Perfekcyjny podręcznik do nauki C!

Rusz głową!

C

Odkryj sekrety mistrzów C

Dowiedz się, jak make może zmienić Twoje życie

Uniknij żenujących wpadek ze wskaźnikami

Dowiedz się, jak funkcje o zmiennej liczbie elementów zapewniły Zuzannie większą elastyczność

Zabaw się z biblioteką standardową C

Napisz staromodną grę zręcznościową

David Griffiths, Dawn Griffiths

O'REILLY®

Helion

Tytuł oryginału: Head First C

Tłumaczenie: Piotr Rajca

ISBN: 978-83-246-5232-7

© 2013 Helion S.A.

Authorized Polish translation of the English edition of *Head First C*, 1st edition, ISBN *9781449399917*
© David Griffiths and Dawn Griffiths.

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

Wydawnictwo HELION dołożyło wszelkich starań, by zawarte w tej książce informacje były kompletne
i rzetelne. Nie bierze jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym
ewentualne naruszenie praw patentowych lub autorskich. Wydawnictwo HELION nie ponosi 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)

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

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

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści (skrócony)

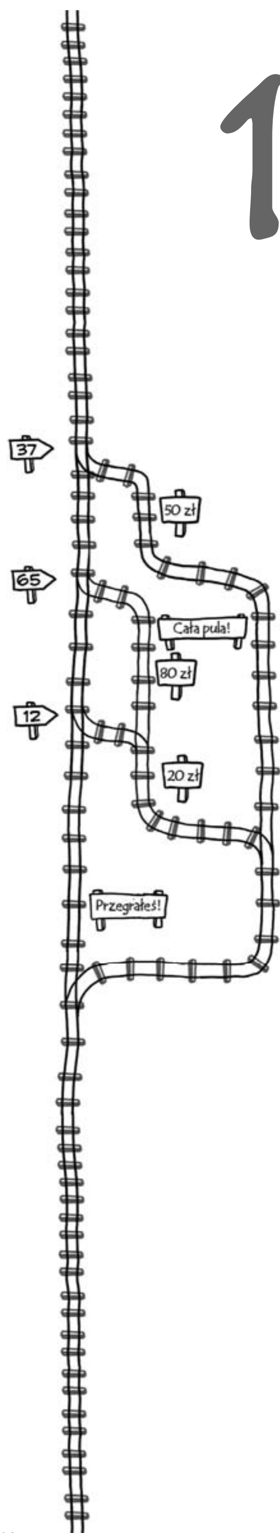
	Wprowadzenie	xxvii
1	Zaczynamy poznawać C. *Dajmy nurka*	1
2	Pamięć i wskaźniki. *Na co wskazujesz?*	41
2,5	Łańcuchy znaków. *Teoria łańcuchów*	83
3	Tworzenie małych programów narzędziowych. *Rób jedną rzecz, ale rób ją dobrze*	101
4	Stosowanie wielu plików źródłowych. *Podziel go, rozbuduj go*	155
	1. laboratorium C. *Arduino*	205
5	Struktury, unie i pola bitowe. *Wytocz swoje własne struktury*	215
6	Struktury danych i pamięć dynamiczna. *Budowanie mostów*	265
7	Zaawansowane funkcje. *Odpicuj swoje funkcje na maksa!*	309
8	Biblioteki statyczne i dynamiczne. *Wymienialny kod*	349
	2. laboratorium C. *OpenCV*	387
9	Procesy i wywołania systemowe. *Przekraczanie granic*	395
10	Komunikacja pomiędzy procesami. *Dobrze jest porozmawiać*	427
11	Gniazda i komunikacja sieciowa. *Nie ma drugiego takiego miejsca jak 127.0.0.1*	465
12	Wątki. *To równoległy świat*	499
	3. laboratorium C. *Blasteroidy*	521
A	Pozostałości. *Dziesięć najważniejszych rzeczy (których nie opisaliśmy)*	537
B	Zagadnienia programowania w C. *Powtórka z całego materiału*	551

Spis treści (z prawdziwego zdarzenia)

Wprowadzenie

Twój mózg jest skoncentrowany na C. Kiedy Ty starasz się czegoś *nauczyć*, Twój *mózg* robi Ci przysługę i stara się, by ta wiedza nie została *utrwalona*. Twój mózg myśli sobie: „Lepiej zostawić miejsce na naprawdę ważne rzeczy, takie jak dzikie zwierzęta, których należy unikać, albo rozważania, czy jeżdżenie na snowboardzie w stroju Adama to dobry pomysł". Jak zatem możesz *oszukać* swój mózg i przekonać go, że Twoje życie zależy od znajomości C?

Dla kogo jest przeznaczona ta książka?	xxviii
Wiemy, co sobie myślisz	xxix
Metapoznanie	xxxi
Zmuś swój mózg do posłuszeństwa	xxxiii
Przeczytaj to	xxxiv
Zespół recenzentów technicznych	xxxvi
Podziękowania	xxxvii



1

Zaczynamy poznawać C

Dajmy nurka

Czy chcesz zajrzeć do głowy komputera?

Musisz napisać **kod działający naprawdę szybko**, na przykład na potrzeby nowej gry? A może program na **Arduino**? Albo we własnej aplikacji na iPhone'a użyć **biblioteki napisanej przez kogoś** innego? Jeśli tak, to skorzystaj z pomocy bohaterskiego C. C działa na **znacznie niższym poziomie** niż większość innych języków programowania, a zatem zrozumienie go daje nam znacznie większe pojęcie o tym, **co się naprawdę dzieje** w programie. C pozwala także lepiej zrozumieć inne języki programowania. A zatem bierz się do pracy, przygotuj kompilator, a już niedługo zaczniesz poznawać C.

C to język do pisania małych, szybkich programów 2
Ale jak wygląda skompilowany program napisany w C? 5
A jak można uruchomić program? 9
Dwa rodzaje poleceń 14
Oto kod, jakim aktualnie dysponujemy 15
Liczenie kart? W języku C? 17
Wartości logiczne to nie tylko sprawdzanie równości 18
Jak aktualnie wygląda nasz kod? 25
Pociąg do Switcherado 26
Czasami jeden raz nie wystarcza… 29
Pętle często mają taką samą strukturę… 30
Instrukcji break używamy, by wydostać się z pętli… 31
Twój niezbędnik C 40

2

Pamięć i wskaźniki

Na co wskazujesz?

Jeśli naprawdę chcesz zrobić coś odlotowego w języku C, musisz się dowiedzieć, w jaki sposób zarządza on pamięcią.

Język C daje nam bardzo dużą *kontrolę* nad tym, jak programy korzystają z **pamięci komputera**. W tym rozdziale zajrzysz za kulisy i zobaczysz, co dokładnie dzieje się podczas **zapisywania i odczytywania zmiennych**. Dowiesz się, **jak działają tablice**, jak unikać **paskudnych błędów w trakcie wykonywania operacji na pamięci**, a przede wszystkim przekonasz się, że opanowanie **operacji na wskaźnikach i adresowania** pozwoli Ci stać się rewelacyjnym programistą C.

Kod C zawiera wskaźniki 42
Grzebiemy w pamięci 43
Stawiamy żagle ze wskaźnikami 44
Spróbujmy przekazać wskaźnik do zmiennej 47
Stosowanie wskaźników 48
Jak przekazać łańcuch znaków do funkcji? 53
Zmienne tablicowe są jak wskaźniki… 54
Co myśli komputer, wykonując nasz kod? 55
Jednak zmienne tablicowe nie są tak do końca wskaźnikami 59
Dlaczego tablice naprawdę zaczynają się od 0? 61
Dlaczego wskaźniki mają typ? 62
Stosowanie wskaźników do wprowadzania danych 65
Używając funkcji scanf(), uważaj! 66
Alternatywą dla scanf() jest fgets() 67
Literały łańcuchowe nie mogą być nigdy modyfikowane 72
Jeśli chcesz zmienić łańcuch — skopiuj go 74
Ściąga z pamięci 80
Twój niezbędnik C 81

2,5

Łańcuchy znaków

Teoria łańcuchów

Korzystanie z łańcuchów nie ogranicza się do ich odczytywania.
Przekonałeś się już, że w języku C łańcuchy znaków są w rzeczywistości *tablicami* danych typu `char`. Pozostaje jednak pytanie, co język C pozwala nam z nimi robić. W tym właśnie momencie na scenę wkracza ***string.h***. To część standardowej biblioteki języka C, przeznaczona do wykonywania **operacji na łańcuchach znaków**. Jeśli chcemy ***połączyć ze sobą*** dwa łańcuchy, ***skopiować*** jeden łańcuch do drugiego bądź też ***porównać*** dwa łańcuchy znaków, to wszystkie te operacje można wykonać przy użyciu funkcji zadeklarowanych w pliku nagłówkowym *string.h.* W tym rozdziale dowiesz się, jak można stworzyć **tablicę łańcuchów**, a następnie przyjrzysz się nieco bliżej ***przeszukiwaniu zawartości łańcuchów*** przy użyciu funkcji **`strstr()`**.

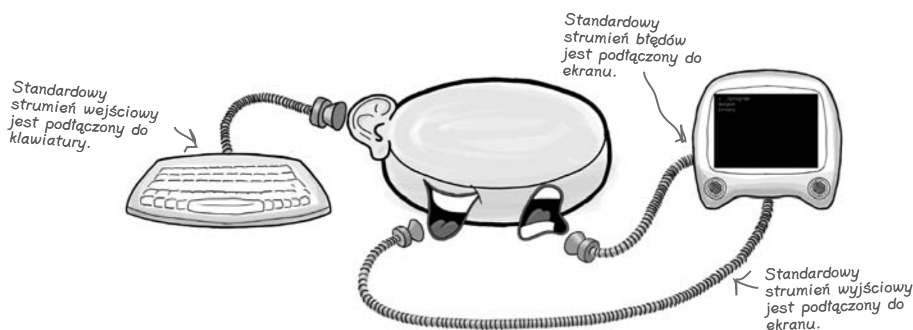
Desperacko poszukuję Franka 84
Utwórz tablicę tablic 85
Odnajdywanie łańcucha zawierającego określony tekst 86
Stosowanie funkcji strstr() 89
Czas na przegląd kodu 94
Tablica tablic czy tablica wskaźników? 98
Twój niezbędnik C 99

3 Tworzenie małych programów narzędziowych

Rób jedną rzecz, ale rób ją dobrze

Każdy system operacyjny udostępnia niewielkie programy narzędziowe. Niewielkie programy narzędziowe pisane w języku C wykonują wyspecjalizowane zadania, takie jak odczytywanie i zapisywanie plików czy też filtrowanie danych. Jeśli chcemy wykonać bardziej złożone zadanie, można nawet *połączyć ze sobą kilka takich programów*. W jaki jednak sposób tworzy się takie małe programy narzędziowe? W tym rozdziale przyjrzymy się elementom używanym podczas ich tworzenia. Dowiesz się, jak korzystać z **opcji wiersza poleceń**, jak zarządzać **strumieniami informacji**, czym są **przekierowania**, i błyskawicznie zdobędziesz nowe narzędzia.

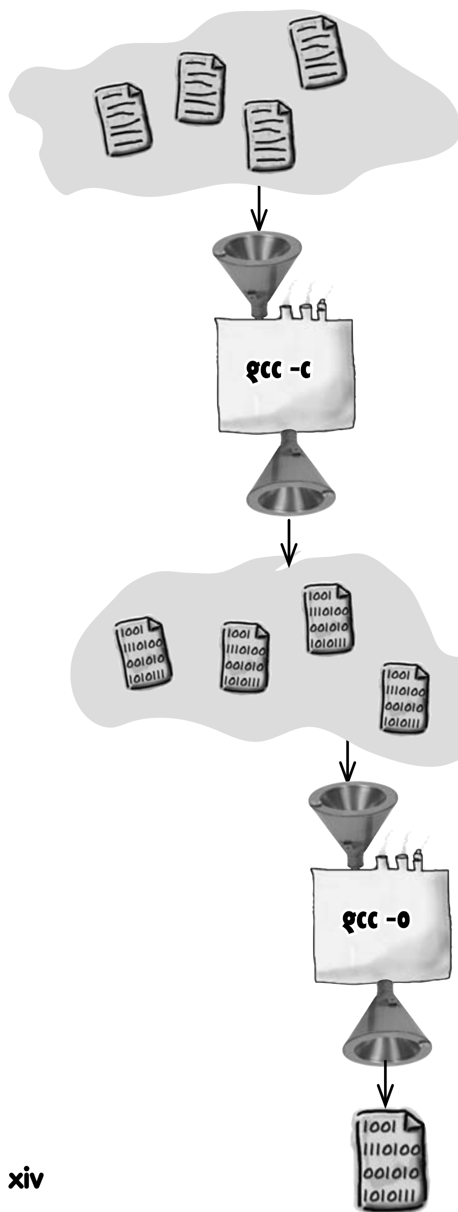
Małe programy narzędziowe mogą rozwiązywać wielkie problemy	102
Oto sposób wykorzystania programu	106
Ale my nie używamy plików…	107
Możesz skorzystać z przekierowania	108
Przedstawiamy standardowy strumień błędów	118
Domyślnie strumień błędów jest wyświetlany na ekranie	119
fprintf() zapisuje dane w strumieniu	120
Zaktualizujmy kod, by korzystał z funkcji fprintf()	121
Niewielkie programy narzędziowe są elastyczne	126
Nie zmieniaj programu geo2json	127
Różne zadania wymagają różnych narzędzi	128
Połącz wejście i wyjście przy użyciu potoku	129
Program narzędziowy bermuda	130
A jeśli chcemy przekazywać wyniki do więcej niż jednego pliku?	135
Stwórz swoje własne strumienie danych	136
Nieco więcej o funkcji main()	139
Niech biblioteka wykona pracę za nas	147
Twój niezbędnik C	154



Stosowanie wielu plików źródłowych

4 Podziel go, rozbuduj go

Jeśli piszesz duże programy, nie chcesz mieć równie dużych plików źródłowych. Czy jesteś sobie w stanie wyobrazić, jak trudna i czasochłonna byłaby pielęgnacja dużego programu korporacyjnego napisanego w formie jednego pliku źródłowego? W tym rozdziale dowiesz się, jak C pozwala na dzielenie kodu źródłowego na **małe, poręczne fragmenty** oraz jak potem utworzyć z nich **jeden duży program**. W międzyczasie dowiesz się także nieco o **niuansach związanych z typami danych** i poznasz swojego nowego najlepszego przyjaciela — program make.



Krótki przewodnik po typach danych	160
Nie umieszczaj czegoś dużego w czymś małym	161
Użyj rzutowania, by zapisać wartość zmiennoprzecinkową w zmiennej całkowitej	162
O nie… to bezrobotni aktorzy…	166
Zobaczmy, co się stało z kodem	167
Kompilatory nie lubią niespodzianek	169
Oddziel deklaracje od definicji	171
Tworzenie pierwszego pliku nagłówkowego	172
Jeśli oprogramowałeś często używane operacje…	180
Możesz rozdzielić kod, umieszczając go w osobnych plikach	181
Za kulisami kompilacji	182
Współdzielony kod trzeba umieścić w osobnym pliku źródłowym	184
To nie jest kosmiczna technologia… a może jest?	187
Nie rekompiluj każdego pliku	188
Najpierw skompiluj źródła do plików obiektowych	189
Ciągłe śledzenie zmian w plikach jest trudne	194
Zautomatyzuj kompilacje, używając narzędzia make	196
Jak działa make	197
Przekaż informacje o kodzie, używając pliku makefile	198
Startujemy!	203
Twój niezbędnik C	204

1. laboratorium C

Arduino

Czy kiedykolwiek marzyłeś o tym, by rośliny mogły Ci powiedzieć, kiedy potrzebują podlania? No i proszę: dzięki Arduino stało się to możliwe! W tym laboratorium stworzysz działający w oparciu o Arduino monitor roślin, napisany w całości w języku C.

5

Struktury, unie i pola bitowe

Wytocz swoje własne struktury

Większość rzeczy w realnym życiu jest nieco bardziej złożona niż proste liczby. Do tej pory poznałeś jedynie podstawowe typy danych dostępne w języku C, ale co można zrobić, by wykroczyć poza to, co dają nam liczby i łańcuchy znaków — gdybyśmy chcieli **odwzorować przedmioty z realnego świata**? W języku C **złożoności realnego świata** można odwzorowywać przy użyciu struktur — typu `struct`. W tym rozdziale dowiesz się, jak **łączyć podstawowe typy danych** przy użyciu ***struktur***, a nawet jak radzić sobie z ***niewiadomymi życia*** przy użyciu ***unii***. Jeśli z kolei wolisz proste odpowiedzi typu „tak" lub „nie", to mogą Ci się przydać ***pola bitowe***.

Czasami musisz rozdawać wiele danych 216
Rozmowa trójstronna 217
Twórz swoje własne strukturalne typy danych przy użyciu struct 218
Po prostu daj im rybkę 219
Odczytuj pola struktur, używając operatora "." 220
Czy można umieścić jedną strukturę wewnątrz drugiej? 225
Jak zaktualizować zawartość struktury? 234
Ten kod klonuje żółwie 236
Potrzebny będzie wskaźnik na strukturę 237
(*t).age kontra *t.age 238
Czasami rzeczy podobnego typu wymagają danych różnych typów 244
Unie pozwalają używać bloku pamięci na różne sposoby 245
Jak stosować unie? 246
Zmienna typu enum przechowuje symbol 253
Czasami potrzebujemy kontroli na poziomie bitów 259
Pola bitowe zawierają dowolną liczbę bitów 260
Twój niezbędnik C 264

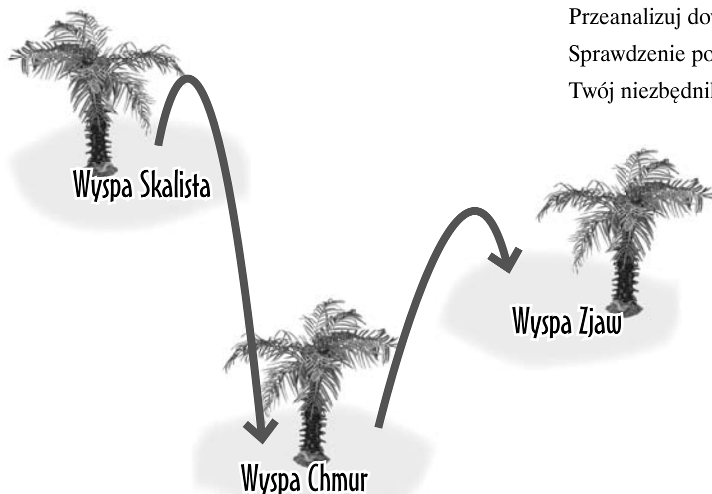
Struktury danych i pamięć dynamiczna

6 Budowanie mostów

Czasami prosta struktura nie wystarcza.

Aby zamodelować bardzo złożone dane, często będziemy musieli **łączyć ze sobą struktury**. W tym rozdziale zobaczysz, jak korzystać ze **wskaźników na dane typu `struct`**, by łączyć niestandardowe typy danych w **duże i złożone struktury danych**. Poznasz *kluczowe zasady*, ucząc się tworzyć **listy połączone**. Dowiesz się także, jak sprawić, by Twoje struktury danych radziły sobie ze zmienną liczbą informacji, wykorzystując do tego celu **dynamiczne przydzielanie pamięci na stercie** i zwalniając używaną pamięć, kiedy nie będzie już potrzebna. A gdy porządkowanie pamięci stanie się zbyt trudne, dowiesz się, jak może nam pomóc program `valgrind`.

Czy potrzebujesz elastycznego sposobu przechowywania danych?	266
Listy połączone przypominają łańcuchy danych	267
Listy połączone pozwalają na dodawanie elementów	268
Tworzenie struktur rekurencyjnych	269
Tworzenie wysp w języku C…	270
Wstawianie elementów pośrodku listy	271
Pamięć dynamiczną rezerwuj na stercie	276
Zwróć pamięć, kiedy nie będzie już potrzebna	277
Proś o pamięć, wywołując malloc()…	278
Popraw kod, używając funkcji strdup()	284
Zwalniaj pamięć, gdy jej nie potrzebujesz	288
Przegląd systemu SPIES	298
Detektywi oprogramowania: stosowanie valgrind	300
Skorzystaj z programu valgrind kilkakrotnie, by zebrać więcej dowodów	301
Przeanalizuj dowody	302
Sprawdzenie poprawek	305
Twój niezbędnik C	307



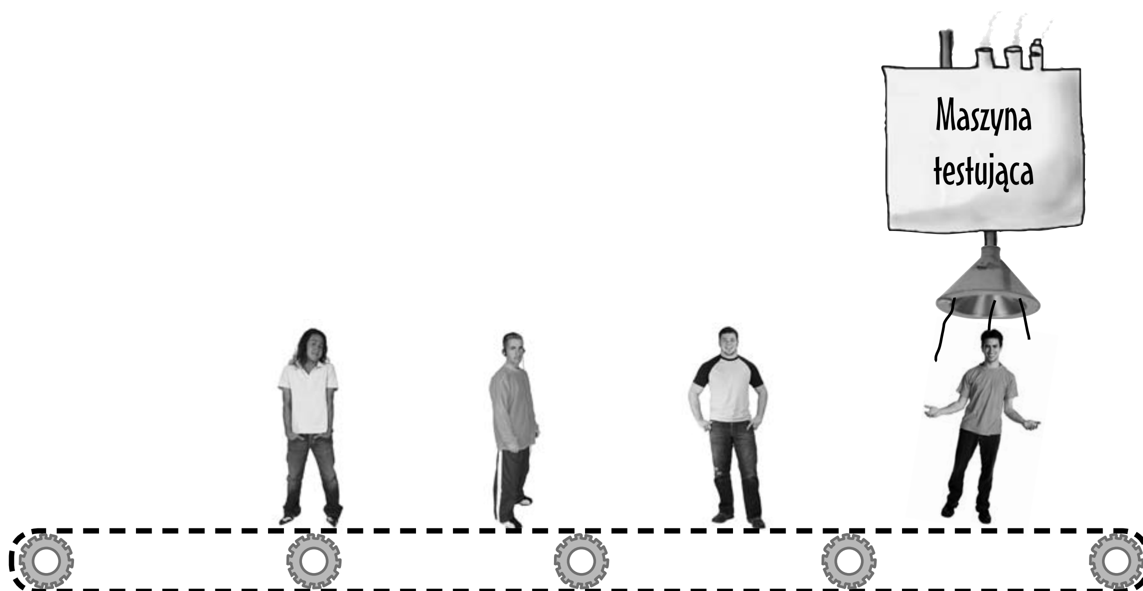
7

Zaawansowane funkcje

Odpicuj swoje funkcje na maksa!

Proste funkcje są w porządku, jednak czasami możemy potrzebować czegoś więcej. Do tej pory koncentrowaliśmy się na sprawach podstawowych, ale co zrobić, kiedy do osiągnięcia celu będziemy potrzebowali większych *możliwości* i większej *elastyczności*? W tym rozdziale zobaczysz, jak **podnieść IQ swojego kodu, przekazując funkcje jako parametry**. Dowiesz się, jak można **sortować, wykorzystując funkcje — komparatory**. A na samym końcu nauczysz się, jak dzięki zastosowaniu **zmiennej liczby argumentów** tworzyć *superelastyczne funkcje*.

Szukając Pana Doskonałego…	310
Przekaż kod do funkcji	314
Musisz przekazać funkcji find() nazwę funkcji testującej	315
Każda nazwa funkcji jest wskaźnikiem do tej funkcji…	316
…ale nie ma żadnego typu danych reprezentującego funkcję	317
Jak utworzyć wskaźnik do funkcji	318
Posortuj to, używając standardowej biblioteki C	323
Użyj wskaźnika do funkcji, by określić porządek sortowania	324
Automatyzacja generowania listów do Jana	332
Stwórz tablicę wskaźników do funkcji	336
Zapewnij swoim funkcjom elastyyyyyczność	341
Twój niezbędnik C	348



8

Biblioteki statyczne i dynamiczne

Wymienialny kod

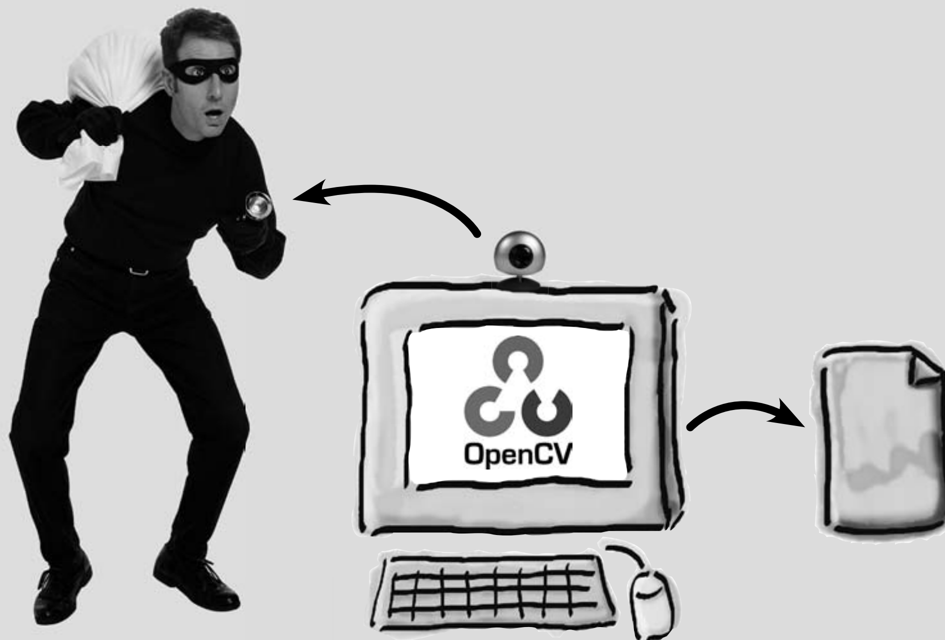
Poznałeś już ogromne możliwości bibliotek standardowych. Nadszedł czas, byś użył mocy swojego *własnego* kodu. W tym rozdziale dowiesz się, jak tworzyć swoje **własne biblioteki** oraz jak **wielokrotnie używać tego samego kodu w różnych programach**. Co więcej, nauczysz się współużytkowania kodu w trakcie działania programu, co jest możliwe dzięki **bibliotekom łączonym dynamicznie**. Poznasz sekrety *mistrzów kodowania*. A pod koniec tego rozdziału będziesz już potrafił pisać kod, który w łatwy i efektywny sposób będzie można skalować oraz którym będzie można równie łatwo zarządzać.

Kod, który możesz zabrać do banku	350
Nawiasy kątowe dołączają standardowe pliki nagłówkowe	352
A co zrobić, jeśli będziesz chciał współużytkować jakiś kod?	353
Współużytkowanie plików nagłówkowych	354
Współużytkowanie plików .o poprzez określanie pełnej ścieżki dostępu	355
Archiwum zawiera pliki .o	356
Utwórz archiwum, używając polecenia ar…	357
I w końcu kompiluj inne programy	358
Siłownia Rusz Głową wchodzi na scenę globalną	363
Obliczanie spalonych kalorii	364
Ale zagadnienie jest nieco bardziej skomplikowane…	367
Programy składają się z wielu fragmentów…	368
Łączenie dynamiczne następuje podczas działania programu	370
Czy pliki .a można łączyć podczas działania programu?	371
Najpierw utwórz plik obiektowy	372
Nazewnictwo bibliotek dynamicznych zależy od platformy systemowej	373
Twój niezbędnik C	385

2. laboratorium C

OpenCV

Wyobraź sobie, że komputer mógłby pilnować Twojego domu, kiedy Cię nie ma, i informować Cię, kto się po nim kręcił. W tym laboratorium napiszesz w C system wykrywania włamywaczy, korzystając przy tym z możliwości biblioteki OpenCV.



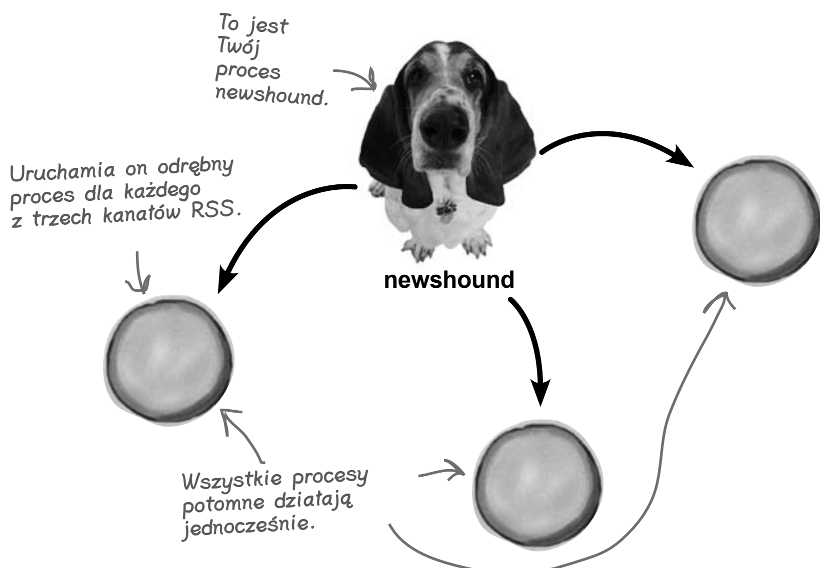
9

Procesy i wywołania systemowe

Przekraczanie granic

Czas, by zacząć myśleć kreatywnie. Dowiedziałeś się już, że można tworzyć złożone aplikacje, łącząc niewielkie programy narzędziowe wywoływane z poziomu wiersza poleceń. Ale co zrobić, gdy będziemy chcieli *korzystać z innych programów* we własnym kodzie? W tym rozdziale dowiesz się, jak korzystać z **usług systemowych**, by tworzyć i kontrolować działanie ***procesów***. Dzięki temu Twoje programy uzyskają dostęp do *poczty elektronicznej, WWW* oraz *wszelkich innych narzędzi zainstalowanych na komputerze*. Po przeczytaniu tego rozdziału zyskasz moc pozwalającą wykraczać **poza język C**.

Wywołania systemowe są Twoją gorącą linią z systemem operacyjnym	396
Wtem ktoś włamał się do systemu…	400
Bezpieczeństwo nie jest jedynym problemem	401
Funkcja exec() zapewnia większą kontrolę	402
Istnieje wiele funkcji exec()	403
Funkcje z tablicą argumentów: execv(), execvp() oraz execve()	404
Przekazywanie zmiennych środowiskowych	405
Większość wywołań systemowych zawodzi w taki sam sposób	406
Czytaj doniesienia, używając RSS	414
exec() jest końcem rodu naszego programu	418
Uruchamianie procesu potomnego przy użyciu funkcji fork() i exec()	419
Twój niezbędnik C	425



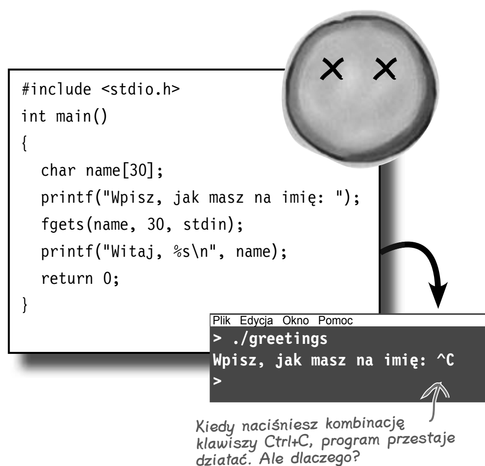
10 Komunikacja pomiędzy procesami

Dobrze jest porozmawiać

Tworzenie procesów to tylko połowa sukcesu.

Co zrobić, gdy chcemy *kontrolować* proces po jego uruchomieniu? Albo *przesłać do niego jakieś dane*? Albo *odczytać generowane przez niego wyniki?* **Komunikacja pomiędzy procesami** zapewnia procesom możliwość podejmowania wspólnych działań *w celu wykonania zadania.* W tym rozdziale pokażemy Ci, jak możesz zwiększyć **możliwości** swojego kodu, pozwalając mu na ***komunikowanie się*** z innymi programami w systemie.

Przekierowania strumieni wejściowych	428
Zajrzyjmy do wnętrza standardowego procesu	429
Przekierowanie zastępuje deskryptor	430
Funkcja fileno() zwraca deskryptor	431
Czasami trzeba poczekać…	436
Bądź w kontakcie ze swymi potomkami	440
Połącz swoje procesy potokami	441
Studium przypadku: otwieranie doniesień w przeglądarce	442
W procesie potomnym	443
W procesie rodzicielskim	443
Otwieranie strony w przeglądarce	444
Śmierć procesu	449
Przechwytywanie sygnałów i wykonywanie własnego kodu	450
Struktury sigaction są rejestrowane przy użyciu funkcji sigaction()	451
Modyfikacja kodu i wykorzystanie procedury obsługi sygnałów	452
Używaj polecenia kill, by wysyłać sygnały	455
Wysyłanie do procesu sygnału pobudki	456
Twój niezbędnik C	464



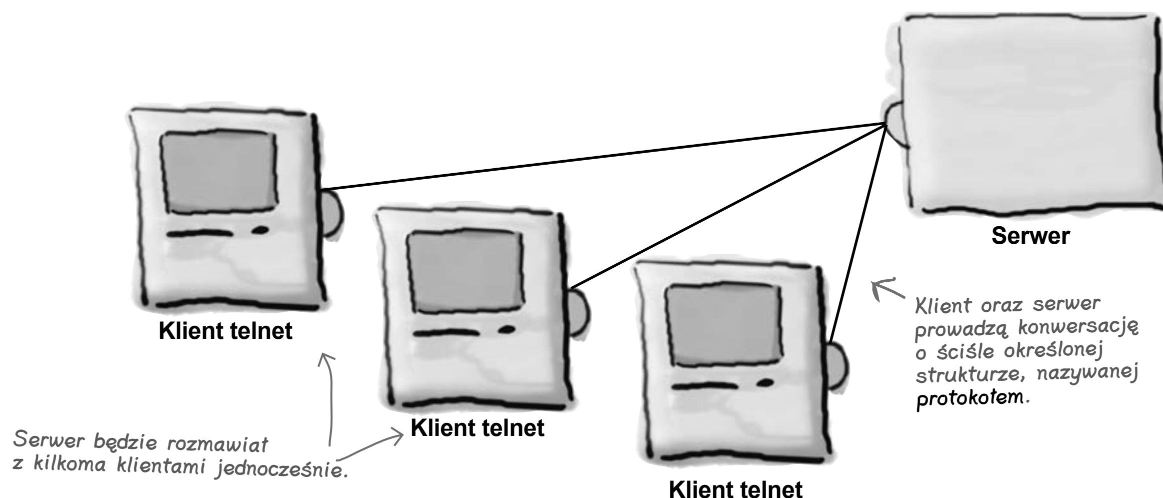
11

Gniazda i komunikacja sieciowa

Nie ma drugiego takiego miejsca jak 127.0.0.1

Programy działające na różnych komputerach muszą się ze sobą komunikować. Dowiedziałeś się już, jak można używać operacji wejścia-wyjścia, by korzystać z plików, oraz w jaki sposób mogą się ze sobą porozumiewać programy działające na tym samym komputerze. Teraz jednak *masz zamiar sięgnąć po resztę świata* i nauczyć się pisać w języku C programy, które będą mogły komunikować się z innymi programami działającymi **w tej samej sieci** oraz **na całym *świecie***. Po przeczytaniu tego rozdziału będziesz potrafił pisać **programy działające jako serwery** oraz **programy pracujące jako klienty**.

Internetowy serwer puk-puk 466
Prezentacja serwera puk-puk 467
PNAR — jak serwery komunikują się z internetem 468
Gniazdo nie jest typowym strumieniem danych 470
Czasami serwer nie uruchamia się prawidłowo 474
Dlaczego mama zawsze powtarzała Ci, byś sprawdzał błędy 475
Odczyt danych przesyłanych przez klienta 476
Serwer może rozmawiać tylko z jednym klientem naraz 483
Możesz użyć fork(), by obsłużyć oba klienty jednocześnie 484
Pisanie klienta WWW 488
To zadanie klienta 489
Utwórz gniazdo dla adresu IP 490
Funkcja getaddrinfo() pobiera adresy domen 491
Twój niezbędnik C 498



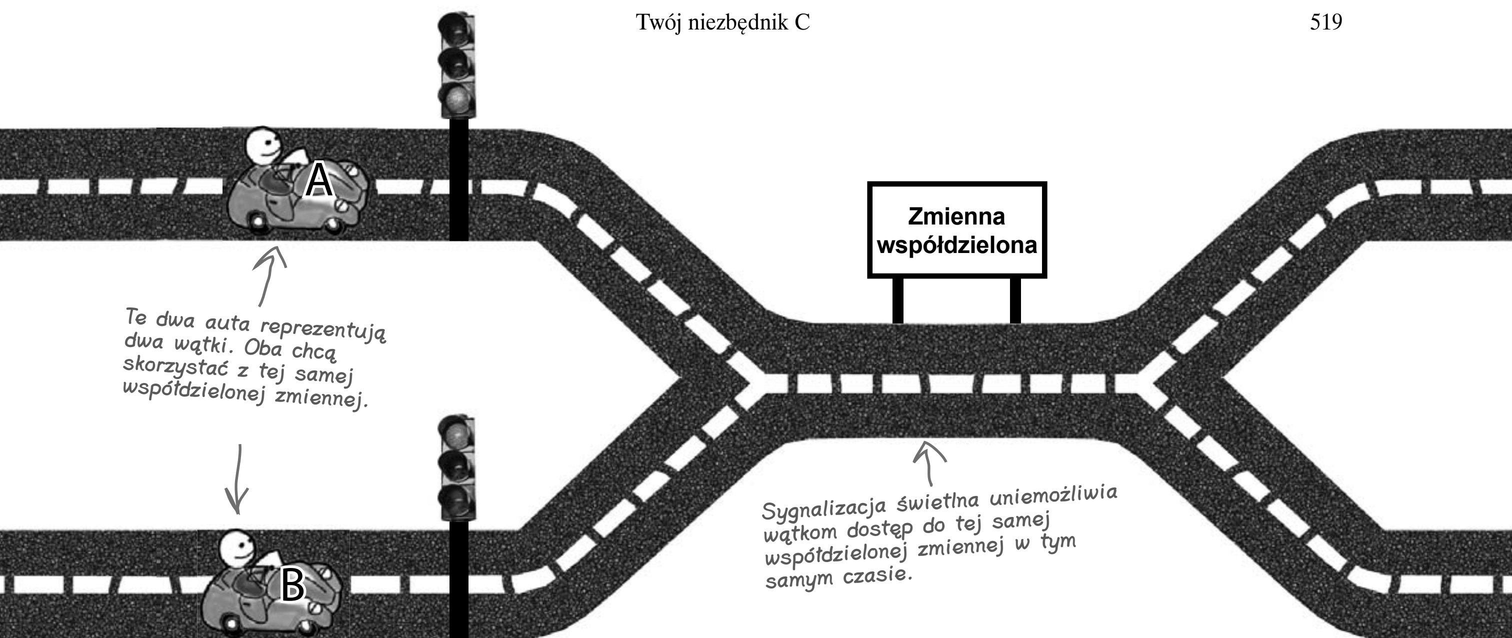
12 Wątki

To równoległy świat

Programy często muszą robić kilka rzeczy naraz.

Wątki POSIX mogą sprawić, że nasz kod będzie sprawniej reagował na poczynania użytkownika, a to dzięki **wydzieleniu kilku fragmentów, które będą działać jednocześnie**. Ale uważaj! Wątki są potężnym narzędziem, jednak na pewno byś nie chciał, żeby sobie wzajemnie przeszkadzały. W tym rozdziale dowiesz się, jak zainstalować **sygnalizację świetlną i wytyczyć linie**, które ***zapobiegną programistycznym karambolom w Twoim kodzie***. Po przeczytaniu tego rozdziału będziesz wiedział, jak **tworzyć wątki POSIX** oraz jak używać **mechanizmów synchronizacji**, by ***chronić integralność ważnych danych.***

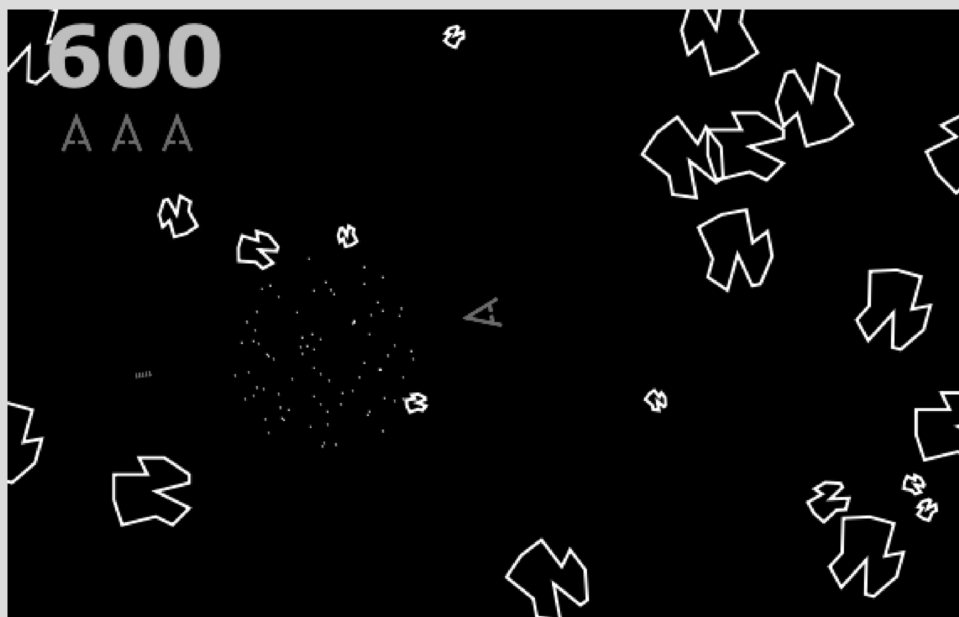
Zadania są sekwencyjne… lub nie… 500
…a procesy nie zawsze są właściwą odpowiedzią 501
Proste procesy robią po jednej rzeczy naraz 502
Zatrudnij dodatkowych pracowników: skorzystaj z wątków 503
Jak się tworzy wątki? 504
Utwórz wątki, używając funkcji pthread_create() 505
Ten kod nie jest wielobieżny 510
Potrzeba Ci sygnalizacji świetlnej 511
Użyj muteksu jako sygnalizacji świetlnej 512
Twój niezbędnik C 519



3. laboratorium C

Blasteroidy

W tym laboratorium oddasz hołd jednej z najbardziej popularnych i długowiecznych gier wideo wszech czasów. Pora, byś napisał własną wersję gry Blasteroids!



A

Pozostałości

Dziesięć najważniejszych rzeczy (których nie opisaliśmy)

Nawet po tym wszystkim, co już napisaliśmy, wciąż jeszcze pozostaje coś, czego nie wyjaśniliśmy.

Jest jeszcze parę zagadnień, o których według nas powinieneś się dowiedzieć. Nie czulibyśmy się dobrze, gdybyśmy zupełnie je zignorowali. Wymagają one choć krótkiego wyjaśnienia, a my naprawdę nie chcemy oddawać Ci do rąk książki, której nie byłbyś w stanie podnieść bez solidnego treningu na lokalnej siłowni. A zatem zanim odłożysz tę książkę na półkę, **przeczytaj zamieszczone tu informacje.**

1. Operatory 538
2. Dyrektywy preprocesora 540
3. Słowo kluczowe static 541
4. Jak duże to jest? 542
5. Automatyczne testowanie 543
6. Więcej o gcc 544
7. Więcej o programie make 546
8. Narzędzia programistyczne 548
9. Tworzenie graficznego interfejsu użytkownika 549
10. Materiały 550

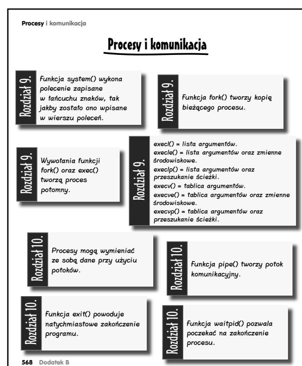
B

Zagadnienia programowania w C

Powtórka z całego materiału

Czy kiedykolwiek pomyślałeś, że fajnie by było, gdyby te wszystkie wspaniałe fakty dotyczące C zostały zebrane w jednym miejscu?

W tym dodatku zostały zebrane wszystkie zagadnienia oraz zasady związane z pisaniem w języku C, które przedstawiliśmy w całej książce. Przejrzyj je wszystkie i przekonaj się, czy je zapamiętałeś. Przy każdym fakcie został podany numer rozdziału, z którego on pochodzi, a zatem nie będziesz miał problemów z odnalezieniem dodatkowych informacji, gdybyś ich potrzebował. Możesz nawet wyciąć te strony z książki i przykleić je sobie na ścianie.


Procesy i komunikacja

Procesy i komunikacja

Rozdział 9. Funkcja system() wykona polecenie zapisane w łańcuchu znaków, tak jakby zostało ono wpisane w wierszu poleceń.

Rozdział 9. Funkcja fork() tworzy kopię bieżącego procesu.

Rozdział 9. Wywołania funkcji fork() oraz exec() tworzą proces potomny.

Rozdział 9. execl() = lista argumentów.
execle() = lista argumentów oraz zmienne środowiskowe.
execlp() = lista argumentów oraz przeszukanie ścieżki.
execv() = tablica argumentów.
execve() = tablica argumentów oraz zmienne środowiskowe.
execvp() = tablica argumentów oraz przeszukanie ścieżki.

Rozdział 10. Procesy mogą wymieniać ze sobą dane przy użyciu potoków.

Rozdział 10. Funkcja pipe() tworzy potok komunikacyjny.

Rozdział 10. Funkcja exit() powoduje natychmiastowe zakończenie programu.

Rozdział 10. Funkcja waitpid() pozwala poczekać na zakończenie procesu.

568 Dodatek B

1. Zaczynamy poznawać C

Dajmy nurka

Czy nie ubóstwiasz niebieskiego oCeanu? Wskakuj — woda jest cudowna!

Czy chcesz zajrzeć do głowy komputera?

Musisz napisać **kod działający naprawdę szybko**, na przykład na potrzeby nowej gry? A może program na **Arduino**? Albo we własnej aplikacji na iPhone'a użyć **biblioteki napisanej przez kogoś** innego? Jeśli tak, to skorzystaj z pomocy bohaterskiego C. C działa na **znacznie niższym poziomie** niż większość innych języków programowania, a zatem zrozumienie go daje nam znacznie większe pojęcie o tym, **co się naprawdę dzieje** w programie. C pozwala także lepiej zrozumieć inne języki programowania. A zatem bierz się do pracy, przygotuj kompilator, a już niedługo zaczniesz poznawać C.

C to język do pisania małych, szybkich programów

Język C został stworzony do pisania małych i szybkich programów. Działa na znacznie niższym poziomie niż większość innych języków programowania, a to oznacza, że *tworzy kod znacznie bliższy temu, co komputery naprawdę są w stanie zrozumieć*.

Sposób działania C

Tak naprawdę komputery rozumieją tylko jeden język: kod maszynowy — binarny strumień składający się jedynie z zer i jedynek. Kod napisany w języku C konwertujemy na kod maszynowy za pomocą **kompilatora**.

```
#include <stdio.h>

int main()
{
  puts("C jest czaderskie!");
  return 0;
}
```

rocks.c

```
Plik  Edycja  Okno  Pomoc  Kompilacja
> gcc rocks.c -o rocks
>
```

W systemie Windows ten plik będzie nosił nazwę rocks.exe, a nie rocks.

rocks

1

Kod źródłowy

Zaczynasz od utworzenia pliku źródłowego. Plik źródłowy zawiera kod napisany w języku C, który jest zrozumiały dla człowieka.

2

Kompilacja

Plik źródłowy jest następnie przetwarzany przy użyciu kompilatora. Kompilator sprawdza kod w poszukiwaniu błędów, a kiedy uzna, że wszystko jest w porządku, kompiluje go.

3

Kod wynikowy

Kompilator tworzy nowy plik, nazywany *plikiem wykonywalnym*. Zawiera on kod maszynowy — strumień jedynek i zer, które jest w stanie zrozumieć komputer. I to właśnie jest program, który możesz wykonać.

Język C jest używany, w przypadku gdy duże znaczenie mają szybkość działania, niewielkie rozmiary oraz możliwość przenoszenia. Większość systemów operacyjnych została napisana w języku C. Także większość innych języków programowania została napisana w C. Dodatkowo przeważająca większość gier jest pisana w C.

Istnieją trzy standardy języka C, z którymi można się zetknąć. ANSI C pochodzi z późnych lat 80. i jest używany w najstarszym kodzie. W nowym standardzie — C99, istniejącym od 1999 roku, poprawiono sporo różnych rzeczy. Natomiast w najnowszym standardzie — C11, opracowanym w 2011 roku, dodano kilka nowych, świetnych możliwości. Różnice pomiędzy tymi trzema standardami nie są wielkie, jednak będziemy o nich wspominać.

Zaostrz ołówek

Spróbuj odgadnąć, co robi każdy z tych fragmentów kodu.

Opisz, co według Ciebie robi ten fragment kodu.

```
int card_count = 11;
if (card_count > 10)
  puts("Nowe rozdanie. Licytujemy.");

int c = 10;
while (c > 0) {
    puts("Nie mogę pisać kodu w klasie.");
    c = c - 1;
}

/* Załóż, że imię ma mniej niż 20 znaków. */
char ex[20];
puts("Podaj imię chłopaka: ");
scanf("%19s", ex);
printf("%s.\n\n\tZ nami już koniec.\n", ex);

char suit = 'K';
switch(suit) {
case 'T':
    puts("Trefle");
    break;
case 'K':
    puts("Kara");
    break;
case 'P':
    puts("Piki");
    break;
default:
    puts("Serca");
}
```

Zaostrz ołówek Rozwiązanie

Nie przejmuj się, jeśli jeszcze nie rozumiesz wszystkiego. Zostanie to szczegółowo wyjaśnione w dalszej części książki.

```
int card_count = 11;
if (card_count > 10)
  puts("Nowe rozdanie. Licytujemy.");
```

int oznacza liczbę całkowitą.

To wyświetla łańcuch znaków w wierszu poleceń lub na terminalu.

- Tworzy zmienną całkowitą i przypisuje jej wartość 11.
- Czy wartość zmiennej jest większa od 10?
- Jeśli jest, to wyświetlamy komunikat.

```
int c = 10;
while (c > 0) {
    puts("Nie mogę pisać kodu w klasie.");
    c = c - 1;
}
```

*Nawiasy klamrowe definiują **instrukcję blokową**.*

- Tworzy zmienną całkowitą i przypisuje jej wartość 10.
- Jak długo wartość jest większa od zera...
- ...wyświetlamy komunikat...
- ...i dekrementujemy wartość zmiennej.
- To koniec powtarzanego bloku kodu.

```
/* Załóż, że imię ma mniej niż 20 znaków. */
char ex[20];
puts("Podaj imię chłopaka: ");
scanf("%19s", ex);
printf("%s.\n\n\tZ nami już koniec.\n", ex);
```

To oznacza: „Wszystko, co użytkownik wpisze, zapisz w tablicy ex”.

To wstawi łańcuch znaków w miejsce %s

- To jest komentarz.
- Tworzymy tablicę 20 znaków.
- Wyświetlamy komunikat na ekranie.
- Zapisujemy w tablicy to, co wpisze użytkownik.
- Wyświetlamy komunikat zawierający wpisany tekst.

```
char suit = 'K';
switch(suit) {
case 'T':
    puts("Trefle");
    break;
case 'K':
    puts("Kara");
    break;
case 'P':
    puts("Piki");
    break;
default:
    puts("Serca");
}
```

*Instrukcja **switch** sprawdza zmienną, porównując ją do różnych wartości.*

- Tworzymy zmienną znakową i zapisujemy w niej K.
- Sprawdzamy wartość zmiennej.
- Czy jest nią „T”?
- Jeśli tak, to wyświetlamy słowo „Trefle”.
- Następnie pomijamy pozostałe testy.
- Czy jest nią „K”?
- Jeśli tak, to wyświetlamy słowo „Kara”.
- Następnie pomijamy pozostałe testy.
- Czy jest nią „P”?
- Jeśli tak, to wyświetlamy słowo „Piki”.
- Następnie pomijamy pozostałe testy.
- W przeciwnym razie...
- ...wyświetlamy słowo „Serca”.
- To już koniec sprawdzania.

Ale jak wygląda skompilowany program napisany w C?

Aby stworzyć pełny program, musisz zapisać jego kod w *pliku źródłowym*. Pliki źródłowe programów pisanych w języku C można utworzyć w dowolnym edytorze tekstów, a ich nazwy zazwyczaj mają rozszerzenie *.c*.

To jest tylko konwencja, jednak powinieneś się do niej zastosować.

Przyjrzyjmy się typowemu plikowi źródłowemu programu w języku C.

1 Programy pisane w języku C zazwyczaj zaczynają się od komentarza. Komentarz opisuje przeznaczenie kodu umieszczonego w tym pliku, a dodatkowo może także zawierać informacje o licencji oraz prawach autorskich. Nie ma potrzeby umieszczania komentarza — ani tu, ani w żadnym innym miejscu pliku — jednak jest to dobrym zwyczajem i większość programistów używających języka C będzie takiego komentarza oczekiwać.

Komentarz zaczyna się od znaków /*.

Te znaki * są opcjonalne. Poprawiają tylko wygląd komentarza.

Komentarz kończy się znakami */.

```
/*
 * Ten program służy do liczenia kart.
 * Kod jest udostępniany na warunkach Vegas Public License.
 * (c)2014, Studencki Klub Graczy w Oczko.
 */
```

2 Teraz kolej na sekcję dyrektyw include.
C jest językiem o bardzo, bardzo małych możliwościach… bez zastosowania *bibliotek zewnętrznych* nie da się w nim zrobić praktycznie niczego. Będziesz musiał poinformować kompilator o tym, jakiego kodu zewnętrznego chcesz używać, podając pliki nagłówkowe odpowiednich bibliotek. Plikiem nagłówkowym, którego będziemy używać zdecydowanie najczęściej, jest *stdio.h*. Biblioteka `stdio` zawiera kod pozwalający odczytywać dane z terminala i wyświetlać je w nim.

```
#include <stdio.h>
```

```
int main()
{
    int decks;
    puts("Podaj liczbę talii");
    scanf("%i", &decks);
    if (decks < 1) {
        puts("To nie jest prawidłowa liczba talii");
        return 1;
    }
    printf("Łącznie jest w nich %i kart\n", (decks * 52));
    return 0;
}
```

3 Ostatnią rzeczą, jaką można znaleźć w plikach źródłowych, są funkcje. Cały kod pisany w języku C jest umieszczany wewnątrz funkcji. Najważniejsza funkcja, którą znajdziemy w każdym programie pisanym w tym języku, nosi nazwę **`main()`**. Stanowi ona punkt początkowy, od którego zaczyna się wykonywanie całego kodu programu.

A zatem przyjrzyjmy się funkcji main() nieco bardziej szczegółowo.

Zbliżenie na semantyczne znaczniki

Komputer zacznie wykonywać nasz program od funkcji `main()`. Jej nazwa ma znaczenie: jeśli w kodzie programu nie będzie funkcji `main()`, nie będzie go można wykonać.

Funkcja `main()` **zwraca** wartość **typu** `int`. Co to oznacza? Cóż, kiedy komputer wykonuje program, musi dysponować jakimś sposobem określenia, czy został on wykonany prawidłowo, czy też nie. Robi to, sprawdzając *wartość wynikową* zwracaną przez funkcję `main()`. Jeśli każemy jej zwrócić wartość 0, będzie to oznaczało, że program został wykonany prawidłowo. Jeśli natomiast zwrócimy inną wartość, będzie to oznaczało, że pojawiły się jakieś problemy.

Ponieważ funkcja nosi nazwę „main", to właśnie od niej rozpocznie się wykonywanie programu.

To typ wyniku. W przypadku funkcji main() zawsze powinien to być typ **int**.

Gdyby funkcja miała jakieś parametry, zostałyby one tu wymienione.

```
int main()
{
    int decks;
    puts("Podaj liczbę talii");
    scanf("%i", &decks);
    if (decks < 1) {
        puts("To nie jest prawidłowa liczba talii");
        return 1;
    }
    printf("Łącznie jest w nich %i kart\n", (decks * 52));
    return 0;
}
```

Zawartość funkcji jest zawsze umieszczana pomiędzy **nawiasami klamrowymi**.

Nazwa funkcji jest podawana za typem wyniku. Za nazwą mogą natomiast zostać podane parametry funkcji, oczywiście jeśli w ogóle takie są. Ostatnim elementem funkcji jest jej *zawartość*. Zawartość funkcji **musi** być zapisana wewnątrz pary *nawiasów klamrowych*.

Dla maniaków

Funkcja `printf()` służy do wyświetlania **sformatowanych danych wynikowych**. Zastępuje ona znaki formatujące wartościami zmiennych. Oto przykład:

Pierwszy parametr zostanie wstawiony jako łańcuch znaków.

Pierwszy parametr

```
printf("%s mówi, że poszukiwaną liczbą jest %i","Janek", 21);
```

Drugi parametr zostanie wstawiony jako liczba całkowita.

Drugi parametr

Wywołując funkcję `printf()`, można w niej podać dowolnie wiele parametrów, musimy się jednak przy tym upewnić, że użyliśmy tyle samo znaków formatujących `%`.

Jeśli chcesz sprawdzić status wykonania programu, to w systemie Windows wpisz:

echo %ErrorLevel%

natomiast w systemach Linux lub Mac OS wpisz:

echo $?

Magnesiki z kodem

Członkowie Studenckiego Klubu Gry w Oczko pracowali nad pewnym kodem, układając go z magnesików na lodówce w swoim akademiku, ale jakiś dowcipniś im je pomieszał! Czy byłbyś w stanie odtworzyć ich prawidłowe ułożenie?

```
/*
 * Ten program służy do liczenia kart.
 * Kod jest udostępniany na warunkach Vegas Public License.
 * (c)2014, Studencki Klub Graczy w Oczko.
 */

....................................................

....................................................

.......... main()

{
    char card_name[3];
    puts("Wpisz symbol karty (card_name): ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {

        ..........................................
    } else if (card_name[0] == ............) {
        val = 10;

    } .............(card_name[0] == ...........) {


        ..........................................
    } else {
        val = atoi(card_name);
    }
    printf("Wartość karty to: %i\n", val);

    ............. 0;
}
```

Wpisz dwuliterowy symbol określający nazwę karty.

Ta funkcja konwertuje tekst na liczbę.

Magnesiki: `<stdlib.h>`, `;`, `;`, `val = 11`, `int`, `'J'`, `#include`, `'A'`, `return`, `else`, `#include`, `if`, `val = 10`, `<stdio.h>`

Magnesiki z kodem. Rozwiązanie

Członkowie Studenckiego Klubu Gry w Oczko pracowali nad pewnym kodem, układając go z magnesików na lodówce w swoim akademiku, ale jakiś dowcipniś im je pomieszał! Twoim zadaniem było ponowne poukładanie magnesików.

```
/*
 * Ten program służy do liczenia kart.
 * Kod jest udostępniany na warunkach Vegas Public License.
 * (c)2014, Studencki Klub Graczy w Oczko.
 */
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char card_name[3];
    puts("Wpisz symbol karty (card_name): ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {
        val = 10;
    } else if (card_name[0] == 'J') {
        val = 10;
    } else if (card_name[0] == 'A') {
        val = 11;
    } else {
        val = atoi(card_name);
    }
    printf("Wartość karty to: %i\n", val);
    return 0;
}
```

Nie istnieją głupie pytania

P: Co oznacza `card_name[0]`?

O: To pierwszy znak wpisany przez użytkownika. Jeśli zatem użytkownik wpisał 10, to wartością wyrażenia `card_name[0]` będzie 1.

P: Czy zawsze zapisujecie komentarze, używając sekwencji znaków /* oraz */?

O: Jeśli używany kompilator obsługuje standard C99, to komentarze można także zapisywać za sekwencją znaków //. W takim przypadku kompilator uzna za komentarz całą zawartość wiersza umieszczoną za tymi znakami.

P: Skąd mam wiedzieć, jaki standard obsługuje mój kompilator?

O: Sprawdź dokumentację kompilatora. Kompilator gcc obsługuje wszystkie trzy standardy: ANSI C, C99 oraz C11.

A jak można uruchomić program?

C jest *językiem kompilowanym*. Oznacza to, że komputer nie zinterpretuje kodu bezpośrednio. Zamiast tego musimy przekonwertować — czy też *skompilować* — zrozumiały dla nas, ludzi, kod źródłowy na *kod maszynowy*, który z kolei jest w stanie zrozumieć komputer.

Aby skompilować kod źródłowy, będzie nam potrzebny program nazywany **kompilatorem**. Jednym z najbardziej popularnych kompilatorów jest *GNU Compiler Collection*, nazywany też skrótowo **gcc**. Jest on dostępny w wielu systemach operacyjnych i pozwala na kompilowanie kodu napisanego nie tylko w C, lecz także w wielu innych językach programowania. A co najważniejsze, można go używać za darmo.

Oto, w jaki sposób można skompilować i uruchomić program przy użyciu gcc:

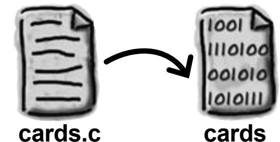
1. **Zapisz kod z ćwiczenia Magnesiki z kodem, umieszczonego na poprzedniej stronie, w pliku cards.c.**

Pliki źródłowe zazwyczaj mają rozszerzenie .c.

2. **Skompiluj plik, wykonując następujące polecenie w wierszu poleceń lub w oknie terminala: `gcc cards.c -o cards`.**

Kompilujemy plik cards.c na plik o nazwie cards.

```
Plik  Edycja  Okno  Pomoc  Kompilacja
> gcc cards.c -o cards
>
```



Jeśli używamy systemu Windows, to ten plik będzie nosił nazwę cards.exe.

3. **Uruchom program, wydając polecenie `cards` w systemie Windows lub `./cards` w systemie Linux bądź na komputerach Mac.**

```
Plik  Edycja  Okno  Pomoc  Kompilacja
> ./cards
Wpisz symbol karty (card_name):
```

Dla maniaków

Na większości komputerów program można jednocześnie skompilować i uruchomić, posługując się następującą sztuczką:

Umieszczone w tym poleceniu dwa znaki && oznaczają: „a następnie, jeśli wszystko się uda, zrób...”.

```
gcc zork.c –o zork && ./zork
```

Na komputerach z zainstalowanym systemem Windows należy użyć „zork” zamiast „./zork”.

To polecenie spowoduje uruchomienie nowego programu wyłącznie wtedy, gdy uda się go poprawnie skompilować. Jeśli podczas kompilacji wystąpią jakieś problemy, program nie zostanie wykonany, a na ekranie pojawią się informacje o błędach.

Zrób to sam!

Teraz powinieneś samemu utworzyć i skompilować plik *cards.c*. W dalszej części rozdziału będziemy go jeszcze wielokrotnie modyfikować.

Jazda próbna

Sprawdźmy, czy uda się skompilować i uruchomić program. Otwórz okno wiersza poleceń lub terminala na swoim komputerze i przekonaj się sam.

To polecenie kompiluje kod i tworzy program cards.

To polecenie wykonuje program. Jeśli używasz systemu Windows, pomiń znaki ./.

Uruchamiamy program jeszcze raz.

Użytkownik wpisuje nazwę karty...

...a program wyświetla odpowiednią wartość.

Pamiętaj: kompilację i uruchomienie programu można połączyć i wykonać w jednym kroku (zerknij na poprzednią stronę, by zobaczyć, jak to zrobić).

```
Plik  Edycja  Okno  Pomoc  21
> gcc cards.c -o cards
> ./cards
Wpisz symbol karty (card_name):
Q
Wartość karty to: 10
> ./cards
Wpisz symbol karty (card_name):
A
Wartość karty to: 11
> ./cards
Wpisz symbol karty (card_name):
7
Wartość karty to: 7
```

Program działa!

Gratulujemy! Właśnie skompilowałeś i uruchomiłeś program napisany w C. Kompilator `gcc` przekonwertował kod źródłowy programu zapisany w pliku *cards.c* i zrozumiały dla ludzi na *kod maszynowy* programu `cards`, który z kolei może zrozumieć komputer. Jeśli korzystasz z komputera Mac lub systemu Linux, kompilator wygeneruje kod maszynowy programu, zapisując go w pliku ***cards***. Z kolei w systemie Windows, w którym wszystkie programy muszą mieć rozszerzenie *.exe*, program ten będzie nosił nazwę ***cards.exe***.

Nie istnieją głupie pytania

P: Dlaczego na komputerach Mac oraz w systemie Linux muszę poprzedzać nazwę programu znakami ./?

O: W systemach operacyjnych wzorowanych na systemie UNIX programy są uruchamiane wyłącznie wtedy, gdy zostanie określony katalog, w którym się znajdują, bądź też gdy katalog ten zostanie podany w zmiennej środowiskowej PATH.

Sam język C nie udostępnia łańcuchów znaków.

Istnieje jednak kilka bibliotek języka C, które **udostępniają** łańcuchy znaków.

C działa na niższym poziomie niż większość pozostałych języków programowania, dlatego też zamiast łańcuchów znaków używa zazwyczaj czegoś zbliżonego: *tablicy pojedynczych znaków*. Jeśli pisałeś już programy w innych językach, to zapewne wiesz, czym są tablice. Tablica jest po prostu listą zawierającą jakieś elementy i posiadającą jedną nazwę. A zatem `card_name` jest nazwą zmiennej umożliwiającą odwoływanie się do listy znaków wpisanych przez użytkownika w wierszu poleceń. Zmienną `card_name` zdefiniowałeś jako *tablicę składającą się z dwóch znaków*, a zatem do pierwszego oraz drugiego z nich możesz się odwołać, używając następujących wyrażeń: `card_name[0]` oraz `card_name[1]`. Aby się przekonać, jak one działają, zajrzyjmy nieco głębiej do pamięci komputera i zobaczmy, w jaki sposób język C obsługuje teksty…

Łańcuchy pod mikroskopem

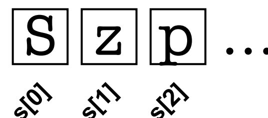
Łańcuchy znaków są jedynie tablicami znaków. Kiedy kompilator C zobaczy następujący łańcuch:

```
s = "Szpadel"
```

odczytuje go tak, jak gdyby stanowił on tablicę niezależnych znaków:

Właśnie w taki sposób definiuje się łańcuchy znaków w języku C.

```
s = { 'S', 'z', 'p', 'a', 'd', 'e', 'l'}
```



Każdy ze znaków w łańcuchu jest elementem tablicy; to właśnie dlatego można się do nich odwoływać przy użyciu indeksów, na przykład: `s[0]` oraz `s[1]`.

Nie wypadnijmy poza koniec łańcucha

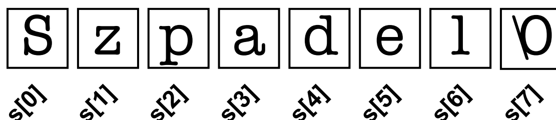
Ale co się dzieje, kiedy program chce odczytać zawartość łańcucha? Na przykład kiedy będzie chciał go wyświetlić. W wielu językach programowania komputer bardzo dokładnie śledzi i pamięta długość tablicy, jednak w porównaniu z wieloma innymi językami C jest językiem niższego poziomu i dlatego nie zawsze jest w stanie określić, *jak długa* jest tablica. Skoro zatem C ma wyświetlić łańcuch znaków na ekranie, musi wiedzieć, kiedy dotrze do końca tablicy znaków. I właśnie do tego służy specjalny znak nazywany **wartownikiem**.

Wartownik to dodatkowy znak umieszczany na końcu łańcucha; ma on wartość `\0`. Kiedy komputer musi odczytać zawartość łańcucha, pobiera kolejne znaki z tablicy, aż dotrze do znaku `\0`. To oznacza, że gdy komputer zobaczy następujący łańcuch znaków:

```
s = "Szpadel"
```

C wie, że po dotarciu do znaku \0 musi się zatrzymać.

to w rzeczywistości w pamięci będzie on zapisany w poniższy sposób:



\0 to znak kodu ASCII o wartości 0.

Programiści piszący w języku C często nazywają go znakiem NULL.

To właśnie z tego względu zmienną `card_name` musieliśmy zdefiniować w następujący sposób:

```
char card_name[3];
```

W zmiennej `card_name` będziemy zapisywać wyłącznie łańcuchy składające się z jednego lub dwóch znaków, jednak ze względu na to, że łańcuchy muszą się kończyć *wartownikiem*, w tablicy musi się znaleźć miejsce dla dodatkowego znaku.

Nie istnieją głupie pytania

P: Dlaczego znaki są numerowane od 0, a nie od 1?

O: Indeks określa przesunięcie; stanowi on miarę odległości od pierwszego znaku.

P: Dlaczego?

O: Komputer będzie przechowywać znaki w sąsiadujących ze sobą bajtach pamięci. Może zatem użyć indeksu do określenia położenia znaku. Jeśli komputer wie, że `c[0]` znajduje się w komórce o numerze 1000000, może bardzo szybko wyliczyć, iż znak `c[96]` będzie się znajdował w komórce 1000000 + 96.

P: A dlaczego potrzebny jest znak wartownika? Czy C nie wie, jak długie są tablice?

O: Zazwyczaj nie wie. Język C nie jest zbyt dobry w zapamiętywaniu długości tablic, a łańcuchy znaków są zwyczajnymi tablicami.

P: C nie wie, jak długie są tablice?

O: Nie. Czasami kompilator może określić ich długość na podstawie analizy kodu, jednak zazwyczaj w języku C za kontrolę długości tablic odpowiada programista.

P: Czy zapisywanie znaków w cudzysłowach lub apostrofach ma jakieś znaczenie?

O: Tak. Apostrofy są używane do zapisywania pojedynczych znaków, natomiast cudzysłowy służą wyłącznie do zapisu łańcuchów znaków.

P: Czy definiując łańcuchy, powinienem zatem zapisywać je w cudzysłowach ("), czy jawnie podawać jako tablicę znaków?

O: Zazwyczaj definiujemy łańcuchy znaków, zapisując je w cudzysłowach. Nazywamy je wtedy **literałami łańcuchowymi**. Zapisywanie łańcuchów w takiej postaci jest znacznie prostsze.

P: Czy istnieje jakakolwiek różnica pomiędzy literałami łańcuchowymi a tablicami znaków?

O: Tylko jedna: literały łańcuchowe są stałymi.

P: A co to znaczy?

O: Oznacza to, że po utworzeniu literału jego znaków nie można zmieniać.

P: A co się stanie, kiedy spróbuję to zrobić?

O: To zależy od kompilatora. W przypadku `gcc` zostanie zgłoszony błąd magistrali.

P: Błąd magistrali? A co to niby znaczy?

O: Język C przechowuje literały łańcuchowe w pamięci w inny sposób. Błąd magistrali oznacza, że program nie może zmodyfikować fragmentu pamięci.

Bezbolesne operacje

Nie wszystkie znaki równości są sobie równe.

W języku C znak równości (=) jest używany do **przypisywania**. Natomiast dwa znaki równości (==) służą do **testowania równości**.

```
teeth = 4;
```
Przypisujemy zmiennej teeth wartość 4.

```
teeth == 4;
```
Sprawdzamy, czy wartość zmiennej teeth jest równa 4.

Jeśli chcemy zwiększyć lub zmniejszyć wartość zmiennej, to używając skróconych operatorów przypisania += oraz -=, możemy zaoszczędzić trochę miejsca:

```
teeth += 2;
```
Dodajemy 2 do zmiennej teeth.

```
teeth -= 2;
```
Odejmujemy 2 od wartości zmiennej teeth.

I w końcu, jeśli chcemy zwiększyć lub zmniejszyć wartość zmiennej o 1, możemy się posłużyć operatorami ++ lub --.

```
teeth++;
```
Powiększamy o 1.

```
teeth--;
```
Zmniejszamy o 1.

Dwa rodzaje poleceń

Wszystkie polecenia, z jakimi do tej pory się spotkałeś, można było zaliczyć do jednej z dwóch następujących kategorii.

Zrób coś

Większość poleceń w języku C to instrukcje. Proste instrukcje są *akcjami*; coś *robią* lub coś *zwracają*. Spotkałeś już instrukcje definiujące zmienne, pobierające dane wejściowe z klawiatury lub wyświetlające informacje na ekranie.

```
split_hand();
```

To jest prosta instrukcja.

Czasami grupujemy instrukcje, tworząc tak zwane *instrukcje blokowe*. Instrukcje blokowe to grupy instrukcji zapisane wewnątrz nawiasów klamrowych.

```
{
    deal_first_card();
    deal_second_card();
    cards_in_hand = 2;
}
```

Te polecenia tworzą ***instrukcję blokową****, gdyż są zapisane wewnątrz pary nawiasów klamrowych.*

Zrób coś, wyłącznie jeśli warunek jest prawdziwy

Instrukcje sterujące, takie jak `if`, przed wykonaniem kodu sprawdzają warunek:

```
if (value_of_hand <= 16)
    hit();
else
    stand();
```

To jest warunek.

Wykonaj tę instrukcję, jeśli warunek zostanie spełniony.

Wykonaj tę instrukcję, jeśli warunek nie zostanie spełniony.

Zazwyczaj, jeśli warunek w instrukcji `if` zostanie spełniony, będziemy chcieli wykonać więcej niż jedną instrukcję; dlatego też w instrukcjach warunkowych często umieszcza się instrukcje blokowe:

```
if (dealer_card == 6) {
    double_down();
    hit();
}
```

Jeśli warunek będzie spełniony, zostaną wykonane OBIE instrukcje. Zostały one umieszczone wewnątrz jednej instrukcji blokowej.

Czy nawiasy są potrzebne?

Dzięki instrukcjom blokowym całe *grupy instrukcji* można traktować jako *jedną*. W języku C instrukcja `if` działa w następujący sposób:

```
if (countdown == 0)
    do_this_thing();
```

Po sprawdzeniu warunku jest wykonywana **jedna instrukcja**. Co zatem zrobić, jeśli po sprawdzeniu warunku chcielibyśmy wykonać kilka instrukcji? Jeśli serię instrukcji zapiszemy w nawiasach klamrowych, to C potraktuje je tak, jak gdyby stanowiły jedną instrukcję:

```
if (x == 2) {
    call_whitehouse();
    sell_oil();
    x = 0;
}
```

Programiści używający języka C starają się, by ich kod był krótki i zwięzły, dlatego też większość z nich będzie pomijać niepotrzebne nawiasy klamrowe w instrukcjach warunkowych (np. `if`) i pętlach (np. `while`). Dlatego zamiast:

```
if (x == 2) {
    puts("Zrób coś!");
}
```

większość programistów C napisze:

```
if (x == 2)
    puts("Zrób coś!");
```

Oto kod, jakim aktualnie dysponujemy

```
/*
 * Ten program służy do liczenia kart.
 * Kod jest udostępniany na warunkach Vegas Public License.
 * (c)2014, Studencki Klub Graczy w Oczko.
 */

#include  <stdio.h>
#include  <stdlib.h>

int main()
{
    char card_name[3];
    puts("Wpisz symbol karty (card_name): ");
    scanf("%2s", card_name);
    int val = 0;

    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {
        val = 10;
    } else if (card_name[0] == 'J') {
        val = 10;
    } else  if  (card_name[0] == 'A') {
        val = 11;
    } else {
        val = atoi(card_name);
    }
    printf("Wartość karty to: %i\n", val);
    return 0;
}
```

Mam pomysł. Czy ten program mógłby sprawdzać, czy wartość karty mieści się w określonym zakresie? To mogłoby być przydatne...

Reklama

MOGĘ SPRAWIĆ, ŻE BĘDZIESZ BOGATY. TAK JAK JA!

Korespondencyjna szkoła gry w oczko Edka Melona

Siema! Jak leci? Wyglądasz mi na sprytnego gościa. Kto jak kto, ale ja to wiem, bo sam jestem sprytnym gościem! Słuchaj, mam tu świetny interes, a ponieważ jestem też miłym gościem, pozwolę Ci w niego wejść. Bo wiesz, jestem ekspertem w liczeniu kart. Capo di tutti capi. A co to jest liczenie kart, spytasz? Cóż, z mojego punktu widzenia to jest kariera!

Mówiąc poważnie, liczenie kart jest zwiększeniem twoich szans podczas gry w oczko. Grając w oczko, gdy na stole pozostaje sporo kart o wysokiej wartości, szanse gracza są nieco większe. I to jest twoja okazja!

Liczenie kart pomaga ci śledzić, ile wysokich kart pozostaje jeszcze w grze. Załóżmy, że zaczynasz liczyć od wartości 0.

Najpierw krupier wydaje królową — to wysoka karta. A zatem w talii jest już o jedną wysoką kartę mniej, czyli ich liczba jest pomniejszana o jeden:

To królowa -> liczba – 1

Jeśli jednak krupier wyda niską kartę, na przykład 4, to liczba jest powiększana:

To czwórka -> liczba + 1

Za wysokie karty uznajemy dziesiątkę oraz waleta, damę i króla. Kartami niskimi są: trójki, czwórki, piątki i szóstki.

W taki sposób postępujemy w przypadku wszystkich kart wysokich i niskich, aż liczba stanie się naprawdę wysoka, a wtedy w następnym zakładzie możesz położyć dużo kasy i ta-dam! Niebawem będziesz miał więcej kasy niż moja trzecia żona!

Jeśli chcesz nauczyć się jeszcze więcej, to już dziś zapisz się na mój Korespondencyjny Kurs Gry w Oczko i dowiedz się znacznie więcej nie tylko o liczeniu kart, lecz także o tym:

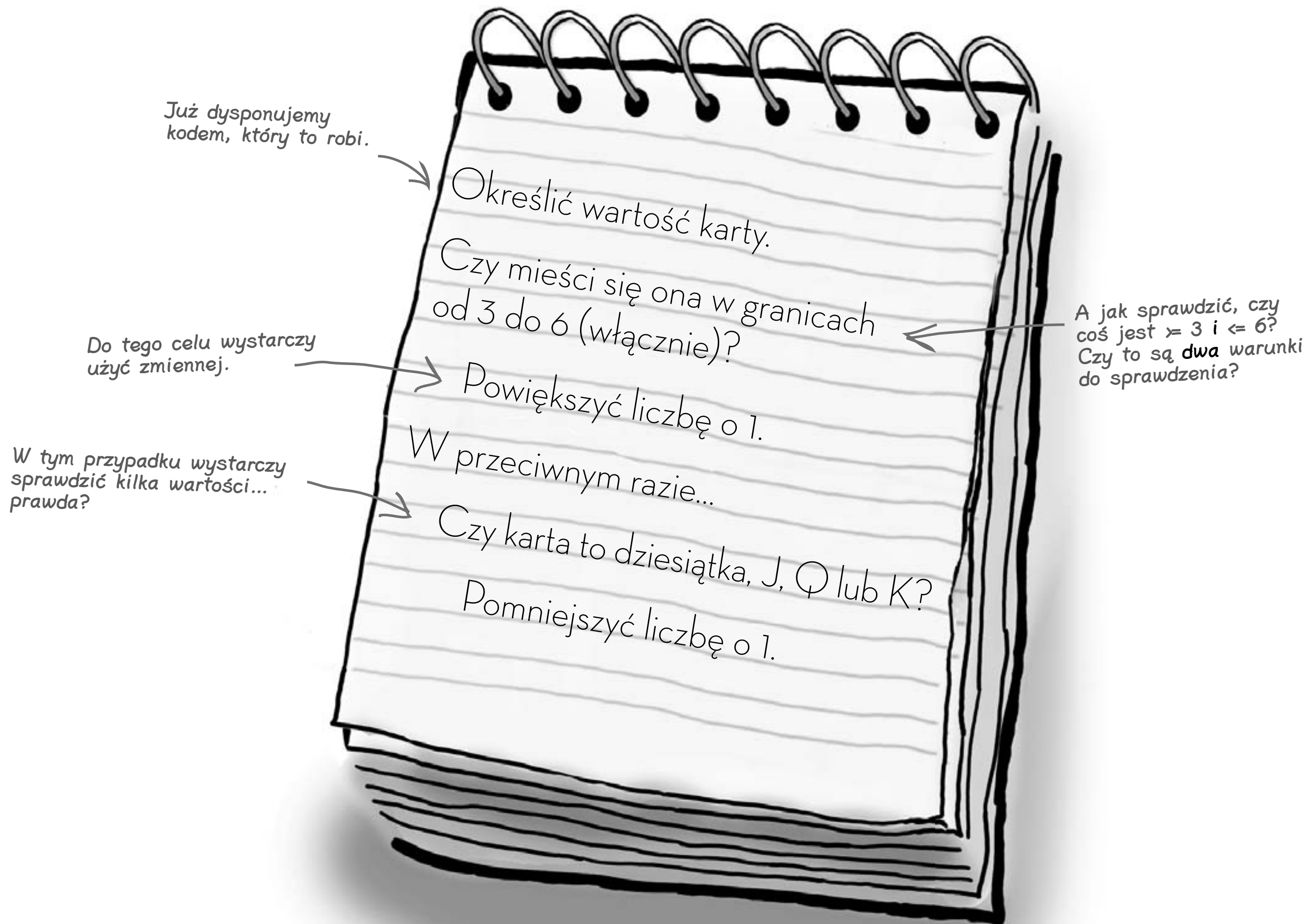
* Jak skorzystać z kryterium Kelly'ego, by zmaksymalizować wartość zakładów.
* Jak uniknąć kompletnej porażki w grze w wojnę.
* Jak się pozbyć tłustych plam z jedwabnego garnituru.
* Co pasuje do ubrania w kratkę.

Więcej informacji można uzyskać od Kuzyna Józka, wicedyrektora Korespondencyjnego Kursu Gry w Oczko.

Kup Edsela

Liczenie kart? W języku C?

Liczenie kart to sposób na powiększenie swoich szans w grze w oczko. Zliczając zagrane karty, gracz może określić najodpowiedniejszy moment do zrobienia dużego zakładu lub określić, kiedy lepiej grać o małe stawki. Mimo, że jest to metoda dająca duże możliwości, to jest ona jednocześnie całkiem prosta.



Jak trudno byłoby napisać taki program w języku C? Dowiedziałeś się już, jak sprawdzić jeden warunek, jednak nasz algorytm liczenia kart wymaga sprawdzenia dwóch warunków: musimy sprawdzić, czy wartość karty jest >= 3 oraz czy jest <= 6.

Musimy zatem poznać operacje, które pozwolą nam łączyć ze sobą warunki.

Wartości logiczne to nie tylko sprawdzanie równości

Jak na razie zetknąłeś się wyłącznie z instrukcją `if`, która kontrolowała, czy został sprawdzony pojedynczy warunek. A co zrobić, jeśli trzeba sprawdzić kilka warunków? Albo zrobić coś, gdy warunek *nie zostanie* sprawdzony?

Operator && sprawdza, czy dwa warunki są prawdziwe

Operator *logicznej koniunkcji* (&&) zwraca prawdę wyłącznie wtedy, gdy **oba** łączone warunki są spełnione.

```
if ((dealer_up_card == 6) && (hand == 11))
    double_down();
```

Aby funkcja została wywołana, muszą być spełnione **oba** warunki.

Operator && działa wydajnie — jeśli pierwszy warunek nie zostanie spełniony, komputer nie będzie sobie zaprzątał głowy przetwarzaniem drugiego. Doskonale bowiem wie, że jeśli pierwszy warunek nie został spełniony, to całe wyrażenie też nie będzie.

Operator || sprawdza, czy jest spełniony jeden z dwóch warunków

Operator *logicznej alternatywy* (||) zwraca prawdę, jeśli **którykolwiek** z warunków jest spełniony.

```
if (cupcakes_in_fridge || chips_on_table)
    eat_food();
```

Którykolwiek z tych warunków powinien być spełniony.

Jeśli pierwszy warunek będzie spełniony, komputer nie będzie sobie zawracał głowy sprawdzaniem drugiego. Doskonale bowiem wie, że jeśli pierwszy warunek został spełniony, to także *całe wyrażenie* musi być prawdziwe.

Operator ! odwraca wartość warunku

! to logiczny operator *przeczenia*. Zmienia on wartość warunku na przeciwną.

```
if (!brad_on_phone)
    answer_phone();
```

! oznacza „nie”

Dla maniaków

W języku C wartości logiczne są reprezentowane przy użyciu liczb. W C wartość 0 reprezentuje logiczny fałsz — nieprawdę. A jaka wartość reprezentuje logiczną prawdę? Logiczną prawdą jest każda wartość różna od zera. A zatem w języku C poniższy fragment kodu jest całkowicie poprawny:

```
int people_moshing = 34;
if (people_moshing)
    take_off_glasses();
```

W rzeczywistości w kodzie pisanym w języku C taki zapis jest często używany, gdyż pozwala w szybki sposób sprawdzić, czy wartość zmiennej jest różna od zera.

Ćwiczenie

Masz zmodyfikować program w taki sposób, by można go było używać do liczenia kart. Będzie on musiał wyświetlać jeden komunikat, jeśli wartość karty mieści się w zakresie od 3 do 6, oraz drugi komunikat, jeśli karta jest dziesiątką, waletem, damą lub królem.

```
int main()

{
    char card_name[3];
    puts("Wpisz symbol karty (card_name): ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {
        val = 10;
    } else if (card_name[0] == 'J') {
        val = 10;
    } else  if  (card_name[0] == 'A') {
        val = 11;
    } else {
        val = atoi(card_name);
    }

    /* Sprawdzamy, czy wartość mieści się w zakresie od 3 do 6 */
    if .........................................
        puts("Liczba została powiększona");
    /* Jeśli nie, to sprawdzamy, czy karta to: 10, J, Q lub K */
    else if  ....................................
        puts("Liczba została zmniejszona");

    return 0;
}
```

Grzeczny przewodnik po standardach

W standardzie ANSI C nie ma żadnych specjalnych wartości reprezentujących logiczną prawdę oraz logiczny fałsz. Programy pisane w C traktują zatem 0 jako wartość reprezentującą logiczny fałsz oraz dowolną inną liczbę jako logiczną prawdę. Standard C99 pozwala stosować w kodzie źródłowym słowa kluczowe `true` oraz `false`, jednak kompilator zamienia je odpowiednio na `1` oraz `0`.

Ćwiczenie Rozwiązanie

Miałeś za zadanie zmodyfikować program w taki sposób, by można go było używać do liczenia kart. Miał on wyświetlać jeden komunikat, jeśli wartość karty mieściła się w zakresie od 3 do 6, oraz drugi — jeśli karta była dziesiątką, waletem, damą lub królem.

```
int main()

{
    char card_name[3];
    puts("Wpisz symbol karty (card_name): ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {
        val = 10;
    } else if (card_name[0] == 'J') {
        val = 10;
    } else  if  (card_name[0] == 'A') {
        val = 11;
    } else {
        val = atoi(card_name);
    }

    /* Sprawdzamy, czy wartość mieści się w zakresie od 3 do 6 */
    if ((val > 2) && (val < 7))
        puts("Liczba została powiększona");
    /* Jeśli nie, to sprawdzamy, czy karta to: 10, J, Q lub K */
    else if (val == 10)
        puts("Liczba została zmniejszona");

    return 0;
}
```

Ten warunek można zapisać na kilka różnych sposobów.

Czy zauważyłeś, że w tym miejscu wystarczy jeden warunek?

Nie istnieją głupie pytania

P: Dlaczego nie mogę użyć & ani |?

O: Jeśli chcesz, to możesz użyć tych operatorów. Operatory & oraz | **zawsze przetwarzają oba warunki**, natomiast w przypadku użycia operatorów && i || często okazuje się, że drugi warunek nie musi być przetwarzany.

P: Po co zatem istnieją operatory & oraz |?

O: Ponieważ nie służą one wyłącznie do wyznaczania warunków logicznych. Wykonują one operacje bitowe na poszczególnych bitach liczb.

P: Słucham? Co masz na myśli?

O: No dobrze. Otóż 6 & 4 zwraca wynik 4, gdyż gdybyś porównał dwójkowy zapis liczb 6 (110 w zapisie dwójkowym) i 4 (100 w zapisie dwójkowym) i sprawdził, na których miejscach cyfry dwójkowe są takie same, to uzyskałbyś liczbę 4 (100 w zapisie dwójkowym).

Jazda próbna

Zobaczmy, co się stanie, kiedy skompilujemy i uruchomimy nasz program:

To polecenie kompiluje i wykonuje program.

Wykonujemy program kilka razy, by sprawdzić, czy różne zakresy wartości działają prawidłowo.

```
Plik  Edycja  Okno  Pomoc  PiątkaPik
> gcc cards.c -o cards && ./cards
Wpisz symbol karty (card_name):
Q
Liczba została zmniejszona

> ./cards
Wpisz symbol karty (card_name):
8

> ./cards
Wpisz symbol karty (card_name):
3
Liczba została powiększona

>
```

Nasz kod działa dobrze. Dzięki połączeniu dwóch warunków przy użyciu operatora logicznego możemy sprawdzać całe zakresy, a nie konkretne wartości.
Teraz dysponujemy już podstawową strukturą programu liczącego karty.

Kompilator bez tajemnic

W tym tygodniu tematem wywiadu jest:

Czy gcc kiedykolwiek coś dla nas zrobił?

Rusz Głową: Zacznę od podziękowań, że w swoim napiętym harmonogramie znalazłeś nieco czasu na wywiad z nami.

gcc: Stary, to naprawdę żaden problem. Cała przyjemność po mojej stronie.

Rusz Głową: `gcc`, powiedz, czy to prawda, że potrafisz mówić w wielu językach?

gcc: Płynnie posługuję się ponad sześcioma milionami form komunikacji…

Rusz Głową: Naprawdę?

gcc: Nie, żartuję. Ale faktycznie znam kilka języków. Oczywiście jest to C, lecz oprócz niego także C++ oraz Objective-C. Radzę sobie także z Pascalem, Fortranem, PL/I i kilkoma innymi. A… no i liznąłem także nieco Go.

Rusz Głową: A jeśli chodzi o stronę sprzętową, to jesteś w stanie generować kod maszynowy na bardzo wiele różnych platform, czy tak?

gcc: Tak, praktycznie na każdy procesor. Właściwie, kiedy projektanci tworzą nowy rodzaj procesora, to pierwszą rzeczą, na jakiej im zależy, jest uruchomienie na nim mnie w jakiejś postaci.

Rusz Głową: W jaki sposób udało ci się osiągnąć tak niesamowitą elastyczność?

gcc: Przypuszczam, że tajemnicą mojego sukcesu są dwie strony mojej osobowości. Strona zewnętrzna rozumie pewne rodzaje kodu źródłowego.

Rusz Głową: Napisanego w takim języku jak C?

gcc: Właśnie tak. Ta zewnętrzna strona mojej osobowości potrafi konwertować ten język na kod pośredni. Wszystkie moje osobowości rozumiejące różne języki generują ten sam rodzaj kodu.

Rusz Głową: Wspominałeś o dwóch stronach swojej osobowości.

gcc: Mam także stronę wewnętrzną: system służący do konwertowania kodu pośredniego na kod maszynowy, który jest rozumiany przez wiele różnych platform sprzętowych. Trzeba jeszcze do tego dodać znajomość konkretnych formatów plików wykonywalnych stosowanych w niemal wszystkich systemach operacyjnych, o których kiedykolwiek słyszałeś…

Rusz Głową: A jednak, pomimo tego wszystkiego, czasami uważają cię za zwyczajny translator. Czy uważasz, że to jest sprawiedliwe? Przecież bez wątpienia to nie wszystko, czym jesteś.

gcc: No cóż, oczywiście, że robię znacznie więcej niż zwyczajne tłumaczenie. Potrafię na przykład wykrywać błędy w kodzie.

Rusz Głową: Takie jak?

gcc: Z rzeczy oczywistych — potrafię wykrywać błędy w nazwach zmiennych. Jednak wykrywam także błędy o bardziej subtelnej naturze, takie jak powtórzona definicja jakiejś zmiennej. Umiem także ostrzegać programistów, by nie nadawali jakiejś zmiennej takiej samej nazwy, jaką ma już istniejąca funkcja i tak dalej.

Rusz Głową: A zatem potrafisz także dbać o jakość kodu?

gcc: O, tak. I nie tylko o jakość, lecz także o wydajność jego działania. Jeśli wykryję, że fragment kodu umieszczony wewnątrz pętli mógłby równie dobrze działać poza nią, mogę go błyskawicznie tam przenieść.

Rusz Głową: Czyli w rzeczywistości robisz całkiem sporo!

gcc: Tak właśnie uważam. Ale robię to po cichu.

Rusz Głową: `gcc`, dziękuję ci bardzo.

BĄDŹ kompilatorem

Każdy z fragmentów kodu, które są przedstawione na tej stronie, reprezentuje kompletny plik źródłowy programu napisanego w języku C. Twoim zadaniem jest wcielić się w rolę kompilatora i określić, czy każdy z nich uda się skompilować, a jeśli nie, to dlaczego. Aby zdobyć dodatkowe punkty, powiedz, jakie według Ciebie będą wyniki wygenerowane przez skompilowany program oraz czy działają one zgodnie z przeznaczeniem.

A

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1)
        card = card - 1;
        if (card < 7)
            puts("Niska karta");
    else {
        puts("As!");
    }
    return 0;
}
```

B

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1) {
        card = card - 1;
        if (card < 7)
            puts("Niska karta");
    else
        puts("As!");
    }
    return 0;
}
```

C

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1) {
        card = card - 1;
        if (card < 7)
            puts("Niska karta");
    } else
        puts("As!");

    return 0;
}
```

D

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1) {
        card = card - 1;
        if (card < 7)
            puts("Niska karta");
    else
        puts("As!");

    return 0;
}
```

BĄDŹ kompilatorem. Rozwiązanie

Każdy z fragmentów kodu, które są przedstawione na tej stronie, reprezentuje kompletny plik źródłowy programu napisanego w języku C. Twoim zadaniem jest wcielić się w rolę kompilatora i określić, czy każdy z nich uda się skompilować, a jeśli nie, to dlaczego. Aby zdobyć dodatkowe punkty, powiedz, jakie według Ciebie będą wyniki wygenerowane przez skompilowany program oraz czy działają one zgodnie z przeznaczeniem.

A

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1)
        card = card - 1;
        if (card < 7)
           puts("Niska karta");
    else {
        puts("As!");
    }
    return 0;
}
```

Kod można skompilować. Program wyświetla komunikat „Niska karta”. Kod nie działa jednak prawidłowo, gdyż klauzula else jest połączona z nieodpowiednią instrukcją if.

B

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1) {
        card = card - 1;
        if (card < 7)
            puts("Niska karta");
    else
        puts("As!");
    }
    return 0;
}
```

Kod można skompilować. Program nic nie wyświetla i nie działa prawidłowo, gdyż klauzula else jest dołączona do niewłaściwej instrukcji if.

C

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1) {
        card = card - 1;
        if (card < 7)
            puts("Niska karta");
    } else
        puts("As!");

    return 0;
}
```

Kod można skompilować. Program wyświetla komunikat „As!” i jest prawidłowo napisany.

D

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1) {
        card = card - 1;
        if (card < 7)
            puts("Niska karta");
    else
        puts("As!");

    return 0;
}
```

Tego programu nie uda się skompilować, gdyż brakuje jednego nawiasu klamrowego.

Jak aktualnie wygląda nasz kod?

```
int main()

{
    char card_name[3];
    puts("Wpisz symbol karty (card_name): ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {
        val = 10;
    } else if (card_name[0] == 'J') {
        val = 10;
    } else  if  (card_name[0] == 'A') {
        val = 11;
    } else {
        val = atoi(card_name);
    }

    /* Sprawdzamy, czy wartość mieści się w zakresie od 3 do 6 */
    if ((val > 2) && (val < 7))
        puts("Liczba została powiększona");
    /* Jeśli nie, to sprawdzamy, czy karta to: 10, J, Q lub K */
    else if  (val == 10)
        puts("Liczba została zmniejszona");

    return 0;
}
```

Hm... A czy można coś zrobić z tą sekwencją instrukcji if? Wszystkie one sprawdzają tę samą wartość, konkretnie card–name[0], a większość z nich także robi to samo – przypisuje zmiennej val wartość 10. Zastanawiam się, czy w języku C można wyrazić to w jakiś bardziej efektywny sposób.

Często zdarza się, że programy pisane w języku C muszą kilkakrotnie sprawdzać tę samą wartość, a następnie dla każdego przypadku wykonywać bardzo podobne fragmenty kodu.

Teraz możesz używać sekwencji instrukcji `if` i najprawdopodobniej wszystko będzie działało bardzo dobrze. Jednak język C pozwala na zapisanie takiej logiki działania także w innej postaci.

W języku C testy logiczne można także wykonywać, używając instrukcji switch.

Pociąg do Switcherado

Czasami, kiedy tworzymy kod z logiką warunkową, konieczne jest wielokrotne sprawdzanie wartości tej samej zmiennej. Aby uniknąć konieczności pisania wielu instrukcji `if`, w języku C wprowadzono alternatywne rozwiązanie: instrukcję **`switch`**.

Instrukcja `switch` jest nieco podobna do `if`, jednak w odróżnieniu od niej pozwala na testowanie wielu wartości *jednej zmiennej*:

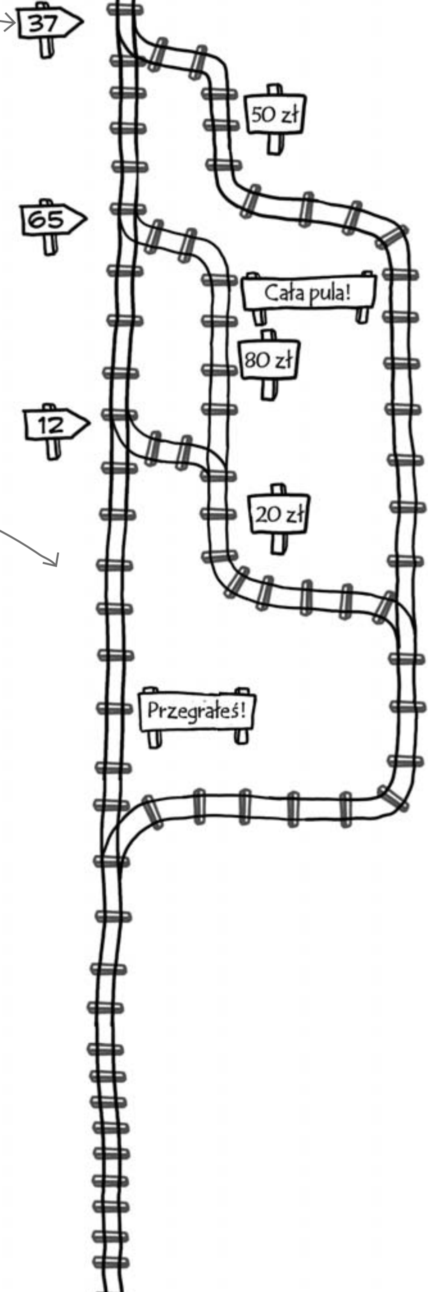
```
switch(train) {
case 37:
  winnings = winnings + 50;
  break;
case 65:
  puts("Cała pula!");
  winnings = winnings + 80;
case 12:
  winnings = winnings + 20;
  break;
default:
  winnings = 0;
}
```

Jeśli wartość zmiennej train == 37, to do zmiennej winnings dodaj 50 i przeskocz na koniec instrukcji.

Jeśli wartość zmiennej train == 65, to do zmiennej winnings dodaj 80, A NASTĘPNIE dodaj do tej samej zmiennej 20 i przeskocz na koniec instrukcji.

Jeśli wartość zmiennej train == 12, to do zmiennej winnings dodaj 20 i przeskocz na koniec instrukcji.

Jeśli zmienna train ma jakąkolwiek inną wartość, to przypisz zmiennej winnings wartość ZERO.



Kiedy komputer dotrze do instrukcji `switch`, sprawdza wartość zmiennej, jaka została w niej podana, a następnie próbuje odnaleźć pasującą do tej wartości klauzulę `case`. Jeśli uda się ją znaleźć, wykonuje *cały* kod aż do napotkania pierwszej instrukcji `break`. **Komputer będzie wykonywał kod aż do momentu, gdy każemy mu przerwać wykonywanie instrukcji `switch`.**

Obejrzyj to!

Brakujące instrukcje break mogą być przyczyną błędnego działania kodu.

W większości programów w języku C każda klauzula `case` *kończy się instrukcją* `break`. *Ułatwia to zrozumienie kodu, choć może nieco pogarszać efektywność jego działania.*

Zaostrz ołówek

Przyjrzyjmy się jeszcze raz poniższemu fragmentowi naszego programu `cards`:

```
int val = 0;
if (card_name[0] == 'K') {
    val = 10;
} else if (card_name[0] == 'Q') {
    val = 10;
} else if (card_name[0] == 'J') {
    val = 10;
} else  if  (card_name[0] == 'A') {
    val = 11;
} else {
    val = atoi(card_name);
}
```

Czy sądzisz, że byłbyś w stanie przepisać ten kod, używając instrukcji `switch`? Odpowiedź zapisz poniżej:

....

....

....

....

....

....

....

....

....

....

....

....

....

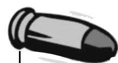
....

Zaostrz ołówek Rozwiązanie

Twoim zadaniem było przepisanie kodu z użyciem instrukcji `switch`.

```
int val = 0;
if (card_name[0] == 'K') {
    val = 10;
} else if (card_name[0] == 'Q') {
    val = 10;
} else if (card_name[0] == 'J') {
    val = 10;
} else  if  (card_name[0] == 'A') {
    val = 11;
} else {
    val = atoi(card_name);
}
```

```
int val = 0;
switch(card_name[0]) {
case 'K':
case 'Q':
case 'J':
    val = 10;
    break;
case 'A':
    val = 11;
    break;
default:
    val = atoi(card_name);
}
```



CELNE SPOSTRZEŻENIA

- Instrukcją `switch` można zastąpić sekwencję instrukcji `if`.
- Instrukcja `switch` sprawdza pojedynczą wartość.
- Komputer rozpocznie wykonywanie kodu od pierwszej pasującej klauzuli `case`.
- Kod będzie wykonywany do pierwszej napotkanej instrukcji `break` bądź aż do końca instrukcji `switch`.
- Upewnij się, że umieściłeś instrukcje `break` we wszystkich właściwych miejscach, w przeciwnym razie instrukcje `switch` mogą działać nieprawidłowo.

Nie istnieją głupie pytania

P: Dlaczego miałbym używać instrukcji `switch` zamiast `if`?

O: Na użycie instrukcji `switch` możesz się zdecydować, jeśli musisz wykonać wiele testów operujących na tej samej zmiennej.

P: Jakie są zalety korzystania z instrukcji `switch`?

O: Jest ich kilka. Przede wszystkim: przejrzystość. Dzięki niej od razu wiadomo, że cały blok kodu operuje na jednej zmiennej.
W przypadku sekwencji instrukcji `if` wcale nie jest to aż tak oczywiste. Poza tym można skorzystać z przechodzenia kolejnych sekcji `case`, by wielokrotnie używać tego samego kodu.

P: Czy w instrukcjach `switch` mogą być używane wyłącznie zmienne? Nie można w nich używać wartości?

O: Owszem, można. Instrukcja `switch` może sprawdzać, czy dwie wartości są sobie równe.

P: Czy w instrukcjach `switch` można sprawdzać łańcuchy znaków?

O: Nie, nie można używać instrukcji `switch` do porównywania ani łańcuchów znaków, ani jakichkolwiek tablic. Instrukcje `switch` mogą operować wyłącznie na pojedynczych wartościach.

Czasami jeden raz nie wystarcza...

Nauczyłeś się już całkiem sporo o języku C, jednak wciąż pozostaje jeszcze wiele ważnych rzeczy, o których musisz się dowiedzieć. Zobaczyłeś już, jak pisać programy określające sposób działania w zależności od wielu różnych sytuacji, istnieje jednak pewne kluczowe zagadnienie, którym się jeszcze nie zajmowaliśmy. Co zrobić, kiedy chcemy, by nasz program wykonywał jakąś czynność *wiele, wiele, wiele razy*?

Stosowanie pętli w języku C

Pętle są szczególnym rodzajem instrukcji sterujących. Instrukcje sterujące określają, *czy* dany blok kodu zostanie wykonany, czy nie. Natomiast pętle określają, *ile razy* dany blok kodu zostanie wykonany.

Najprostszym rodzajem pętli dostępnym w języku C jest pętla `while`. Wykonuje ona umieszczony wewnątrz niej blok kodu *dopóty*, dopóki określony w niej warunek logiczny jest spełniony.

Ten warunek jest sprawdzany przed wykonaniem zawartości pętli.

```
while (<jakiś_warunek>) {
  ... /* Tu wykonujemy jakieś operacje. */
}
```

Zawartość pętli jest umieszczana pomiędzy nawiasami klamrowymi.

Jeśli zawartość pętli składa się z jednego wiersza kodu, nawiasy klamrowe nie są potrzebne.

Po wykonaniu całego bloku kodu umieszczonego wewnątrz pętli komputer sprawdza, czy warunek podany na jej początku wciąż jest spełniony, a jeśli jest, to zawartość pętli jest wykonywana jeszcze raz.

```
while (more_balls)
  keep_juggling();
```

Przynajmniej jeden do while

Istnieje także inna postać pętli `while`, w której warunek jest sprawdzany *po* wykonaniu kodu umieszczonego wewnątrz pętli. Oznacza to, że taka pętla zawsze zostanie wykonana **przynamniej jeden raz**. **Pętlą** tą jest `do … while`:

```
do {
  /* Kupujemy kupon na loterię */
while (have_not_won);
```

Pętle często mają taką samą strukturę...

Pętli while możemy używać zawsze wtedy, gdy konieczne jest wielokrotne wykonanie jakiegoś fragmentu kodu. Jednak w bardzo wielu przypadkach nasze pętle będą miały taką samą strukturę:

- Wykonać jakąś prostą operację przed rozpoczęciem pętli, na przykład ustawić wartość licznika.
- Wykonać prosty test na początku pętli.
- Wykonać jakąś operację na końcu pętli, na przykład zmienić wartość licznika.

Na przykład poniższa pętla while liczy od 1 do 10:

```
int counter = 1;
while (counter < 11) {
  printf("%i zielonych butelek stoi pod ścianą\n", counter);
  counter++;
}
```

To jest kod początkowy pętli.

To jest warunek pętli.

To jest kod aktualizujący, który zmienia wartość licznika i jest wykonywany na samym końcu pętli.

Pamiętaj: counter++ oznacza „powiększ wartość zmiennej counter o jeden".

Takie pętle składają się z kodu, który przygotowuje zmienne używane w danej pętli, jakiejś logiki warunkowej sprawdzanej przed każdym wykonaniem pętli oraz jakiegoś kodu umieszczonego na końcu pętli, który aktualizuje licznik lub wykonuje jakąś inną, podobną operację.

...a dzięki instrukcji for tworzenie takich pętli jest łatwe

Ponieważ pętle tworzone według takiego wzorca występują bardzo często, projektanci języka C stworzyli instrukcję **for**, dzięki której kod takich pętli może być nieco krótszy i bardziej zwarty. Oto kod realizujący dokładnie to samo zadanie, ale napisany z użyciem pętli for:

```
int counter;
for (counter = 1; counter < 11; counter++) {
  printf("%i zielonych butelek stoi pod ścianą\n", counter);
}
```

Ten fragment inicjalizuje zmienną używaną w pętli.

To jest warunek sprawdzany **przed** każdym wykonaniem pętli.

To kod, który zostanie zrealizowany **po** każdym wykonaniu zawartości pętli.

Ponieważ nasza pętla zawiera tylko jeden wiersz kodu, moglibyśmy pominąć nawiasy klamrowe.

Pętle for są bardzo *często* używane w programach pisanych w języku C — równie często, a może nawet częściej niż pętle while. Nie tylko umożliwiają one nieznaczne skrócenie kodu, lecz także ułatwiają jego zrozumienie, gdyż cały kod używany do sterowania działaniem pętli — czyli związany ze sprawdzaniem i aktualizacją zmiennej counter — jest umieszczony w samej instrukcji for, a nie rozsiany w kodzie umieszczonym wewnątrz niej.

Każda pętla for musi mieć jakąś zawartość.

Instrukcji break używamy, by wydostać się z pętli...

W języku C możemy tworzyć pętle sprawdzające warunek logiczny na samym jej początku lub końcu. Ale co zrobić, jeśli będziemy chcieli przerwać działanie pętli gdzieś w środku kodu umieszczonego wewnątrz pętli? Zawsze można odpowiednio zmienić strukturę kodu, jednak czasami łatwiej jest wyjść z pętli natychmiast, w dowolnym jej miejscu, używając do tego instrukcji **break**:

```
while(feeling_hungry) {
    eat_cake();
    if (feeling_queasy) {
        /* Wychodzimy z pętli while */
        break;
    }
    drink_coffee();
}
```

Instrukcja „break" powoduje natychmiastowe wyjście z pętli.

Obejrzyj to! **Instrukcja break jest używana do przerywania działania pętli oraz instrukcji switch.**

Używając instrukcji `break`*, upewnij się, że wiesz, z której instrukcji chcesz wyjść.*

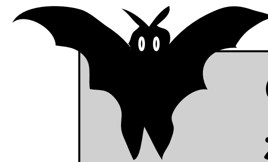
Użycie instrukcji `break` spowoduje natychmiastowe wyjście z aktualnie wykonywanej pętli i pominięcie całej reszty umieszczonego w niej kodu. Stosowanie tych instrukcji jest bardzo przydatne, gdyż czasami stanowią one najszybszy i najprostszy sposób zakończenia działania pętli. Warto jednak unikać stosowania zbyt wielu instrukcji `break`, gdyż mogą one utrudniać zrozumienie działania kodu.

...a instrukcji continue, by ją kontynuować

Jeśli chcemy pominąć dalszą część kodu umieszczonego wewnątrz pętli i wrócić do jej początku, możemy skorzystać z pomocy instrukcji `continue`:

```
while(feeling_hungry) {
    if (not_lunch_yet) {
        /* Wracamy na początek pętli, do warunku */
        continue;
    }
    eat_cake();
}
```

Instrukcja continue przenosi nas z powrotem na początek pętli.



Opowieści z krypty

Break nie przerywa instrukcji if.

15 stycznia 1990 roku uległ awarii system rozmów długodystansowych firmy AT&T, przez co 60 tysięcy osób nie było w stanie korzystać z usług telefonicznych. A co było przyczyną tej awarii? Otóż programista, który pisał w języku C kod odpowiadający za wymianę połączeń, spróbował użyć instrukcji `break`*, by zakończyć wykonywanie instrukcji* `if`*. Jednak instrukcje* `break` *nie działają w instrukcjach* `if`*. W efekcie program pomijał całą sekcję kodu, co było przyczyną błędu, który doprowadził do przerwania 70 milionów rozmów telefonicznych w ciągu dziewięciu godzin.*

Pisanie funkcji pod lupą

Zanim wypróbujemy magiczne moce nowo poznanych pętli, udajmy się na krótki objazd, by przyjrzeć się funkcjom.

Do tej pory w każdym tworzonym programie musiałeś zdefiniować jedną funkcję — `main()`:

```
int main()
{
    puts("Zbyt młodzi na sen; zbyt piękni, by żyć");
    return 0;
}
```

Ta funkcja zwraca wartość typu *int*. (→ int)

To jest nazwa funkcji. (→ main)

Wewnątrz tych nawiasów nic nie ma. (→ ())

To jest zawartość funkcji — to właśnie ona robi to, co trzeba. (→ puts)

Zawartość funkcji jest zapisana pomiędzy nawiasami klamrowymi.

Po zakończeniu funkcja zwraca wartość. (→ return 0;)

Niemal wszystkie funkcje w języku C wyglądają podobnie. Na przykład poniższy program zawiera napisaną przez nas niestandardową funkcję, która jest wywoływana przez funkcję `main()`:

```
#include <stdio.h>

int larger(int a, int b)
{
  if (a > b)
    return a;
  return b;
}

int main()
{
  int greatest = larger(100, 1000);
  printf("%i jest większe!\n", greatest);
  return 0;
}
```

Zwracana jest wartość typu int. (→ int larger)

Ta funkcja przyjmuje dwa **argumenty**: a oraz b. Oba są wartościami typu int.

W tym miejscu wywołujemy funkcję. (→ larger(100, 1000))

Funkcja `larger()` nieznacznie różni się od funkcji `main()`, gdyż pobiera ***argumenty***, nazywane także ***parametrami***. **Argument** jest zmienną lokalną, której wartość jest przekazywana przez kod wywołujący funkcję. Funkcja `larger()` pobiera dwa argumenty — `a` oraz `b` — i zwraca większą z przekazanych liczb.

Uprzejmy przewodnik po standardach

Ponieważ w definicji funkcji `main()` został podany typ wyniku `int`, zatem na jej końcu powinna się znaleźć instrukcja `return`. Jeśli jednak ją pominiemy, to kod i tak będzie można skompilować — choć kompilator może wygenerować ostrzeżenie. W przypadku stosowania kompilatora **C99** w razie pominięcia instrukcji `return` zostanie ona automatycznie dodana. Aby kompilator działał w standardzie C99, w wierszu jego wywołania należy podać parametr `-std=99`.

Funkcje void pod lupą

Większość funkcji pisanych w języku C zwraca jakąś wartość. Jednak od czasu do czasu zdarzają się funkcje niedysponujące żadnymi użytecznymi informacjami, które mogłyby zwrócić. Takie funkcje zazwyczaj coś *robią*, a nie *wyliczają*. Funkcje zawsze powinny zawierać instrukcję `return`, wyjątkiem są te spośród nich, w których typem wyniku jest **`void`**.

Typ void oznacza, że funkcja nic nie zwraca.

```
void complain()
{
  puts("Jestem naprawdę nieszczęśliwy!");
}
```

Instrukcja return nie jest potrzebna, gdyż funkcja nic nie zwraca.

W języku C słowo kluczowe `void` oznacza, że coś *nie ma znaczenia*. Jeśli tylko poinformujemy kompilator, że nie interesuje nas zwracanie wyniku z funkcji, to nie będziemy musieli umieszczać w niej instrukcji `return`.

Nie istnieją głupie pytania

P: Czy zastosowanie typu `void` w definicji funkcji oznacza, że wewnątrz niej nie można używać instrukcji `return`?

O: W takiej funkcji wciąż można używać instrukcji `return`, jednak może to spowodować wygenerowanie ostrzeżenia przez kompilator. Poza tym umieszczanie instrukcji `return` w takiej funkcji jest bezcelowe.

P: Naprawdę? Dlaczego?

O: Gdyż próba pobrania wartości z funkcji typu `void` spowoduje zgłoszenie błędu przez kompilator.

Łańcuchy przypisań

Niemal wszystko w języku C zwraca jakąś wartość; nie dotyczy to wyłącznie funkcji. W rzeczywistości nawet przypisania mają swoją wartość wynikową. Przyjrzyjmy się na przykład poniższej instrukcji:

```
x = 4;
```

Instrukcja ta przypisuje zmiennej liczbę 4. Interesujące jest to, że *samo* wyrażenie `x = 4` ma wartość równą wartości przypisywanej, czyli 4. A dlaczego to ma znaczenie? Ponieważ oznacza to, że możemy stosować ciekawe sztuczki, takie jak tworzenie łańcuchów przypisań:

Przypisanie „x = 4” ma wartość 4.

A zatem także zmiennej y zostanie przypisana wartość 4.

```
y = (x = 4);
```

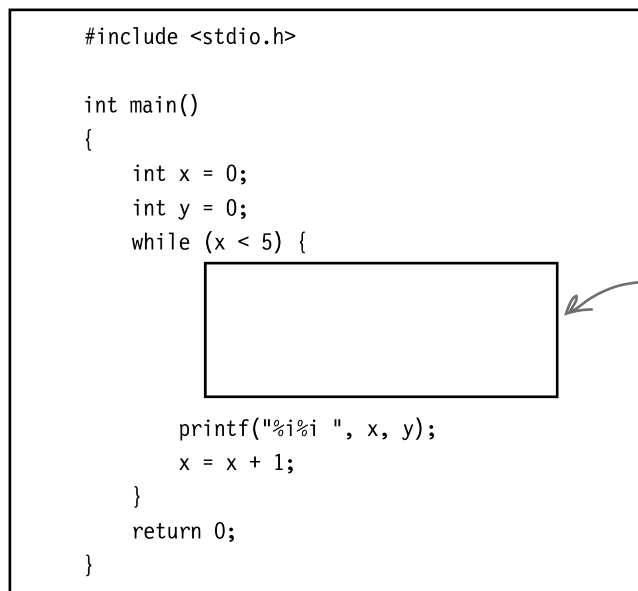
Powyższy wiersz kodu przypisze wartość 4 zarówno zmiennej `x`, **jak i** `y`. Okazuje się, że możemy nieco skrócić ten kod, usuwając z niego nawiasy:

```
y = x = 4;
```

Takie łańcuchowe przypisania można bardzo często zobaczyć w kodzie, w którym kilku zmiennym trzeba przypisać tę samą wartość.

Pomieszane wiadomości

Poniżej został przedstawiony kod prostego programu napisanego w C. Brakuje w nim jednego fragmentu. Twoim zadaniem jest dopasowanie **proponowanych bloków kodu** (widocznych poniżej z lewej strony) do **wyników**, które zostałyby wygenerowane przez program w razie umieszczenia w nim danego kodu. Nie wszystkie wiersze wyników zostaną użyte, a niektóre z nich mogą zostać wykorzystane więcej niż jeden raz. Narysuj linie łączące proponowane bloki kodu z generowanymi przez nie wynikami.



```
#include <stdio.h>

int main()
{
    int x = 0;
    int y = 0;
    while (x < 5) {
        [          ]

        printf("%i%i ", x, y);
        x = x + 1;
    }
    return 0;
}
```


Dopasuj proponowane bloki kodu do generowanych przez nie wyników.

Proponowane bloki kodu

```
y = x - y;
```

```
y = y + x;
```

```
y = y + 2;
if (y > 4)
    y = y - 1;
```

```
x = x + 1;
y = y + x;
```

```
if (y < 5) {
    x = x + 1;
    if (y < 3)
        x = x - 1;
}
y = y + 2;
```

Możliwe wyniki

```
22 46
```

```
11 34 59
```

```
02 14 26 38
```

```
02 14 36 48
```

```
00 11 21 32 42
```

```
11 21 32 42 53
```

```
00 11 23 36 410
```

```
02 14 25 36 47
```

Ćwiczenie

Skoro już wiesz, jak tworzyć pętle `while`, spróbuj zmodyfikować program, by na bieżąco zliczał wyniki gry. Wyświetlaj liczbę po każdej karcie i zakończ działanie programu, gdy użytkownik naciśnie klawisz *X*. Wyświetlaj komunikat o błędzie, gdy użytkownik wpisze niewłaściwą wartość karty, taką jak 11 lub 24.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char card_name[3];
    int count = 0;
    while (............................................) {
        puts("Wpisz symbol karty (card_name): ");
        scanf("%2s", card_name);
        int val = 0;
        switch(card_name[0]) {
        case 'K':
        case 'Q':
        case 'J':
            val = 10;
            break;
        case 'A':
            val = 11;
            break;
        case 'X':
            ...............................................
        default:
            val = atoi(card_name);
            ...............................................
            ...............................................
            ...............................................
            ...............................................
            ...............................................
        }
        if ((val > 2) && (val < 7)) {
            count++;
        } else if (val == 10) {
            count--;
        }
        printf("Bieżąca liczba: %i\n", count);
    }
    return 0;
}
```

Masz przerwać program, gdy użytkownik naciśnie klawisz X. (→ warunek pętli while)

Co tu zrobisz? (→ case 'X')

Masz wyświetlić komunikat o błędzie, jeśli wartość karty nie będzie należeć do zakresu od 1 do 10. W takim przypadku powinieneś także pominąć resztę kodu w pętli i rozpocząć ją od nowa.

Do wartości count dodajemy 1. (→ count++)

Od wartości count odejmujemy 1. (→ count--)

Pomieszane wiadomości. Rozwiązanie

Poniżej został przedstawiony kod prostego programu napisanego w C. Brakuje w nim jednego fragmentu. Twoim zadaniem było dopasowanie **proponowanych bloków kodu** (widocznych poniżej z lewej strony) do **wyników**, które zostałyby wygenerowane przez program w razie umieszczenia w nim danego kodu. Nie wszystkie wiersze wyników zostały użyte. Miałeś narysować linie łączące proponowane bloki kodu z generowanymi przez nie wynikami.

```
#include <stdio.h>

int main()
{
    int x = 0;
    int y = 0;
    while (x < 5) {
        [                    ]

        printf("%i%i ", x, y);
        x = x + 1;
    }
    return 0;
}
```

Tutaj umieść proponowany blok kodu.

Proponowane bloki kodu

```
y = x - y;
```

```
y = y + x;
```

```
y = y + 2;
if (y > 4)
    y = y - 1;
```

```
x = x + 1;
y = y + x;
```

```
if (y < 5) {
    x = x + 1;
    if (y < 3)
        x = x - 1;
}
y = y + 2;
```

Możliwe wyniki

```
22 46
11 34 59
02 14 26 38
02 14 36 48
00 11 21 32 42
11 21 32 42 53
00 11 23 36 410
02 14 25 36 47
```

Rozwiązanie ćwiczenia

Skoro już wiesz, jak tworzyć pętle `while`, spróbuj zmodyfikować program, by na bieżąco zliczał wyniki gry. Wyświetlaj liczbę po każdej karcie i zakończ działanie programu, gdy użytkownik naciśnie klawisz *X*. Wyświetlaj komunikat o błędzie, gdy użytkownik wpisze niewłaściwą wartość karty, taką jak 11 lub 24.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char card_name[3];
    int count = 0;
    while (card_name[0] != 'X') {
        puts("Wpisz symbol karty (card_name): ");
        scanf("%2s", card_name);
        int val = 0;
        switch(card_name[0]) {
        case 'K':
        case 'Q':
        case 'J':
            val = 10;
            break;
        case 'A':
            val = 11;
            break;
        case 'X':
            continue
        default:
            val = atoi(card_name);
            if ((val < 1) || (val > 10)) {
              puts("Nie rozumiem tej wartości!");
              continue;
            }
        }
        if ((val > 2) && (val < 7)) {
            count++;
        } else if (val == 10) {
            count--;
        }
        printf("Bieżąca liczba: %i\n", count);
    }
    return 0;
}
```

Masz przerwać program, gdy użytkownik naciśnie klawisz X.

*Instrukcja **break** nie spowodowałaby przerwania pętli, gdyż znajdujemy się aktualnie wewnątrz instrukcji switch. Dlatego, aby wrócić na sam początek pętli i sprawdzić jej warunek, musimy skorzystać z instrukcji **continue**.*

To jeden z możliwych sposobów zapisania tego warunku.

*W tym miejscu musimy ponownie skorzystać z instrukcji **continue**, gdyż chcemy kontynuować działanie pętli.*

Jazda próbna

Teraz nasz program liczący karty jest już skończony i nadszedł czas, by go wypróbować. Jak sądzisz, będzie działać?

Pamiętaj: jeśli uruchamiasz program w systemie Windows, to nie musisz używać znaków ./ przed jego nazwą.

To polecenie skompiluje i uruchomi program.

Teraz sprawdzamy, czy jest to prawidłowa wartość karty.

Liczba rośnie!

```
Plik  Edycja  Okno  Pomoc  ZapętliłemSię
> gcc card_counter.c -o card_counter && ./card_counter
Wpisz symbol karty (card_name):
4
Bieżąca liczba: 1
Wpisz symbol karty (card_name):
K
Bieżąca liczba: 0
Wpisz symbol karty (card_name):
3
Bieżąca liczba: 1
Wpisz symbol karty (card_name):
5
Bieżąca liczba: 2
Wpisz symbol karty (card_name):
23
Nie rozumiem tej wartości!
Wpisz symbol karty (card_name):
6
Bieżąca liczba: 3
Wpisz symbol karty (card_name):
5
Bieżąca liczba: 4
Wpisz symbol karty (card_name):
3
Bieżąca liczba: 5
Wpisz symbol karty (card_name):
X
```

Dorobiłem się fortuny, obstawiając duże zakłady, gdy liczba była wysoka.

Program zliczający karty działa!

W ten sposób napisałeś swój pierwszy program w języku C. Korzystając z możliwości, jakie dają instrukcje, pętle oraz warunki, stworzyłeś w pełni funkcjonalny program do liczenia kart.

Świetna robota!

Uwaga prawna: w wielu krajach i stanach korzystanie z komputerów do liczenia kart jest zabronione, a właściciele kasyn mogą się nieco zdenerwować, gdy spróbujesz tej metody. Zatem lepiej tego nie rób, dobra?

Nie istnieją głupie pytania

P: Dlaczego programy pisane w C trzeba kompilować? Inne języki, takie jak JavaScript, nie wymagają kompilacji, prawda?

O: Język C jest kompilowany, by pisane w nim programy były szybkie. Choć są języki niewymagające kompilacji, takie jak JavaScript oraz Python, to jednak niektóre z nich, w niezauważalny sposób, kompilują kod, by przyspieszyć jego działanie.

P: Czy C++ to inna wersja języka C?

O: Nie. Język C++ został początkowo opracowany jako rozszerzenie C, jednak aktualnie jest już czymś więcej. C++ oraz Objective-C zostały stworzone, by zapewnić możliwość korzystania z obiektowości w języku C.

P: Czym jest obiektowość? Czy w tej książce dowiem się czegoś o niej?

O: Obiektowość to technika radzenia sobie ze złożonością. W tej książce nie będziemy o niej pisali.

P: Kod C wygląda podobnie do kodu pisanego w językach JavaScript, Java, C# itd.

O: Język C ma bardzo zwartą składnię i wywarł wpływ na wiele innych języków programowania.

P: Co oznacza `gcc`?

O: To skrót od „Gnu Compiler Collection”.

P: A skąd słowo „collection”. Czy jest tam więcej kompilatorów?

O: Gnu Compiler Collection można używać do kompilowania kodu pisanego w kilku różnych językach, choć najprawdopodobniej wciąż najczęściej używany jest język C.

P: Czy można stworzyć pętlę działającą w nieskończoność?

O: Tak. Jeśli w warunku pętli umieścimy wartość 1, to taka pętla nigdy się nie skończy.

P: Czy tworzenie takich nieskończonych pętli jest dobrym pomysłem?

O: Czasami. Pętle nieskończone (działające w nieskończoność) są często stosowane w takich programach jak serwery sieciowe, które wykonują jakąś czynność nieprzerwanie aż do momentu, gdy zostaną zakończone. Jednak większość programistów tworzy swoje pętle w taki sposób, by kiedyś się zakończyły.

CELNE SPOSTRZEŻENIA

- Pętla `while` wykonuje umieszczony wewnątrz blok kodu dopóty, dopóki spełniony jest jej warunek.
- Pętla `do-while` jest podobna, lecz umieszczony wewnątrz niej kod będzie wykonany przynajmniej raz.
- Pętla `for` jest zwartym sposobem tworzenia pewnego rodzaju pętli.
- Wykonywanie pętli można w każdej chwili przerwać, używając instrukcji `break`.
- Instrukcja `continue` pozwala w dowolnej chwili przejść na początek pętli, do jej warunku.
- Instrukcja `return` zwraca wartość z funkcji.
- W funkcjach zdefiniowanych z użyciem typu `void` nie trzeba używać instrukcji `return`.
- Większość wyrażeń w języku C ma wartość.
- Instrukcje przypisania mają wartość, a zatem można je ze sobą łączyć (`x = y = 0`).

Twój niezbędnik C

Masz już za sobą rozdział 1., a do swojego niezbędnika dodałeś podstawowe wiadomości dotyczące języka C. Pełną listę porad i wskazówek znajdziesz w dodatku B.

Instrukcja switch w wydajny sposób sprawdza różne wartości jednej zmiennej.

Proste instrukcje można porównać do poleceń.

Każdy program musi zawierać funkcję main().

Instrukcje blokowe są umieszczane pomiędzy nawiasami klamrowymi — { oraz }.

Przed uruchomieniem programu napisanego w C trzeba go skompilować.

W wierszu poleceń możesz umieścić &&, by wykonać program wyłącznie wtedy, gdy uda się go skompilować.

Dyrektywa #include dołącza zewnętrzny kod wykonujący różne operacje, na przykład operacje wejścia-wyjścia.

Instrukcje if wykonują kod, jeśli określony warunek jest spełniony.

Parametr -o określa plik wynikowy.

Do łączenia warunków można używać operatorów && oraz ||.

Pętla while wykonuje kod dopóty, dopóki jest spełniony określony warunek.

gcc jest najpopularniejszym kompilatorem języka C.

Pliki źródłowe powinny mieć rozszerzenie .c.

Pętle do-while wykonują kod przynajmniej raz.

count++ oznacza, że do wartości count należy dodać 1.

count-- oznacza, że od wartości count należy odjąć 1.

Pętle for są bardziej zwartym sposobem tworzenia pętli.

Skorowidz

- !, operator, 18
- %p, łańcuch formatujący, 52
- &&, operator, 18
- &, operator, 20, 43, 48, 51, 539
- *, operator, 48, 51
- --, operator, 13
- ., operator, 220
- ^, operator, 539
- |, operator, 20, 539
- ||, operator, 18
- ~, operator, 539
- ++, operator, 13
- +=, operator, 13
- <, operator, 109
- <<, operator, 539
- =, operator, 13
- -=, operator, 13
- ==, operator, 13
- >, operator, 110
- ->, operator, 560
- >>, operator, 539

A

- abstrakcyjne struktury danych, 267
- accept(), 469
- AceUnit, 543
- alarm(), 457, 463, 569
- Allegro, 524, 525, 529, 533
  - instalacja, 524
- analogRead(), 212
- analogWrite(), 212
- ANSI C, 2, 19
- ant, program, 200
- ar, format, 361
- ar, polecenie, 357, 361, 566
- archiwum, 355, 356, 357, 361
- Arduino, 207
- Arduino IDE, 207, 211
  - analogRead(), 212
  - analogWrite(), 212
  - delay(), 212
  - digitalRead(), 212
  - digitalWrite(), 212
  - funkcje, 212
  - pinMode(), 212
  - Serial.begin(), 212
  - Serial.println(), 212
- argumenty
  - przekazywanie przez wartość, 46
  - wiersza poleceń, 139, 557
- arytmetyka wskaźników, 61, 62, 64
- autoconf, program, 200

B

- biblioteki
  - Cygwin, 375, 384
  - dynamiczne, 371, 373, 381, 383, 384, 566, 567
  - katalog, 357, 358
  - Linux, 374
  - Mac, 374
  - MinGW, 375, 384
  - nazwy, 361, 373
  - statyczne, 357, 566
  - Windows, 375
- bind(), 475, 570
- bity, 539
- blokowe, instrukcje, 14
- błąd magistrali, 13
- break, 28, 31, 39

C

- C Standard Library, 86
- C, język, 2, 9
- C++, język, 39
- C99, 2, 8, 19, 32
- Carbon, 549
- char, 160
- char**, 318, 331
- CMake, 524
- const, 79
- const char, 218
- continue, 31, 39
- convert, 447
- CreateProcess(), 424
- curl, 447
- cvCalcOpticalFlowFarneback(), 391
- cvCreateCameraCapture(), 390
- cvQueryFrame(), 390
- Cygwin
  - biblioteki, 375, 384
  - fork(), 424
  - telnet, 466
- czujnik wilgotności
  - budowa, 208, 209
  - działanie, 210

D

- deklaracja, 79
- dekrementacja, 538
- delay(), 212
- deskryptor pliku, 429, 430
- digitalRead(), 212
- digitalWrite(), 212
- DNS, 491, 570
- do ... while, pętla, 29, 553
- docelowy, plik, 197
- double, 160, 558
- drzewo binarne, 294, 298
- dup2(), 431, 438, 569
- dyrektywy preprocesora, 540

E

- enum, 253, 258
- exec(), 402, 403, 404, 409, 419, 424, 568
- execl(), 403, 568

execle(), 403, 568
execlp(), 403, 568
execv(), 404, 568
execve(), 404, 568
execvp(), 404, 568
exit(), 438, 439, 568

F

fgets(), 67, 68, 554
FILE, typ, 137
fileno(), 431, 438, 569
filtry, 107
 head, 107
 sed, 107
 tail, 107
find(), 313
float, 160, 558
fopen(), 136
for, pętla, 30, 39, 553
fork(), 418, 419, 421, 424, 568, 570
 Cygwin, 424
 pid_t, 421, 424
 Windows, 424
fprintf(), 120, 556
free(), 277, 278, 563
fscanf(), 136
funkcje, 32, 33, 564
 accept(), 469
 alarm(), 457, 463, 569
 bind(), 475, 570
 CreateProcess(), 424
 definiowanie, 171
 deklaracja, 171
 dup2(), 431, 438, 569
 exec(), 402, 403, 404, 409, 419, 424, 568
 execl(), 403, 568
 execle(), 403, 568
 execlp(), 403, 568
 execv(), 404, 568
 execve(), 404, 568
 execvp(), 404, 568
 exit(), 438, 439, 568
 fgets(), 67, 68, 554
 fileno(), 431, 438, 569
 find(), 313
 fopen(), 136
 fork(), 418, 419, 421, 424, 568, 570
 fprintf(), 120, 556
 free(), 277, 278, 563
 fscanf(), 136
 getaddrinfo(), 490, 491, 497, 570
 getenv(), 405
 getopt(), 147, 153, 557
 gets(), 67
 listen(), 469, 570
 main(), 5, 6, 139, 153
 malloc(), 276, 278, 563
 mkfifo(), 448
 nazwa, 316, 564
 o zmiennej liczbie argumentów, 347
 pipe(), 442, 448, 568
 printf(), 6, 108, 122, 556
 przekazywanie łańcucha znaków, 53
 pthread_create(), 505
 pthread_join(), 505
 pthread_mutex_lock(), 571
 pthread_mutex_unlock(), 571
 qsort(), 324, 325, 331, 340, 564
 raise(), 455, 463, 569
 recv(), 476, 497
 scanf(), 65, 66, 68, 79, 108, 122, 556
 send(), 470, 497
 settimer(), 457
 sigaction(), 450, 451, 463, 569
 sleep(), 456, 463, 506
 socket(), 570
 strcat(), 88
 strchr(), 88, 555
 strcmp(), 88, 331, 555
 strcpy(), 88
 strdup(), 283, 292, 563
 strerror(), 406
 strlen(), 88, 555
 strstr(), 83, 88, 89, 555
 system(), 396, 399, 400, 401, 409, 424, 568
 waitpid(), 436, 437, 438, 439, 568
 wskaźniki, 318, 322

G

gcc, 9, 22, 39, 544, 553
 -c, 189
 -fPIC, 372, 383
 -g, 300
 -I, 354
 kompilacja, 9, 10
 -L, 361
 -O, 544
 -O2, 544
 -O3, 544
 -Ofast, 544
 opis flag, 544
 optymalizacje, 544
 -shared, 373, 566
 -Wall, 545
 -Werror, 545
 -Wextra, 545
 wyświetlanie ostrzeżeń, 545
gcov, 548
gdb, 548
getaddrinfo(), 490, 491, 497, 570
getenv(), 405
getopt(), 147, 153, 557
gets(), 67
globalny segment danych, 43
gniazda, 470, 497
 akceptacja połączenia, 469
 bind(), 475, 570
 getaddrinfo(), 491, 497, 570
 klient, 490, 491
 listen(), 469, 570
 nasłuchiwanie, 469
 powiązanie z portem, 468
 recv(), 476, 497
 send(), 470, 497
 serwer, 468, 469, 470, 476
GNU Compiler Collection, *Patrz* gcc
gprof, 548

graficzny interfejs użytkownika, 549
 Linux, 549
 Mac, 549
 Windows, 549
GTK, 549

H

head, 107
HTTP, 467

I

if, instrukcja, 14, 28, 552
include, 5, 179, 552
inkrementacja, 538
instrukcje, 14
 blokowe, 14
 sterujące, 14
int, 160, 558
IP, 467, 489

J

jądro systemu, 401
język C, 2, 9
język C++, 39
język kompilowany, 9
język Objective-C, 39

K

kanały RSS, 414
kill, polecenie, 455, 463, 569
klient, 488, 489, 497
kod maszynowy, 2
kod przesuwny, 372
kod wynikowy, 2
kod źródłowy, 2
komentarze, 5, 8
kompilacja, 2, 39
 do plików obiektowych, 189
 etapy, 182, 183
 -lfred, 361
 warunkowa, 540
kompilator, 2, 8, 9
kompilowany, język, 9
konsolidacja, 183

L

LD_LIBRARY_PATH, 374, 383, 384
liczniki czasu, 456, 457, 463
limits.h, 542
Linux
 biblioteki, 374
 graficzny interfejs użytkownika, 549
 GTK, 549
 telnet, 466
listen(), 469, 570
listy dwukierunkowe, 294
listy połączone, 267, 268, 272, 294, 562
literały dwójkowe, 263
literały łańcuchowe, 13, 72, 73, 555
literały szesnastkowe, 259
long, 160, 162, 558

Ł

łańcuchy przypisań, 33
łańcuchy znaków, 11, 12, 13, 73, 555
 długość, 88
 kopia, 74
 kopiowanie, 88
 łączenie, 88
 plik nagłówkowy, 86
 położenie znaku, 88, 89
 porównywanie, 88
 wartownik, 12, 13
łączenie dynamiczne, 370, 382, 383, 384
łączenie statyczne, 382, 383, 384

M

Mac
 biblioteki, 374
 Carbon, 549
 graficzny interfejs użytkownika, 549
 telnet, 466
mail, 447
main(), 5, 6, 139, 153
make, narzędzie, 196, 197, 200, 202, 203, 546
 reguła, 196
 reguły niejawne, 547
makefile, plik, 198, 200, 203, 546
makra, 344
 LONG_MAX, 542
 SHORT_MIN, 542
 va_list, 343
 WEXITSTATUS(), 439
malloc(), 276, 278, 563
mapy, 294
maszynowy, kod, 2
MinGW, biblioteki, 375, 384
mingw32-make, 197
mkfifo(), 448
muteksy, 511, 512, 518, 571
mutt, 447

N

nagłówkowy, plik, 86, 94, 96, 171
 cudzysłów, 352
 nawiasy kątowe, 352, 361
 tworzenie, 172
 współużytkowanie, 354
nm, polecenie, 358
NMAKE, 197
NULL, 270

O

obiektowość, 39
Objective-C, język, 39
odwzorowanie pamięci, 372
opcje wiersza poleceń, 146, 147, 153
OpenCV, 389, 391
 cvCalcOpticalFlowFarneback(), 391
 cvCreateCameraCapture(), 390
 cvQueryFrame(), 390
operacje bitowe, 20, 539
 &, operator, 539
 ^, operator, 539
 |, operator, 539
 <<, operator, 539
 >>, operator, 539
operatory, 538
 !, 18

- operatory
  - &, 20, 43, 48, 51, 539
  - &&, 18
  - *, 48, 51
  - ., 220
  - ^, 539
  - |, 20, 539
  - ||, 18
  - ~, 539
  - ++, 13
  - +=, 13
  - <, 109
  - <<, 539
  - =, 13
  - -=, 13
  - ==, 13
  - >, 110
  - ->, 560
  - >>, 539
  - dekrementacja, 538
  - inkrementacja, 538
  - sizeof, 53, 56, 59, 64, 278
  - trójargumentowe, 538
- optymalizacje, 544

P

- pamięć dynamiczna, 274, 276, 289, 563
  - rezerwacja pamięci, 276
  - zwalnianie, 277
- pamięć globalna, 43
- pętle, 29
  - do ... while, 29, 553
  - for, 30, 39, 553
  - nieskończone, 39
  - while, 29, 30, 39, 553
- pid_t, 421, 424
- pinMode(), 212
- pipe(), 442, 448, 568
- plik docelowy, 197
- plik makefile, 198, 200, 203, 546
- plik nagłówkowy, 86, 94, 96, 171
  - cudzysłów, 352
  - nawiasy kątowe, 352, 361
  - tworzenie, 172
  - współużytkowanie, 354
- plik wynikowy, 196
- plik źródłowy, 5
- PNAR, 468, 570
- pola bitowe, 259, 260, 263, 561
- port, 468, 475
  - którego używać, 470
- potok, 129, 134, 441, 448
- preprocesor, 178, 344
  - dyrektywy, 540
- printf(), 6, 108, 122, 556
- procedura obsługi, 450
- proces
  - fork(), 418, 424
  - potomny, 418
  - rodzicielski, 418
  - tworzenie kopii, 418, 424
- programy narzędziowe, 102, 107, 126, 127, 134
- protokół, 467, 497
- przesuwny, kod, 372
- przypisywanie, 13
- pthread, biblioteka, 504, 571
- pthread_create(), 505
- pthread_join(), 505
- PTHREAD_MUTEX_INITIALIZER, 512
- pthread_mutex_lock(), 571

Q

- qsort(), 324, 325, 331, 340, 564

R

- raise(), 455, 463, 569
- recv(), 476, 497
- referencje, 52
- return, 33, 39
- równość, testowanie, 13
- RSS, 414
- RSS Gossip, 414
- rzutowanie, 162

S

- scanf(), 65, 66, 68, 79, 108, 122, 556
  - przepełnienie bufora, 66
- sed, 107
- send(), 470, 497
- Serial.begin(), 212
- Serial.println(), 212
- serwer, 468, 469, 489, 497
  - kilku klientów, 484
- settimer(), 457
- short, 160, 558
- shunit2, 543
- SIG_DFL, 457
- SIG_IGN, 457
- sigaction(), 450, 451, 463, 569
- SIGALRM, 456, 463, 569
- SIGEGV, 454
- SIGFPE, 454
- SIGINT, 454
- signal.h, 457
- SIGPIPE, 454
- SIGQUIT, 454
- SIGTERM, 454
- SIGTRAP, 454
- SIGWINCH, 454
- sizeof, operator, 53, 56, 59, 64, 278
- sleep(), 456, 463, 506
- słowa kluczowe, 179
- socket(), 570
- sortowanie, 324, 325, 340, 564
- stałe, 80
- standardowa biblioteka C, 86
- standardowy strumień błędów, 118, 119, 122, 556
- standardowy strumień wejściowy, 108, 109, 556
  - przekierowania, 428
- standardowy strumień wyjściowy, 108, 110, 556
  - przekierowania, 430
- static, 541
- stdarg.h, 343
- stderr, 120
- stdin, 120
- stdio.h, 86
- stdout, 120
- sterta, 80, 276, 277, 290, 291, 292, 563
- stos, 43, 80, 276, 290, 291, 563

strcat(), 88
strchr(), 88, 555
strcmp(), 88, 331, 555
strcpy(), 88
strdup(), 283, 292, 563
strerror(), 406
string.h, 83, 86, 555
strlen(), 88, 555
strstr(), 83, 88, 89, 555
struct, 218, 223, 230, 233
strukturalne typy danych, 218
struktury, 218, 219, 223, 230, 233, 258, 560, 562
  aktualizacja zawartości, 234
  anonimowe, 233
  nazwy zastępcze, 230, 233
  odczytywanie pól, 220
  rekurencyjne, 269
  zawierające inne struktury, 225
strumienie danych, 136, 428, 556
  ilość, 137
  tworzenie, 136
  zamykanie, 136
switch, instrukcja, 25, 26, 28, 552
  break, 26, 28
sygnały, 454, 463, 569
  raise(), 455, 463, 569
  SIG_DFL, 457
  SIG_IGN, 457
  sigaction(), 450, 451, 463, 569
  SIGALRM, 456, 463, 569
  SIGFPE, 454
  SIGINT, 454
  SIGPIPE, 454
  SIGQUIT, 454
  SIGSEGV, 454
  SIGTERM, 454
  SIGTRAP, 454
  SIGWINCH, 454
  wysyłanie, 455, 456
system nazw domen, 491
system(), 396, 399, 400, 401, 409, 424, 568
szyfrowanie, 180

T

tablice, 13, 61
  adres, 59
  deskryptorów, 429
  długość, 59
  skojarzeniowe, 294
  tablic, 85, 98
  wskaźników, 98
  wskaźników do funkcji, 336, 339, 340
  znaków, 11, 13
tail, 107
telnet, 466, 570
  Cygwin, 466
  Linux, 466
  Mac, 466
  Windows, 466
testowanie, 543
typedef, 230, 233, 269, 560
typy danych, 158, 161, 166, 558
  char, 157, 160
  double, 157, 160, 558
  float, 157, 160, 558
  int, 157, 160, 558
  long, 157, 160, 558
  short, 157, 160, 558
  strukturalne, 218
  zakres wartości, 542
typy wyliczeniowe, 253, 263, 561

U

unie, 245, 246, 247, 258, 263, 561
union, 245
unistd.h, 147
unsigned, 162

V

valgrind, program, 300, 301, 306
void, 33

W

waitpid(), 436, 437, 438, 439, 568
wartości logiczne, 18
wartownik, 12, 13
wątki, 502, 503, 510, 518, 571
  muteksy, 511, 512, 518, 571
  pthread_create(), 505
  pthread_join(), 505
  PTHREAD_MUTEX_INITIALIZER, 512
  pthread_mutex_lock(), 571
  pthread_mutex_unlock(), 571
  tworzenie, 504, 505
WEXITSTATUS(), 437, 439
wget, 447
while, pętla, 29, 30, 39, 553
Windows
  biblioteki, 375
  fork(), 424
  graficzny interfejs użytkownika, 549
  telnet, 466
wskaźniki, 42, 43, 47, 51, 52
  &, operator, 48, 51
  *, operator, 48, 51
  arytmetyka, 61, 62, 64
  do funkcji, 318, 322
  różne wielkości, 56
  stosowanie, 48
  typ, 62
  zmienna wskaźnikowa, 48
wycieki pamięci, 277, 563
wykrywacz włamań, 388, 390, 391, 392
wynikowy, kod, 2
wynikowy, plik, 196
wywołania systemowe, 396, 399, 401, 406, 409

Z

zmienne
  globalne, 47, 51, 80, 94
  lokalne, 47, 51
  środowiskowe, 405
  tablicowe, 54, 56, 59, 61, 64, 554
  wskaźnikowe, 48, 56
  współużytkowanie, 184

Ź

źródłowy, plik, 5
źródłowy, kod, 2

Notatki

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

GRUPA WYDAWNICZA

Perfekcyjny podręcznik do nauki C!

C. Rusz głową!

W obecnych czasach triumfy święcą platforma .NET, Java oraz HTML5 i JavaScript. Mogłoby się wydawać, że język C i inne podobne języki odeszły w niepamięć. Nic bardziej mylnego! W dalszym ciągu są one niezastąpione w wielu dziedzinach. Znajdują zastosowanie wszędzie tam, gdzie wymagana jest pełna kontrola nad sprzętem oraz gwarancja czasu wykonania powierzonych zadań. Dlatego specjaliści znający ten język wciąż są poszukiwani na rynku pracy.

Dzięki tej książce możesz dołączyć do ich grona! Kolejne wydanie z serii *Rusz głową!* to gwarancja sukcesu. Zastosowanie nowatorskich technik nauki pozwala na błyskawiczne przyswojenie wiedzy. W trakcie lektury poznasz składnię języka C, dostępne typy zmiennych, sposoby zarządzania pamięcią oraz zasady tworzenia przejrzystego kodu. Ponadto nauczysz się biegle obsługiwać kompilator, korzystać z plików nagłówkowych oraz przesyłać komunikaty między procesami. Dzięki licznym ćwiczeniom bez problemu utrwalisz zdobytą wiedzę. Książka ta jest wprost genialną pozycją dla wszystkich osób chcących wkroczyć w świat języka C. Przyda się również studentom na zajęciach z programowania. Warto ją mieć!

Poznaj język C — ta wiedza zaprocentuje!

Zainwestuj czas w naukę C i poznaj:

- składnię języka
- sposoby zarządzania pamięcią
- metody komunikacji z siecią
- możliwości kompilatora

helion.pl księgarnia internetowa

Nr katalogowy: 13284

Księgarnia internetowa: http://helion.pl

Zamówienia telefoniczne: 0 801 339900 0 601 339900

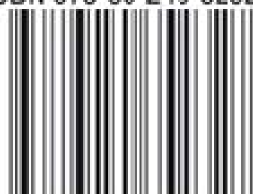
Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

ISBN 978-83-246-5232-7



Cena 99,00 zł