencji, 27
 - błędy podczas stosowania, 54
 - priorytet, 114
- indeksu, 43
- nierówności, 101
- porównania, 42, 102
 - łańcuchów, 147
- równości, 101
- rzutowania
 - określanie porządku bajtów, 220
- sizeof, 31, 33, 53, 66
 - ilość bajtów do alokacji, 61
- niewłaściwe
 - stosowanie, 203
- określenie liczby elementów tablicy, 119
- stosowanie ze
 - wskaźnikami, 35
- tablica a wskaźnik, 112
- używanie, 236
 - z tablicą, 107
- wskazujący, 168
- wyłuskiwania
 - do odejmowania dwóch liczb, 42
- niewłaściwe
 - stosowanie, 201
- operatory wskaźników, 37
- optymalizacja kodu, 226
- organizacja pamięci, 15

P

- pamięć
 - alokowana
 - dla wskaźnika, 63
 - na stercie, 16
 - automatyczna, 16
 - dynamiczna, 16
 - globalna, 16
 - modele, 32
 - niezainicjowana, 21
 - organizacja przestrzeni danych, 196

- poza granicami tablicy
 - problemy, 202
- przydzielenie
 - operator dereferencji, 27
- przyporządkowanie, 23
- pula literałów, 139
- ramka stosu, 81
- ręczne sterowanie, 76
- sprzątanie, 76
- statyczna, 16
- systemowa, 70
- tablica, 113
- wirtualna, 26
- wypełnianie
 - zerami, 63
- zarządzanie, 51
- zerowanie, 218
- parametr
 - argc, 43, 157
 - argv, 43, 157
 - elementSize, 62
 - numElements, 62
 - przekazywanie, 84
- pci, 45
 - jako wskaźnik na stałą typu integer, 46
- pętla while, 115
- zmienna tmp, 117
- podtablica, 122
- podwójne uwalnianie, 68
- problemy, 211
- podwójne wskaźniki, 43
- pole
 - data, 180
 - length, 228
 - next, 18, 180
 - subCode, 159
 - sum, 228
- polecenie
 - malloc
 - zwracanie wskaźnika, 88
 - return, 88
- polimorfizm, 237
- porównywanie
 - elementów tablicy
 - funkcja sortowania, 162
 - wskaźników, 42

- port, 219
 - sprzętowy, 219
- Portable Operating System Interface, 228
- porządek bajtów maszyny, 216
 - określanie, 220
- POSIX, 228
- potomkowie, 190
- problemy
 - aplikacji
 - wielowątkowych, 227
 - arytmetyka
 - wskaźnikowa, 207
 - dealokacja pamięci, 211
 - łańcuchy, 206
 - stosowanie wskaźników, 199
 - użycie
 - niezainicjalizowanego wskaźnika, 198
 - wskaźniki na funkcję, 209
- programowanie, 10
 - inicjowanie wskaźnika, 24
- prototyp funkcji, 96
- przechowywanie
 - adresów wskaźnika, 36
 - odniesień, 30
- przedwczesne uwolnienie, 71
- przekazywanie
 - argumentów do aplikacji, 157
 - funkcji sterującej
 - porównaniem, 163
 - łańcuchów, 153
 - obiektu do funkcji, 88
 - poprzez wartość, 86
- tablicy
 - jednowymiarowej, 118
 - mniejszy rozmiar, 119
 - wielowymiarowej, 125
- wskaźnika, 94
 - do argumentu, 84
 - do funkcji, 79, 84, 91
 - do obiektu, 84
 - do stałej, 86
 - do tablicy liczb całkowitych, 91

- przekazywanie
 - wskaźnika
 - do wskaźnika, 91
 - list powiązanych, 234
 - na funkcję, 99
 - na łańcuch, 155
 - pustego, 91
 - wymagającego
 - inicjalizacji, 155
 - znaku NULL jako
 - adresu buforu, 156
- przepelnienie stosu, 83
- przestrzeń
 - większa od alokowanej, 65
- przetwarzanie potokowe, 95
- przewidywanie
 - rozgałęzień, 95
- przypisywanie
 - wartości zerowej, 29
 - za pomocą rzutowania, 36
- przyporządkowanie
 - pamięci, 23
- pula literałów
 - łańcuchowych, 139
- pule struktur, 174

Q

- Queue, 185

R

- RAII, 76
- ramka, 26
- ramka stosu, 10, 31, 79, 80
 - nadpisywanie, 152
 - odkładanie parametrów, 82
 - organizacja, 81
- realokacja pamięci, 63
 - dodatkowej, 65
- rekordy aktywacji, 80
- reprezentacja
 - liczb
 - zmiennoprzecinkowych, 224

- Resource Acquisition Is Initialization, 76
- rodzaje wskaźników, 32
- rozmiar
 - bloku pamięci, 61
 - obiektu, 33
 - wskaźnika, 32
- rzędy, 106
- rzutowanie
 - adresu typu integer, 221
 - jawne, 59
 - wskaźnika, 216
 - na funkcję, 102
 - na void, 31
 - zmiennej typu integer, 216

S

- schemat pamięci, 21
- sekwencja porządkująca, 108
- słowo kluczowe
 - const, 44, 46
 - restrict, 221, 226
 - stosowanie, 227
- struct, 168
- typedef, 168
- volatile, 219
- sortowanie bąbelkowe, 163
- specyfikator
 - %d, 35
 - %lu, 34
 - %o, 25
 - %p, 25
 - %u, 34
 - %x, 25
 - %zu, 34, 35
- specyfikatory pola, 25
- sprzątanie pamięci, 76
- stała
 - będąca wskaźnikiem na stałą, 49
 - limit, 87
 - wskaźnika pustego, 28
 - znakowa, 138
- standard C99
 - obsługa tablic o zmiennej długości, 114
- status systemu, 209

- sterta, 9, 31, 70, 80
 - do dołu, 81
 - fragmentacja, 81
 - przechowywanie
 - niepotrzebnych obiektów, 57
 - rozmiar, 70
- stos, 9, 79, 80, 176
 - definiowanie jako listy
 - powiązanej, 188
 - do góry, 81
 - lista powiązana, 188
 - odkładanie danych, 188
 - przepelnienie, 83
 - struktura danych, 188
 - ściąganie danych, 188
 - zarządzanie, 81
 - zwracanie adresu
 - łańcucha lokalnego, 162
- strict aliasing, 222, 225
 - stosowanie ograniczeń, 225
 - wyłączanie, 222
- strona aplikacji, 26
- struktura, 167
 - _linkedList, 234
 - _person, 168, 171
 - alokowanie w pamięci, 169
 - AlternatePerson, 170
 - danych
 - implementacja, 176
 - Product, 228
 - rodzaje, 176
 - VectorInfo, 228
 - wskaźniki, 176
 - dealokacja, 170
 - deklarowanie, 168
 - derywowane, 239
 - dostęp do pól, 168
 - employee, 176
 - FactorialData, 231
 - item, 208
 - LinkedList, 179
 - Node, 179
 - Person, 170, 236
 - pochodne, 242

- Rectangle, 237
- Shape, 237
 - tworzenie
 - egzemplarzy, 239
 - unikanie narzutu, 174
- vFunction, 238
- symbol
 - \0, 138
 - 0, 28
 - gwiazdki, 21, 30
 - NULL, 28
- system
 - operacyjny
 - obsługa pamięci, 70
 - zabezpieczenia, 196
 - typów, 223
 - wykonawczy, 51
 - zaalokowanie pamięci dla struktury, 170
 - zarządzanie stosem programu, 82

T

- tablica, 105, 106
 - a lista powiązana, 177
 - a wskaźnik
 - różnice, 112
 - alokacja nieciągła, 128
 - alokowana na sterce, 113
 - alokowanie pamięci, 62, 90
 - AlternatePerson, 170
 - bestBooks, 43
 - deklaracja, 106
 - dwuwymiarowa, 106, 108
 - a tablica
 - wskaźników, 106
 - alokacja o ciągłym obszarze, 129
 - alokacja o nieciągłym obszarze, 129
 - deklaracja, 108
 - dostęp do tablicy, 108
 - dynamiczna alokacja pamięci, 128
 - graficzna
 - interpretacja, 124

- identyfikacja
 - elementu, 108
 - mapowanie, 108
- elementów typu char, 138
- inicjalizacja, 141
- header, 139
- i struktury, 167
- indeksy i wskaźniki, 110
- inicjowanie blokiem wartości, 100
- jednowymiarowa, 107
 - nazwa, 107
 - określanie liczby elementów, 107
 - przekazywanie, 118
 - wewnętrzna
 - reprezentacja, 107
- kolejność elementów, 42
- liczb całkowitych
 - alokacja pamięci, 88
- matrix, 123
 - przekazywanie, 125
- metod wirtualnych, 238
- name, 148
- nazwa, 105
 - użycie, 110
- o zmiennym rozmiarze, 19, 66, 106
 - obsługa, 63
 - ograniczenie, 114
 - rozmiar, 66
- operowanie, 131
- postrzępiona, 131
 - deklaracja, 133
 - dostęp do elementów, 133, 134
 - inicjowanie
 - elementów, 133
 - sprawdzenie
 - poprawności, 133
 - tworzenie, 122
- przekazywanie
 - rozmiaru, 106
- pusta, 100
- rozmiar, 107
 - błędne obliczenie, 203
- sposób alokacji, 128
- statyczna, 90

- titles, 43
- tworzenie, 105
- vector
 - indeksy, 107
- wewnątrz funkcji
 - deklarowanie, 118
- wielowymiarowa, 109
 - przekazywanie, 125
 - wskaźniki, 122
- wirtualna, 238
 - Vtable, 201
- wskaźników
 - alokacja pamięci, 121
 - jednowymiarowa, 120
 - na funkcję, 100, 201
- wyrażień
 - wskaźnikowych, 122
- zachowanie granic, 203
- znaków
 - a łańcuchy, 138
- tail, 177
- TCMalloc, 75
- techniki obiektowe, 233
- treści
 - abstrakcyjne, 13
 - komend, 13
- typ
 - danych, 223
 - i rozmiary, 38
 - unsigned char, 176
 - w języku C, 33
 - wchar_t, 138
 - definicji
 - fpPtrOperator, 101
 - float, 223
 - int, 138
 - intptr_t, 33, 36
 - ptrdiff_t, 33, 42
 - size_t, 31, 33, 58
 - uintptr_t, 33, 36
 - unsigned integer, 223
 - wskaźników, 49
- type punning, 223
- typedef, 100

U

- ubijanie typu, 223
- uchwyt, 217
- ułożenie wskaźników
 - globalnych i statycznych, 31
- unia
 - konwersja typu zmiennej, 223
 - reprezentacja wartości, 223
- unieważnienie wskaźnika, 68
- usuwanie nieużytków, 76
- utożsamianie, 148
 - nazw, 69
- utrata adresu, 56
 - pamięci alokowanej dynamicznie, 57
- uwolnienie pamięci
 - nadpisanie danych, 74

V

- Virtual Table, 201
- VLA, 66
- volatile, 220

W

- wartości
 - bez znaku, 34
 - binarne, 55
 - ENOMEM, 63
 - lewostronne, 27, 112
 - null, 27
 - NULL, 30
 - przypisywanie do wskaźnika, 68
 - wiszące wskaźniki, 74
 - zwalniany wskaźnik, 54
 - typu unsigned, 34
 - ujemne, 34
 - wskaźników, 24

- zerowa
 - wykrywanie, 200
 - zwracane przez funkcję problemy, 210
- wątek
 - alokowanie stosu programu, 83
 - i wskaźniki, 227
 - suma wektorów, 229
 - tworzenie, 230, 232
 - współdzielenie danych, 227
- wczytywanie ciągu łańcuchów, 147
- wektor, 106
 - sumowanie, 228
- wersja testowa
 - wycieki pamięci, 74
- węzeł, 19
 - dziecko, 190
 - rodzic, 190
 - terminalny, 190
- wielopoziomowe
 - adresowanie pośrednie, 43, 49
- wiersz, 109
- wirtualny system operacyjny, 26
- wskaźniki, 15
 - a zmienne typu integer, 24
 - aliasing, 221
 - unikanie problemów, 222
 - argumenty wyrażenia logicznego, 29
 - bazowe, 81, 103
 - będące stałą
 - na obiekt niebędący stałą, 49
 - bliskie, 37
 - buffer, 116
 - dalekie, 37
 - dane wymagające modyfikacji, 84
 - deklarowanie, 20, 22
 - niewłaściwe, 197
 - dereferencji, 21

- do danych lokalnych, 89
- do obsługi struktur danych, 176
- do realokowanej pamięci, 63
- do struktury, 32
- do wskaźnika, 44
- do int, 92
 - zastosowanie, 94
- do znaku, 32
- dodawanie liczby całkowitej, 38
- dzikie, 198
- errno, 64
- foo, 27, 96
- fpBase, 102
- globalne, 31
- i łańcuchy, 137
- i pamięć, 16
- i stałe, 44, 87
- i struktury, 167
- i tablice, 105
 - różnice, 112
 - wielowymiarowe, 122
 - zastosowanie, 109
- implementacja struktur danych, 18
- inicjalizacja
 - niepowodzenia, 198, 213
 - zewnętrzne narzędzia, 199
 - znak NULL, 198
- int, 30
- kopiowanie, 149
- liczb, 224
- logika działania, 18
- na char, 39, 43
- na dane, 31, 32
 - operacje arytmetyczne, 37
- na elementy tablicy porównywanie, 42
- na funkcje, 27, 32, 44, 79, 95
 - bazowe, 102
 - COMPARE, 177, 190
 - deklarowanie, 96

- DISPLAY, 177
- działanie programu, 95
- i łańcuchy, 162
- lista argumentów, 103
- obsługa wywołań
 - zwrrotnych, 231
- porównywanie, 101
- prefiks fptr, 96
- przekazywanie, 99
- rozmiary, 35
- rzutowanie, 102
- tablice, 100
- zastosowanie, 97, 182
- zwracanie, 99
- na niezainicjowaną
 - pamięć, 21
- na notację wskaźnikową, 122
- na obiekt niebędący
 - stałą, 49
- na obiekt typu char, 17, 137, 223
- inicjalizowanie, 142
- rozmiar, 35
- zapis, 139
- na obiekt typu integer, 24
- rzutowanie, 36
- na short, 39
- na siebie, 39
- na stałą, 44, 49
 - będącą wskaźnikiem
 - na stałą, 49
 - typu char, 155
 - typu integer, 45
- na tablice
 - deklaracja, 123
- na void, 25, 30
 - dodawanie, 40
- niekompatybilne, 213
- nieprzeźroczyste, 215, 233
 - odbudowywanie
 - danych, 233
 - stosowanie, 233
 - tworzenie, 233
- niewłaściwe
 - przyporządkowanie, 208
- niezainicjowane, 29
- rozwiązywanie
 - problemów, 198
- odejmowanie, 41
 - liczby całkowitej, 40
- ograniczone, 205
- określenie typu, 17, 22
- operatory, 37
 - sizeof, 35
- pamięć wirtualna, 26
- pc, 40
- pi, 22
- podwójne, 43
- poprawność użycia, 96
- porównywanie, 42, 146
- predefiniowane typy
 - danych, 33
- problemy, 19
 - deklaracja
 - i inicjalizacja, 197
 - dopasowanie typów, 204
 - sprawdzanie granic, 203
- przechowywanie, 34
 - adresów, 36
- przekazywanie, 84
 - danych, 84
 - do stałej, 86
- przypisanie
 - wartości null, 28
 - wartości zerowej, 29
- ps, 40
- puste, 28
 - dereferencja, 29
 - zastosowanie, 28
- pv
 - tablice, 110
- ramki, 82
- rodzaje, 32
- root, 191
- rozmiary, 32, 35
- rysowanie schematów, 23
- rzutowane, 216
- składnia i semantyka, 20
- sprytne, 206
- statyczne, 31
- stosu, 81, 82
 - bazowe, 82
- tożsame, 73, 221
 - ostrzeżenia
 - kompilatorów, 223
- typu constant, 46
 - na obiekt niebędący
 - stałą, 46
 - stosowane do stałych, 48
- uruchomienie
 - programu, 31
- uzyskanie dostępu
 - do portu, 219
- w funkcjach, 79
- w implementacji operacji
 - dodawania skalarów, 112
- w języku C, 9
- w pamięci podręcznej, 226
- wiszące, 67, 71, 72
 - problemy, 71, 201
- wskazujące
 - to samo miejsce, 149, 222
 - ten sam obiekt, 225
- współdzielenie przez
 - wątki, 228
- wyłuskiwanie, 26
- wymagające inicjalizacji, 155
- wyświetlanie wartości, 24
- zachowanie, 20
- zapis, 19
 - znak spacji, 21
- zarządzanie pamięcią, 15
- zastosowanie, 17, 42
 - do obsługi drzewa, 190
 - do obsługi stosu, 188
 - niewłaściwe, 195, 212
 - obsługa kolejek, 185
 - zwracanie, 84, 87
- współdzielenie danych, 228
- wycieki pamięci, 55, 93
 - łańcuch alokowany
 - dynamicznie, 161

- wycieki pamięci
 - stosowanie wersji testowej, 74
 - struktury, 172
 - ukryte, 57
 - uwalnianie węzłów, 235
- wyczyszczenie pamięci, 62
- wyluskiwanie, 26
 - cp, 47
 - uwolnionego wskaźnika, 68
 - wartości, 84
 - wskaźnika, 26
 - na stałą, 45
 - portu, 219
 - do wskaźnika do liczby całkowitej, 93
- wyrażenie
 - &vector, 110
 - matrix[0], 124
 - warunkowe
 - unikanie stosowania, 95
 - wskaźnikowe, 122
 - wyświetlanie wartości wskaźników, 74
 - wywołanie zwrotne, 215, 231
 - wyzerowanie wskaźnika, 33

Z

- zabezpieczenia
 - aplikacji, 195
 - danych, 227
 - deklaracja i inicjalizacja wskaźników, 197
 - łańcuchy, 206
 - narzędzia analizy statycznej, 212
 - przepelnienie bufora, 199
 - stosowanie wskaźników, 196
 - wiszący wskaźnik, 201
- zachowanie
 - polimorficzne, 233

- wskaźnika, 20
 - nieokreślone, 20
 - niezdefiniowane, 20
 - określone miejscowo, 20
 - zdefiniowane przez implementację, 20
- zapis
 - poza końcem łańcucha, 206
 - poza obszarem tablicy, 203
 - w porządku little-endian, 220
 - wskaźników, 19
- zapobieganie wykonywania danych, 196
- zarządzanie pamięcią, 51, 67
- zastosowania wskaźników, 42
- zawartość pamięci, 196
- zdanie alokujące pamięć, 62
- zero, 30
 - binarne, 29
- zmienna
 - age, 176
 - alokowanie, 16, 51
 - arr
 - jako statyczna, 90
 - automatyczna, 80
 - będąca wskaźnikiem, 17
 - buffer, 115
 - character, 115
 - chunk, 56
 - command, 146
 - count, 148
 - currentPosition, 115
 - deklarowanie, 20
 - wewnątrz funkcji, 16
 - dyrektywy, 197
 - errno, 63
 - getLine, 115
 - globalna, 16
 - head, 18
 - lokalna
 - ramka stosu, 80, 82
 - length, 115
 - maxLength, 115
 - name, 77
 - newBuffer, 116
 - nieulotna, 220
 - pv, 110
 - rzutowanie na wskaźnik, 24
 - sizeIncrement, 114, 115
 - statyczna, 16
 - a funkcja malloc, 62
 - tmp, 73
 - typu
 - integer, 169
 - short, 169
 - size_t, 34
 - vector
 - zwrócenie adresu, 110
 - zasięg i okres istnienia, 17
 - znak
 - NUL, 138
 - określenie końca łańcucha, 118
 - NULL, 59
 - operacji, 99
 - zwolnienie pamięci, 67
 - bloku pamięci, 69
 - po zakończeniu programu, 70
 - zalety, 70
 - zwracanie adresu
 - literału, 158
 - łańcucha lokalnego, 161
 - pamięci adresowanej dynamicznie, 160
 - łańcuchów, 158
 - wskaźnika, 87
 - do wskaźnika, 92
 - na funkcję, 99
 - problemy, 88
 - przez funkcję, 84

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA



Helion SA

Wskaźniki w języku C. Przewodnik



Jeśli chcesz błyskawicznie opanować programowanie w języku C, sięgnij po tę książkę! Gdy już poznasz podstawy, nauczysz się także korzystać ze wskaźników. To prawdziwa zmora wszystkich programistów, błędne wykorzystanie wskaźnika może bowiem w okamgnieniu zrujnować Twój program. Zobacz, jak tego uniknąć i zaprzyjaźnić się ze wskaźnikami.

Inne książki opisują wskaźniki w jednym lub dwu rozdziałach, natomiast my poświęciliśmy im całą książkę. Dzięki temu dogłębnie poznasz ten mechanizm, zrozumiesz go i przekonasz się, że przy odrobinie uwagi nie jest wcale taki straszny! W trakcie lektury wykorzystasz wskaźniki na funkcję, przygotujesz tablicę wskaźników oraz zobaczysz, jak współdziałają one z łańcuchami znaków. Twoją uwagę z pewnością zwrócą fragmenty omawiające zabezpieczenia oraz niewłaściwe wykorzystanie wskaźników. Książka ta jest jedyną pozycją na rynku w całości poświęconą wskaźnikom w języku C. To lektura obowiązkowa każdego programisty!

Poznaj:

- koncepcję wskaźników
- zastosowanie tablic wskaźników
- funkcje dynamicznego alokowania pamięci
- zagrożenia wynikające ze stosowania wskaźników

Odkryj tajniki wskaźników w języku C i wykorzystaj ich potencjał!

helion.pl
księgarnia
internetowa

Nr katalogowy: 16986



Księgarnia internetowa:
<http://helion.pl>



Zamówienia telefoniczne:
0 801 339900



0 601 339900



Helion

Sprawdź najnowsze promocje:

🔗 <http://helion.pl/promocje>

Książki najchętniej czytane:

🔗 <http://helion.pl/bestsellery>

Zamów informacje o nowościach:

🔗 <http://helion.pl/novosci>

Helion SA

ul. Kościuszki 1c, 44-100 Gliwice

tel.: 32 230 98 63

e-mail: helion@helion.pl

<http://helion.pl>

sięgnij po **WIECEJ**



KOD KORZYŚCI

ISBN 978-83-246-8289-8



Cena 44,90 zł