
Helion
PROGRAMOWANIE
W
C
Sprytne podejście do trudnych zagadnień,
których wolałbyś unikać (takich jak język C)
ZED A. SHAW

Spis treści

Podziękowania

Chciałbym podziękować trzem grupom osób, które pomogły mi w przygotowaniu tej książki: hejterom, pomocnikom i artystom malarzom.

Hejterzy przyczynili się do znacznego usprawnienia i bardziej rzetelnego przedstawienia materiału zawartego w książce, ponieważ wykazują się zupełnym brakiem elastyczności, irracjonalnym podejściem do starego, dobrego języka C oraz zupełnym brakiem doświadczenia pedagogicznego. Mając na uwadze takie osoby (absolutnie nie chciałbym być kimś takim), nigdy nie pracowałbym tak ciężko, aby niniejsza książka stała się kompletnym wprowadzeniem, pozwalającym Czytelnikowi zostać lepszym programistą.

Pomocnikami okazali się Debra Williams Cauley, Vicki Rowland, Elizabeth Ryan, cały zespół wydawnictwa Addison-Wesley oraz każdy internauta, który zdecydował się na przysłanie korekt i propozycji zmian. Dzięki pracy tych osób, zaproponowanym zmianom, korektom i poprawkom książka ta otrzymała ostateczną postać profesjonalnej i dopracowanej publikacji.

Artyści malarze, czyli Brian, Arthur, Vesta i Sarah, pomogli mi w znalezieniu nowego sposobu na wyrażenie samego siebie i oderwanie się od jasno wyznaczonych przez Deb i Vicki terminów zakończenia pracy, które notorycznie przekraczałem. Bez możliwości malowania i daru wrażliwości na sztukę, jaki otrzymałem od wymienionych malarzy, moje życie byłoby na pewno mniej sensowne i urozmaicone.

Dziękuję wszystkim osobom, które pomogły mi w napisaniu tej książki. Wprawdzie nie jest ona perfekcyjna — żadna książka nie jest taka — ale jest przynajmniej wystarczająco dobra, aby można było ją wydać.

Ta książka tak naprawdę nie jest o języku C

Proszę, nie czuj się oszukany, gdy powiem, że ta książka nie ma na celu nauczenia Cię programowania w języku C. Wprawdzie dowiesz się, jak tworzyć programy w C, ale najważniejsza umiejętność, jaką możesz zdobyć podczas lektury, to *ścisłe programowanie defensywne*. Obecnie zbyt wielu programistów przyjmuje założenie, że tworzony przez nich kod będzie działać zawsze i nigdy nie ulegnie awarii. W szczególności dotyczy to osób, które poznały przede wszystkim nowoczesne języki programowania, rozwiązujące wiele problemów za programistów. Dzięki lekturze tej książki i wykonaniu przedstawionych ćwiczeń dowiesz się, jak pisać oprogramowanie, które samo będzie w stanie bronić się przez złośliwą aktywnością i defektami.

Programuję w języku C z konkretnego powodu: uważam, że ten język jest zepsuty. Charakteryzuje się wieloma decyzjami projektowymi, które miały sens w latach 70. ubiegłego stulecia, a teraz są zupełnie pozbawione sensu. Wszystko, od niczym nieograniczonego użycia wskaźników aż po bezlitośnie zepsute ciągi tekstowe kończone znakiem NULL, można winić za praktycznie wszelkie luki w zabezpieczeniach, wykrywane w programach utworzonych w języku C. Według mnie język C jest tak bardzo zepsuty, że choć jest dość powszechnie stosowany, to jednak pozostaje językiem, w którym najtrudniej utworzyć bezpieczny kod. Mógłbym w tym miejscu pokusić się o stwierdzenie, że napisanie bezpiecznego kodu jest łatwiejsze w asemblerze niż w C. Szczerze mówiąc — wkrótce się przekonasz, że jestem pod tym względem niezwykle szczery — nie uważam, że należy tworzyć nowy kod źródłowy w C.

W takim razie mógłbyś zapytać, dlaczego zamierzam uczyć Cię tego języka. Odpowiedź jest prosta: ponieważ chcę, abyś stał się lepszym programistą — z dwóch następujących powodów. Po pierwsze, w języku C brakuje niemalże wszystkich nowoczesnych funkcji zabezpieczeń, co wymaga od programisty większej czujności i wiedzy o działaniu kodu. Jeżeli potrafisz utworzyć bezpieczny i niezawodny kod w języku C, to bez wątpienia będziesz umiał również tworzyć bezpieczny i niezawodny kod w dowolnym innym języku programowania. Zaprezentowane w tej książce techniki możesz zastosować w praktycznie każdym języku programowania, którego będziesz później używać. Po drugie, jeśli znasz C, zyskujesz bezpośredni dostęp do ogromnej ilości starszego kodu źródłowego, a ponadto masz opanowaną składnię bazową wielu języków pochodnych od C. Kiedy więc poznasz C, znacznie łatwiej będziesz mógł nauczyć się programowania w językach C++, Java, Objective-C i JavaScript. Także inne języki programowania staną się dla Ciebie łatwiejsze do opanowania.

Nie zamierzam Cię zniechęcać; mam nadzieję, że lektura dostarczy Ci niesamowitej frajdy, będzie łatwa i wciągająca. Dużą zaletą książki są zawarte w niej projekty, których prawdopodobnie nie realizowałeś w innych językach programowania. Książka jest łatwa — wykorzystałem sprawdzone wzorce ćwiczeń, które powinieneś *wykonać* w języku C, a nowy materiał wprowadzam powoli. Jest wciągająca, ponieważ pokazuje, jak zepsuć kod źródłowy, a następnie zabezpieczyć go, co pomaga w jeszcze lepszym zrozumieniu poszczególnych

koncepcji. Dowiesz się więc, jak przepełnić stos i jak uzyskać dostęp do niezarezerwowanej pamięci. Poznasz też częste mankamenty w programach utworzonych w języku C. Dzięki temu będziesz wiedział, czego należy unikać.

Podobnie jak w przypadku pozostałych moich książek, lektura tej może być pewnym wyzwaniem. Jednak jeśli przez to przebrniesz, staniesz się programistą znacznie lepszym i pewniejszym siebie.

Niezdefiniowane zachowania

Zanim zakończysz lekturę, zajmiesz się debugowaniem, analizą i poprawianiem praktycznie każdego uruchamianego programu w C, a następnie utworzysz nowy, solidny kod w języku C, którego potrzebujesz. Jednak tak naprawdę nie zamierzam uczyć Cię oficjalnego języka C. Wprawdzie poznasz język i dowiesz się, w jaki sposób go używać, ale oficjalny język C nie należy do zbyt bezpiecznych. Większość programistów nie tworzy solidnego kodu, co wynika z tak zwanego *niezdefiniowanego zachowania* (ang. *undefined behavior*). Niezdefiniowane zachowanie to część standardu **ANSI C** (ang. *American National Standards Institute*) wymieniającego wszystkie sposoby, na jakie kompilator języka C może odczytać tworzony przez Ciebie kod źródłowy. Nawet jeżeli będziesz tworzyć kod zgodny ze standardem ANSI C, działanie kompilatora wcale nie musi być spójne. Z niezdefiniowanym zachowaniem mamy do czynienia, gdy program w C odczytuje koniec ciągu tekstowego, co jest niezwykle często występującym błędem podczas programowania w języku C. Warto w tym miejscu przedstawić nieco kontekstu — C definiuje ciąg tekstowy jako bloki pamięci zakończone bajtem NULL, inaczej bajtem 0 (upraszczam tutaj definicję). Ponieważ wiele ciągów tekstowych pochodzi z zewnątrz programu, więc bardzo często zdarza się, że program w języku C otrzymuje ciąg tekstowy niezakończony bajtem NULL. W takim przypadku program próbuje wczytać do pamięci kolejne dane, co nieuchronnie prowadzi do awarii. Każdy inny język programowania opracowany po C próbuje uniknąć tego rodzaju sytuacji, ale nie C. Język C w bardzo niewielkim stopniu stara się chronić przed wystąpieniem niezdefiniowanego zachowania, co skłania programistów C do wniosku, że nie trzeba się przejmować tym problemem. Powstaje więc kod pełny potencjalnych pułapek związanych z brakiem znaku NULL na końcu ciągu tekstowego. Kiedy wskazujesz problematyczny kod, zwykle słyszysz: „To jest niezdefiniowane zachowanie, nie muszę się tym przejmować". Tego rodzaju ślepe zaufanie do C przyczynia się do powiększania się problemu niezdefiniowanego zachowania, co sprawia, że kod w języku C jest często bardzo niebezpieczny.

Kiedy tworzę kod w języku C, staram się uniknąć niezdefiniowanego zachowania. W tym celu opracowuję kod w taki sposób, aby wyeliminować niejasności, lub też tworzę kod, który próbuje bronić się przed niezdefiniowanym zachowaniem. Okazuje się jednak, że to zadanie bywa wręcz niemożliwe do zrealizowania, ponieważ istnieje *tak ogromna* liczba sytuacji, w których może wystąpić niezdefiniowane zachowanie, że stanowi to prawdziwy węzeł gordyjski połączonych ze sobą usterek kodu źródłowego w C. W książce pokażę Ci, jak można wywołać niezdefiniowane zachowanie, jak można go uniknąć (o ile to możliwe) oraz jak wywoływać je w kodzie źródłowym utworzonym przez innych programistów (o ile to możliwe). Powinieneś mieć jednak świadomość, że ucieczka przed właściwie losową naturą niezdefiniowanego zachowania jest praktycznie niemożliwa i można co najwyżej *starać się* tworzyć pozbawiony go kod.

OSTRZEŻENIE Przekonasz się, że fanatycy języka C będą bardzo często próbowali zbagatelizować istnienie niezdefiniowanego zachowania. To kategoria programistów, którzy nie tworzą zbyt dużej ilości kodu źródłowego w C, ale mają nieco wiedzy o niezdefiniowanym zachowaniu i posługując się nią, próbują okazać swoją wyższość nad początkującymi programistami C. Jeżeli natkniesz się na takie osoby, proszę, nie przejmuj się nimi. Najczęściej nie są one praktykującymi programistami C, są aroganckie, agresywne i zamęczą Cię niezliczonymi pytaniami, starając się okazać Ci swoją wyższość, a na pewno nie pomogą Ci podczas tworzenia kodu źródłowego. Jeżeli kiedykolwiek będziesz potrzebował pomocy w trakcie tworzenia kodu źródłowego w C, po prostu napisz do mnie na adres *help@learncodethehardway.org*. Pomogę Ci z przyjemnością.

C to język zarazem świetny i paskudny

Niezdefiniowane zachowanie to jeden z dodatkowych powodów, dla którego nauka języka C jest dobrym posunięciem, jeśli chcesz stać się lepszym programistą. Gdy będziesz umiał tworzyć dobry, solidny kod w C, wykorzystując przekazaną przeze mnie wiedzę, poradzisz sobie w *każdym* języku programowania. Oczywiście, C ma także dobre strony — pod pewnymi względami to naprawdę elegancki język. Jego składnia jest naprawdę prosta, biorąc pod uwagę potężne możliwości samego języka. Bez wątpienia w tym należy się dopatrywać przyczyny tego, że przez ostatnie 45 lat tak wiele języków programowania zapożyczyło składnię z C. Warto również dodać, że C oferuje niezwykle duże możliwości przy minimalnym wykorzystaniu technologii. Kiedy już poznasz C, docenisz elegancję i piękno tego języka, a jednocześnie dostrzeżesz jego wady. C jest starym językiem i podobnie jak piękny pomnik — z daleka wygląda fantastycznie, natomiast z bliska widać wszystkie rysy i pęknięcia.

Dlatego też zamierzam zaprezentować najnowszą wersję języka C, współdziałającą z najnowszymi wydaniami kompilatorów. Otrzymasz w ten sposób wiedzę o praktycznym, prostym i kompletnym podzbiorze C, który działa doskonale, sprawdza się wszędzie i pozwala na uniknięcie wielu pułapek. To będzie język C, którego osobiście używam w pracy, a nie encyklopedyczna wersja C, nieudolnie forsowana przez zatwardziałych fanatyków.

Wiem, że używany przeze mnie język C jest solidny, ponieważ już od ponad dwóch dekad tworzę w nim przejrzysty i niezawodny kod, przeznaczony do przeprowadzania ogromnych operacji. Jak dotąd kod ten mnie nie zawiódł. Prawdopodobnie przetworzył już tryliony transakcji, skoro jest wykorzystywany przez firmy takie jak Twitter i Airbnb. Rzadko zdarzały się awarie lub udane ataki na luki w zabezpieczeniach. Utworzony przeze mnie kod od lat stanowi podstawę świata sieci Ruby on Rails — gdzie działa znakomicie i nawet chroni przed atakami wykorzystującymi luki w zabezpieczeniach. Inne serwery WWW w internecie często padają ofiarami nawet najprostszych ataków.

Wypracowany przeze mnie styl tworzenia kodu źródłowego okazuje się niezawodny. Co ważniejsze, takie nastawienie do tworzenia kodu w języku C powinno być przyjmowane przez każdego programistę. To podejście do języka C i ogólnie programowania opiera się na jak

najlepszym wykonaniu zadania i na założeniu, że nic nie działa prawidłowo. Inni programiści — co zaskakujące, nawet dobrzy programiści C — mają tendencję do zakładania, że wszystko będzie działać zgodnie z oczekiwaniami, choć jednocześnie liczą na ratunek ze strony niezdefiniowanego zachowania lub systemu operacyjnego; z reguły jednak nie sprawdza się to jako rozwiązanie. Pamiętaj o tym, gdy ktokolwiek zarzuci Ci, że kod przedstawiony w książce nie jest „rzeczywistym C". Jeżeli taka osoba nie ma takiego samego doświadczenia i takich osiągnięć jak moje, może warto wykorzystać przedstawioną tutaj wiedzę i pokazać adwersarzowi, dlaczego jego kod nie jest zbyt bezpieczny.

Czy to oznacza, że opracowany przeze mnie kod jest doskonały? Oczywiście, że nie. To jest kod w języku C. Utworzenie idealnego kodu w języku C jest niemożliwe, zresztą podobnie jak w każdym innym języku programowania. Stanowi to frajdę i jednocześnie frustrację w programowaniu. Mógłbym wziąć kod opracowany przez innego programistę i rozłożyć go na części pierwsze — podobnie ktoś inny może postąpić z kodem utworzonym przeze mnie. Każdy kod zawiera pewne wady. Ale najważniejsza różnica polega na tym, że zawsze staram się przyjąć założenie o ułomności mojego kodu, a następnie próbuję eliminować mankamenty. Oto mój prezent dla Ciebie — postaraj się dotrzeć do końca tej książki i poznać nastawienie określane mianem *programowania defensywnego*, które doskonale mi służy od ponad dwóch dekad. To nastawienie pomogło mi w napisaniu wysokiej jakości, niezawodnego oprogramowania.

Czego się nauczysz?

Celem tej książki jest dostarczenie Ci solidnej wiedzy z zakresu języka C, abyś umiał z jego wykorzystaniem samodzielnie tworzyć oprogramowanie lub modyfikować kod źródłowy opracowany przez innych. Po lekturze powinieneś sięgnąć po książkę napisaną przez twórców języka C — Briana W. Kernighana i Dennisa Ritchiego, *Język ANSI C. Programowanie. Wydanie II* (Helion 2010). Dzięki mojej książce:

- poznasz podstawy składni C,
- poznasz kompilację, pliki Makefile i linkery,
- dowiesz się, jak wyszukiwać błędy i jak ich unikać,
- opanujesz praktyki programowania defensywnego,
- przekonasz się, jak łamać kod utworzony w C,
- zobaczysz, jak tworzyć podstawowe oprogramowanie dla systemu UNIX.

Zanim dotrzesz do ostatniego ćwiczenia, będziesz miał wystarczającą wiedzę, aby tworzyć proste oprogramowanie systemowe, biblioteki oraz inne mniejsze projekty.

Jak czytać tę książkę?

Ta książka jest przeznaczona dla programistów, którzy opanowali już przynajmniej jeden inny język programowania. Jeżeli nie znasz jeszcze żadnego, proponuję Ci inną moją książkę, *Learn Python the Hard Way* (Addison-Wesley 2013), która jest przeznaczona dla początkujących i doskonale sprawdza się jako pierwsza książka poświęcona programowaniu. Po zakończeniu lektury *Learn Python the Hard Way* powróć do tej książki.

Jeżeli masz już doświadczenie w tworzeniu kodu, na początku moja książka może wydawać się nieco dziwna. Nie przypomina innych, gdzie czytasz akapit po akapicie, a następnie wpisujesz wskazane polecenia. Zamiast tego przygotowałem do każdego ćwiczenia klip wideo — kod wprowadzasz na samym początku, a dopiero później omawiam, co zostało zrobione. Takie podejście sprawdza się lepiej, ponieważ polega na wyjaśnianiu tego, co już zrobiłeś, zamiast na odwoływaniu się do czegoś zupełnie abstrakcyjnego, gdy nie masz o tym żadnego pojęcia.

W zastosowanej przeze mnie strukturze książki istnieje kilka reguł, których *koniecznie* powinieneś przestrzegać:

- O ile nie powiem inaczej, najpierw obejrzyj wideo dla danego ćwiczenia.
- Samodzielnie wpisz cały kod, nie kopiuj go i nie wklejaj!
- Wpisz kod dokładnie w takiej postaci, w jakiej jest przedstawiony w książce, łącznie z komentarzami.
- Uruchom kod i upewnij się o otrzymaniu takich samych danych wyjściowych.
- Jeżeli wprowadzony kod zawiera jakiekolwiek błędy, usuń je.
- Wykonaj zadania dodatkowe, ale możesz pominąć te, na których zupełnie utknąłeś.
- Zawsze staraj się najpierw samodzielnie rozwiązać problem, a dopiero później szukaj pomocy.

Jeżeli zastosujesz się do powyższych reguł i wykonasz wszystkie ćwiczenia podane w książce, a mimo to nadal nie będziesz umiał tworzyć kodu w języku C, to będziesz wiedział, że przynajmniej spróbowałeś. Język C nie jest dla każdego, ale próba jego opanowania i tak czyni Cię lepszym programistą.

Wideo

Do każdego ćwiczenia przygotowałem wideo, do niektórych ćwiczeń nawet więcej niż tylko jedno. Klipy te powinny być uznawane za istotny składnik całości — mają ogromny wpływ na zastosowaną metodę edukacyjną. Wynika to z prostej przyczyny: *wiele* problemów pojawiających się podczas programowania w C to interaktywne kwestie związane z awarią kodu, debugowaniem i wydawaniem poleceń. Język C wymaga znacznie większej interakcji w celu usunięcia problemów, co jest przeciwieństwem sytuacji w językach takich jak Python i Ruby, gdzie kod po prostu działa. Dlatego też pewne zagadnienia (na przykład wskaźniki lub zarządzanie pamięcią) znacznie lepiej jest objaśnić na wideo, ponieważ wtedy mogę pokazać, jak faktycznie zachowuje się komputer.

O ile nie zostanie wskazane inaczej, przed przystąpieniem do lektury danego ćwiczenia należy więc najpierw obejrzeć wideo, a dopiero później zabrać się do wykonywania ćwiczeń. W niektórych ćwiczeniach jeden klip wideo wykorzystuję do zaprezentowania problemu, natomiast rozwiązanie znajdziesz w kolejnym klipie. W większości pozostałych ćwiczeń używam wideo do przedstawienia problemu, a następnie odsyłam Cię do ćwiczeń praktycznych, co wieńczy poznawanie danego zagadnienia.

Klipy wideo znajdziesz pod adresem: *helion.pl/pcspry*

Podstawowe umiejętności

Zgaduję, że masz doświadczenie w tworzeniu kodu w *mniej* wymagającym języku programowania. *Używalnymi* językami pozwalającymi na ucieczkę od niechlujnego myślenia i niepewnych sztuczek są między innymi Python i Ruby. Być może programowałeś wcześniej w języku LISP, który udawał, że komputer to wyłącznie funkcjonalna fantazja dla małych dzieci. A może poznałeś Prolog i uważasz, że cały świat powinien być bazą danych, po której się poruszasz i w której szukasz wskazówek. Co gorsza, jestem pewien, że używałeś zintegrowanych środowisk programistycznych (ang. *integrated development environment*) i Twój mózg jest pełen dziur w pamięci, więc możesz mieć nawet problem z wpisaniem pełnej nazwy funkcji bez naciskania klawiszy *Ctrl*+spacja po każdych trzech znakach.

Niezależnie od tego, jakie masz doświadczenie, prawdopodobnie będziesz mógł nieco poprawić umiejętności w wymienionych poniżej aspektach.

Czytanie i pisanie

To dotyczy w szczególności osób, które intensywnie korzystały ze zintegrowanych środowisk programistycznych. Zauważyłem ogólną prawidłowość, że programiści zbyt często jedynie przeglądają tekst i mają problemy z czytaniem ze zrozumieniem. Po prostu tylko przeglądają kod, którego sposób działania powinni dokładnie przeanalizować, i nie poświęcają wystarczająco dużo czasu na jego zrozumienie. Inne języki programowania oferują narzędzia pozwalające programistom na uniknięcie rzeczywistego pisania kodu. Gdy taka osoba stanie przed koniecznością utworzenia kodu w C, to pojawia się prawdziwy kłopot. Trzeba zacząć od uzmysłowienia sobie, że *każdy* ma taki problem. Rozwiązaniem jest zwolnienie tempa oraz skrupulatne czytanie tekstu i pisanie kodu. Na początku może się to wydawać bolesne i irytujące, ale jeśli często będziesz robił przerwy, zadanie stanie się łatwiejsze do wykonania.

Zwracanie uwagi na szczegóły

Każdy ma z tym problem i to jest jedna z najczęstszych przyczyn powstawania nieprawidłowo działającego oprogramowania. W innych językach programowania być może nie trzeba na to zwracać dużej uwagi, ale w przypadku C wymagane są pełna koncentracja i skupienie — kod jest uruchamiany bezpośrednio w maszynie, a sama maszyna jest bardzo wybredna. Gdy programujesz w C, nie ma miejsca na podejście w stylu „podobne do" lub „prawie" i dlatego trzeba zwracać dużą uwagę na szczegóły. Dwukrotnie sprawdzaj kod. Ponadto zakładaj, że może działać nieprawidłowo, o ile nie udowodnisz, że jest inaczej.

Wychwytywanie różnic

Poważnym problemem osób posiadających doświadczenie w pracy z innymi językami programowania jest to, że przystosowali mózg do wychwytywania różnic w *znanym* im języku, a nie w C. Kiedy utworzony przez siebie kod porównujesz z moim, Twój wzrok zatrzymuje się na znakach, o których sądzisz, że nie mają znaczenia, lub które pozostają dla Ciebie nieznane. Pokażę Ci, jak starać się wychwytywać popełnione błędy. Pamiętaj jednak, że jeśli Twój kod nie jest *dokładnie* taki sam jak w tej książce — będzie błędny.

Planowanie i debugowanie

Uwielbiam inne, łatwiejsze języki programowania, ponieważ mogę z nimi eksperymentować. Mam możliwość wprowadzenia koncepcji w interpreterze danego języka i natychmiast otrzymuję wynik. Takie rozwiązanie doskonale sprawdza się podczas wypróbowywania pomysłów. Czy jednak zwróciłeś uwagę, że jeśli stosujesz podejście *sztuczki aż do chwili, gdy kod działa*, to ostatecznie przygotowane rozwiązanie nie działa? Język C jest trudniejszy, ponieważ wymaga wcześniejszego zaplanowania oczekiwanego wyniku. Oczywiście możesz poeksperymentować z ideami, ale wcześniej niż w innych językach programowania musisz zabrać się do rzeczywistej pracy nad projektem. Pokażę Ci, jak planować kluczowe aspekty programu, jeszcze zanim przystąpisz do tworzenia kodu — to również pomaga w staniu się lepszym programistą. Nawet niewielki etap planowania może ułatwić późniejszą pracę nad oprogramowaniem.

Nauka języka C czyni z Ciebie lepszego programistę, ponieważ z wymienionymi powyżej kwestiami spotykasz się w jej trakcie odpowiednio wcześnie i często. Jeżeli napiszesz kod niechlujnie, po prostu nie będzie działał. Wielką zaletą języka C jest jego prostota — można go opanować samodzielnie. To doskonały język, pozwalający na poznanie maszyny i poszerzenie umiejętności w zakresie programowania.

Kody do pobrania

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

ftp://ftp.helion.pl/przyklady/pcspry.zip

Przygotowania

Rozdział wstępny jest tradycyjnie poświęcony przygotowaniu komputera na potrzeby pozostałych ćwiczeń. Tutaj zainstalujemy więc niezbędne pakiety i oprogramowanie, w zależności od używanego systemu operacyjnego.

Jeżeli napotkasz problemy, po prostu obejrzyj wideo przeznaczone dla tego ćwiczenia i wykonuj pokazane kroki. W ten sposób dowiesz się, jak wykonać poszczególne działania i rozwiązać ewentualne problemy, które mogą się pojawić po drodze.

Linux

Linux to system operacyjny prawdopodobnie najłatwiejszy do skonfigurowania pod kątem programowania w języku C. W przypadku systemów opartych nad dystrybucji Debian przejdź do powłoki i wydaj następujące polecenie:

```
$ sudo apt-get install build-essential
```

Jeżeli tę samą konfigurację chcesz uzyskać w dystrybucji opartej na pakietach RPM, na przykład Fedora, Red Hat lub CentOS, powinieneś wydać poniższe polecenie:

```
$ sudo yum groupinstall development-tools
```

Jeżeli korzystasz z jeszcze innej dystrybucji, po prostu wpisz w wyszukiwarce „c development tools" i nazwę dystrybucji, a dowiesz się, co i jak należy zainstalować. Po przeprowadzeniu instalacji i wydaniu polecenia:

```
$ cc --version
```

zobaczysz numer wersji zainstalowanego kompilatora. Najczęściej to będzie GNU C Compiler (`gcc`), ale nie przejmuj się, jeśli będzie inny niż użyty w książce. Możesz spróbować zainstalować kompilator Clang C, korzystając z instrukcji w sekcji *Getting Started* dla Twojej dystrybucji systemu Linux, bądź też poszukać odpowiednich informacji w internecie.

OS X

W systemie OS X instalacja jest jeszcze łatwiejsza. Wystarczy zainstalować najnowszą wersję programu Xcode lub odszukać ją na płytach dołączonych do komputera. Ponieważ archiwum może być ogromne, a jego pobranie wymagać dużej ilości czasu, w zależności od połączenia z internetem, więc zalecam przeprowadzenie instalacji z płyty. Odpowiednie instrukcje znajdziesz po wpisaniu w ulubionej wyszukiwarce internetowej „instalacja xcode". Inną możliwością jest instalacja Xcode ze sklepu AppStore, która przebiega podobnie jak instalacja każdego innego programu z tego sklepu. W takim przypadku będziesz miał dostęp do automatycznych aktualizacji Xcode.

Aby potwierdzić działanie kompilatora C, wydaj poniższe polecenie:

```
$ cc --version
```

Powinieneś otrzymać komunikat o użyciu kompilatora Clang C. W przypadku starszej wersji Xcode kompilatorem będzie GCC. Każdy z nich doskonale nadaje się do kompilacji kodu źródłowego przedstawionego w książce.

Windows

Jeżeli korzystasz z systemu Windows, gorąco zachęcam Cię do użycia oprogramowania Cygwin, ponieważ w ten sposób uzyskasz dostęp do wielu narzędzi programistycznych systemu UNIX. Wprawdzie instalacja Cygwin powinna być łatwa, ale zawsze możesz obejrzeć wideo, gdzie dokładnie omówiłem ten proces. Alternatywą dla Cygwin jest MinGW, to znacznie bardziej minimalistyczne rozwiązanie, choć powinno być wystarczające. Muszę Cię w tym miejscu ostrzec, że firma Microsoft stara się powoli wycofywać z obsługi języka C, więc możesz napotkać problemy z kompilacją kodu przedstawionego w książce za pomocą kompilatorów Microsoft.

Nieco bardziej zaawansowaną opcją będzie wykorzystanie programu VirtualBox w celu instalacji dystrybucji systemu Linux i uruchomienie pełnego Linuksa w systemie Windows. Zaletą takiego podejścia jest możliwość całkowitego usunięcia maszyny wirtualnej bez naruszania konfiguracji Windows. To także stwarza możliwość poznania systemu Linux, którego użycie dostarcza przyjemności, jak i może być korzystne w karierze programisty. Linux jest obecnie wdrażany jako główny system operacyjny dla wielu komputerów rozproszonych i infrastruktury w chmurze. Poznanie systemu Linux niewątpliwie zapewni Ci wiedzę przydatną w przyszłości.

Edytor tekstu

Wybór edytora tekstu dla programisty nie jest łatwy. Jeżeli dopiero zaczynasz przygodę z programowaniem, zaproponowałbym użycie gedit, ponieważ to prosty edytor i doskonale sprawdza się podczas tworzenia kodu. Jednak nie jest wystarczający w pewnych sytuacjach i jeśli masz choć niewielkie doświadczenie w programowaniu, to zapewne masz również ulubiony edytor tekstu.

Mając to na uwadze, proponuję Ci wypróbowanie kilku standardowych edytorów tekstu dla programistów, dostępnych na używanej platformie, a następnie wybór najodpowiedniejszego. Jeżeli używasz edytora gedit i lubisz go, pozostań przy nim. Natomiast jeśli chcesz poznać coś innego, wypróbuj kilka edytorów i zdecyduj się na któryś.

Najważniejsze jest to, aby *nie utknąć, próbując wybrać perfekcyjny edytor tekstu*. Podobnie jak każda inna kategoria oprogramowania, także edytory tekstu mają swoje wady. Po prostu wybierz jeden i pozostań przy nim, a jeśli później znajdziesz inny, wypróbuj go. Nie spędzaj całych dni na konfiguracji i próbie perfekcyjnego opanowania obsługi danego edytora tekstu.

Oto kilka edytorów, które warto wypróbować:

- gedit w systemach Linux i OS X,
- TextWrangler w systemie OS X,
- GNU nano działający w terminalu, dostępny dla wielu różnych systemów operacyjnych,
- GNU Emacs i GNU Emacs For Mac OS X; w tym przypadku przygotuj się na konieczność poświęcenia nieco czasu na poznanie sposobu obsługi edytora,
- Vim i MacVim.

Każdy ma jakieś upodobania, jeśli chodzi o edytor tekstu; powyżej wymieniłem jedynie kilka propozycji. Wypróbuj je i kilka innych, w tym komercyjnych, aż wreszcie znajdziesz edytor, który polubisz.

Nie używaj IDE

OSTRZEŻENIE Podczas nauki języka programowania staraj się unikać zintegrowanego środowiska programistycznego (IDE). Wprawdzie okazuje się ono pomocne, gdy trzeba wykonać pewne zadania, ale oferowana przez nie pomoc z reguły utrudnia faktyczne poznanie języka. Z mojego doświadczenia wynika, że dobry programista nie musi korzystać ze środowiska IDE i jest w stanie bez problemu tworzyć kod równie szybko, jak użytkownik pracujący z IDE. Ponadto przekonałem się, że kod wygenerowany przez IDE jest zwykle gorszej jakości. Nie wiem, dlaczego tak się dzieje, ale jeśli chcesz zdobyć solidne umiejętności w danym języku programowania, zachęcam do unikania stosowania IDE podczas nauki.

Umiejętność korzystania z profesjonalnego edytora tekstu dla programistów również jest użyteczna w karierze. Kiedy jesteś zależny od środowiska IDE, musisz czekać na wydanie jego nowej wersji, zanim będziesz mógł przystąpić do poznawania nowych języków programowania. Wiąże się to z pewnym kosztem w karierze zawodowej, czyli wstrzymaniem nauki, dopóki nowy język nie zyska większej popularności. Mając do dyspozycji zwykły edytor tekstu, jesteś w stanie tworzyć kod w dowolnym języku programowania, kiedy tylko zechcesz i nie czekając, aż ktoś dołączy jego obsługę do IDE. Umiejętność korzystania z edytora tekstu ogólnego przeznaczenia oznacza swobodę w rozwijaniu kariery programisty.

ĆWICZENIE 1

Odkurzenie kompilatora

Po zainstalowaniu wymaganego oprogramowania trzeba sprawdzić, czy kompilator na pewno działa. Najłatwiejszym sposobem jest utworzenie programu w języku C. Ponieważ powinieneś znać przynajmniej jeden język programowania, więc możemy zacząć od małego, choć nieco obszernego przykładu.

Plik *ex1.c*:

```
#include <stdio.h>

/* To jest komentarz. */
int main(int argc, char *argv[])
{
    int distance = 100;

    // To również jest komentarz.
    printf("Jesteś %d kilometrów stąd.\n", distance);

    return 0;
}
```

Jeżeli masz jakiekolwiek problemy z uruchomieniem powyższego kodu, najpierw obejrzyj wideo przeznaczone dla tego ćwiczenia.

Omówienie kodu w pliku

W przedstawionym powyżej fragmencie kodu wykorzystałem kilka funkcji języka C, których być może nie rozszyfrowałeś podczas wprowadzania poleceń. Dlatego też teraz omówię kod źródłowy wiersz po wierszu, a następnie przejdziemy do ćwiczeń pozwalających na jeszcze lepsze zrozumienie poszczególnych fragmentów programu. Nie przejmuj się, jeśli nie wszystko będzie tutaj dla Ciebie jasne. Po prostu chcę szybko pokazać Ci kod w języku C i *obiecuję*, że do wszystkich zawartych w nim koncepcji powrócimy w dalszej części książki.

Poniżej przedstawiłem omówienie kodu źródłowego wiersz po wierszu:

ex1.c:1. Polecenie `include` pozwala na zaimportowanie zawartości wskazanego pliku i umieszczenie jej w bieżącym pliku kodu źródłowego. W języku C stosowana jest konwencja użycia rozszerzenia *.h* dla plików nagłówkowych zawierających listy funkcji przeznaczonych do wykorzystania w programie.

ex1.c:3. To jest komentarz wielowierszowy. Między znakami otwierającym /* i zamykającym */ możesz umieścić dowolną liczbę wierszy tekstu.

ex1.c:4. Funkcja `main()` bardziej skomplikowana niż jej wersje, których używałeś dotąd. Działanie programu w języku C przedstawia się następująco: system operacyjny wczytuje program, a następnie wykonuje zdefiniowaną w nim funkcję `main()`.

Pełne wykonanie funkcji wymaga zwrotu wartości typu `int` oraz pobrania dwóch parametrów. Pierwszy, typu `int`, określa liczbę elementów w tablicy argumentów, natomiast drugi to tablica ciągów tekstowych (`*string`) przedstawiających argumenty. Nie mieści Ci się to w głowie? Nie przejmuj się, powrócimy do tego jeszcze w dalszej części książki.

ex1.c:5. Początkiem zawartości każdej funkcji jest nawias otwierający {, który wskazuje na początek tak zwanego *bloku*. W języku Python wystarczy wstawić dwukropek i zastosować wcięcie. W innych językach programowania może wystąpić konieczność użycia słowa kluczowego `begin` lub `do`.

ex1.c:6. Deklaracja zmiennej i przypisanie jej wartości. W taki właśnie sposób następuje utworzenie zmiennej za pomocą składni `nazwa = wartość;`. Polecenia w języku C (z wyjątkiem logiki) muszą być zakończone średnikiem.

ex1.c:8. To jest inny rodzaj komentarza. Przypomina komentarze stosowane w językach Python i Ruby, gdzie komentarz rozpoczyna się od znaków // i rozciąga aż do końca wiersza.

ex1.c:9. Odwołanie do starego dobrego przyjaciela, czyli funkcji `printf()`. Podobnie jak w wielu innych językach programowania, wywołanie funkcji ma składnię `nazwa(argument1, argument2);`, przy czym funkcja może nie mieć argumentów lub mieć ich dowolną liczbę. Funkcja `printf()` jest w rzeczywistości dość nietypowa i może pobierać wiele argumentów, o czym przekonasz się w dalszej części książki.

ex1.c:11. Wartość zwrotna funkcji `main()` przekazująca systemowi operacyjnemu wartość wyjścia. Być może nie wiesz, w jaki sposób oprogramowanie w systemie UNIX używa wartości zwrotnych, i dlatego do tego tematu również powrócimy w dalszej części książki.

ex1.c:12. Na końcu mamy nawias zamykający zawartość funkcji — }. W omawianym przykładzie jest to jednocześnie koniec programu.

To całkiem spora ilość informacji. Przeanalizuj je wiersz po wierszu i upewnij się, że przynajmniej ogólnie wiesz, o co tutaj chodzi. Wprawdzie nie musisz wiedzieć wszystkiego, ale wiele rzeczy prawdopodobnie i tak odgadniesz, zanim przejdziemy dalej.

Co powinieneś zobaczyć?

Przedstawiony powyżej kod źródłowy wpisz w pliku *ex1.c*, a następnie wydaj poniższe polecenia w powłoce. Jeżeli nie wiesz, jak to wszystko działa, obejrzyj wideo przeznaczone dla ćwiczenia 1.

Sesja dla ćwiczenia 1.:

```
$ make ex1
cc -Wall -g ex1.c -o ex1
$ ./ex1
Jesteś 100 kilometrów stąd.
$
```

Pierwsze polecenie to wywołanie narzędzia, które potrafi kompilować programy w języku C (oraz w wielu innych językach programowania). Po wydaniu polecenia `make` wraz z argumentem `ex1` nakazujesz narzędziu odszukanie pliku *ex1.c*, wywołanie kompilatora w celu kompilacji programu w podanym pliku i umieszczenie wyniku w pliku *ex1*. Wspomniany plik *ex1* jest wykonywalny, a więc możesz go uruchomić za pomocą polecenia `./ex1` i zobaczyć wygenerowane dane wyjściowe.

Jak to zepsuć?

Będę umieszczał w ćwiczeniach niewielką sekcję, w której będę pokazywał, jak można zepsuć omawiany program, oczywiście o ile istnieje taka możliwość. Będę Cię prosił o zrobienie nietypowych rzeczy z programami, na przykład o uruchamianie w programu dziwny sposób lub wprowadzenie zmian w kodzie źródłowym, które mogą doprowadzić do awarii programu i wygenerowania błędów przez kompilator.

W przypadku omawianego tutaj programu spróbuj usuwać losowo wybrane jego elementy i zobacz, czy nadal istnieje możliwość kompilacji kodu źródłowego. Po prostu zgaduj, co można usunąć, a następnie przeprowadzaj ponowną kompilację i sprawdzaj, czy w jej wyniku pojawił się błąd.

Zadania dodatkowe

- Otwórz plik *ex1.c* w edytorze tekstu, a następnie zmieniaj lub usuwaj losowo wybrane fragmenty programu. Spróbuj uruchomić zmodyfikowany program i obserwuj, jaki będzie wynik.
- Wyświetl pięć dodatkowych wierszy tekstu lub coś bardziej skomplikowanego niż jedynie „Witaj, świecie!”.
- W powłoce wydaj polecenie `man 3 printf`, aby dowiedzieć się nieco więcej na temat funkcji `print()` i innych.
- Z każdego wiersza kodu wypisz niezrozumiałe symbole lub polecenia, a następnie spróbuj odgadnąć ich przeznaczenie. Odpowiedzi zapisz na kartce, a później sprawdź, czy miałeś rację.

ĆWICZENIE 2

Użycie pliku Makefile podczas kompilacji

Wykorzystamy narzędzie o nazwie `make` do ułatwienia kompilacji kodu źródłowego wprowadzanego w poszczególnych ćwiczeniach. Narzędzie `make` istnieje od bardzo dawna i dlatego zostało dostosowane do kompilacji wielu rodzajów oprogramowania. W tym ćwiczeniu poznasz składnię pliku *Makefile*, co ułatwi Ci pracę w kolejnych ćwiczeniach. W dalszej części książki utrwalisz sobie składnię i sposób użycia tego pliku.

Użycie narzędzia make

Sposób działania narzędzia `make` polega na zadeklarowaniu zależności, a następnie opisaniu, jak je skompilować. Ewentualnie można wykorzystać wewnętrzną logikę tego narzędzia, obsługującą kompilację najczęściej spotykanych rodzajów oprogramowania. W narzędziu tym skumulowane są dekady wiedzy o kompilacji różnego rodzaju plików na podstawie innych plików. W poprzednim ćwiczeniu kompilację przeprowadziliśmy następująco:

```
$ make ex1
# Ewentualnie w poniższy sposób:
$ CFLAGS="-Wall" make ex1
```

W pierwszym wywołaniu informujesz narzędzie `make`, że chcesz utworzyć plik o nazwie *ex1*. Następnie program pobiera informacje dodatkowe i wykonuje następujące kroki:

1. Sprawdzenie, czy plik o nazwie *ex1* już istnieje.
2. Nie istnieje. Sprawdzenie, czy istnieje inny plik o nazwie rozpoczynającej się od *ex1*.
3. Tak, mamy plik *ex1.c*. Czy wiadomo, jak kompilować pliki *.c*?
4. Tak, kompilacja pliku *.c* wymaga wydania polecenia `cc ex1.c -o ex1`.
5. Utworzony powinien być jeden plik *ex1* za pomocą kompilatora `cc` kompilującego kod źródłowy w pliku *ex1.c*.

Natomiast w drugim wywołaniu widzimy przekazanie *modyfikatorów* narzędziu `make`. Jeżeli nie znasz sposobu działania powłoki w systemie UNIX, oto krótkie wyjaśnienie: tworzymy wymienione *zmienne środowiskowe*, które będą wykorzystywane przez uruchamiane programy. Czasami do utworzenia zmiennych środowiskowych stosuje się polecenie takie jak `export CFLAGS="Wall"`, w zależności od używanej powłoki. Definicję zmiennej środowiskowej można umieścić też przed poleceniem przeznaczonym do wykonania i wówczas dana zmienna będzie zdefiniowana jedynie dla tego polecenia.

W omawianym przykładzie mamy polecenie `CFLAGS="-Wall" make ex1`, co oznacza dodanie opcji `-Wall` do wykonywanego polecenia `cc`. Opcja ta nakazuje kompilatorowi `cc` wyświetlanie wszystkich komunikatów z ostrzeżeniami (które jednak chorym zrządzeniem losu nie są wszystkimi ostrzeżeniami, jakie mogą być wygenerowane).

Używając narzędzia `make` w pokazany powyżej sposób, możesz zdziałać naprawdę wiele. Jednak teraz przejdziemy do pliku *Makefile*, co pozwoli Ci jeszcze lepiej zrozumieć sposób działania `make`. Rozpoczynamy od utworzenia pliku z podaną poniżej zawartością.

Plik *ex2.1.mak*:

```
CFLAGS=-Wall -g

clean:
    rm -f ex1
```

Zapisz plik w katalogu bieżącym i nadaj mu nazwę *Makefile*. Program automatycznie zakłada, że istnieje plik o nazwie *Makefile*, i używa go.

OSTRZEŻENIE Upewnij się, że stosujesz jedynie tabulatory, a nie połączenia tabulatorów i spacji.

Powyższy plik *Makefile* pokazuje nowe opcje przeznaczone do wykorzystania przez `make`. Przede wszystkim widzimy ustawienie zmiennej środowiskowej `CFLAGS` w pliku, co uwalnia nas od konieczności jej każdorazowego definiowania w powłoce. Ponadto dodaliśmy opcję `-g` wskazującą na debugowanie. Dalej mamy sekcję o nazwie `clean` wskazującą działania podejmowane podczas operacji porządkujących w trakcie kompilacji naszego małego projektu.

Upewnij się, że utworzony plik *Makefile* znajduje się w tym samym katalogu wraz z *ex1.c*, a następnie wydaj poniższe polecenia:

```
$ make clean
$ make ex1
```

Co powinieneś zobaczyć?

Jeżeli wszystko przebiega zgodnie z oczekiwaniami, otrzymasz następujące dane wyjściowe.

Sesja dla ćwiczenia 2.:

```
$ make clean
rm -f ex1
$ make ex1
cc -Wall -g ex1.c -o ex1
ex1.c: In function 'main':
ex1.c:3: warning: implicit declaration of function 'puts'
$
```

Jak możesz zobaczyć, najpierw mamy wywołanie `make clean`, co oznacza wykonanie zdefiniowanej wcześniej sekcji `clean`. Powróć do kodu źródłowego pliku *Makefile* i zobacz, jakie polecenia znajdują się w wymienionej sekcji. Mamy więc wcięcie oraz polecenie powłoki, które będzie automatycznie wykonane. Możesz w tym miejscu umieścić dowolną liczbę poleceń — otrzymujesz tym samym doskonałe narzędzie automatyzacji.

OSTRZEŻENIE Jeżeli zmieniłeś kod źródłowy w pliku *ex1.c*, aby zawierał polecenie `#include <stdio.h>`, to wygenerowane dane wyjściowe nie będą zawierały komunikatu z ostrzeżeniem (to tak naprawdę powinien być błąd) dotyczącym funkcji `printf()`. W omawianym przykładzie mamy komunikat o błędzie, ponieważ nie zmodyfikowałem wymienionego polecenia w kodzie źródłowym.

Zwróć uwagę na brak jakiejkolwiek wzmianki `ex1` w pliku *Makefile*, a mimo to narzędzie `make` nadal wie, jak skompilować plik *oraz* wykorzystuje zdefiniowane przez nas ustawienia specjalne.

Jak to zepsuć?

Przedstawione w ćwiczeniu informacje powinny wystarczyć na początek. W tym miejscu zepsujemy plik *Makefile* w określony sposób, aby zobaczyć, jaki będzie tego efekt. Przejdź do wiersza `rm -f ex1` i usuń wcięcie (przesuń kod do lewej krawędzi), i zobacz, co się stanie. Po ponownym wykonaniu polecenia `make clean` powinien pojawić się komunikat podobny do poniższego:

```
$ make clean
Makefile:4: *** missing separator. Stop.
```

Nigdy nie zapominaj o wcięciach. Jeżeli pojawią się dziwne komunikaty, podobne do powyższego, dokładnie sprawdź, czy spójnie używasz tabulatorów, ponieważ niektóre wersje narzędzia `make` są pod tym względem naprawdę niezwykle wybredne.

Zadania dodatkowe

- Utwórz sekcję `all: ex1`, która przeprowadzi kompilację `ex1` po prostu za pomocą polecenia `make`.
- Zapoznaj się z zawartością podręcznika systemowego `man make`, aby dowiedzieć się więcej na temat sposobów użycia wymienionego narzędzia.
- Zapoznaj się z zawartością podręcznika systemowego `man cc`, aby dowiedzieć się więcej na temat działania opcji `-Wall` i `-g`.
- Poszukaj w internecie informacji dotyczących pliku *Makefile* i zobacz, czy potrafisz usprawnić plik utworzony w tym ćwiczeniu.
- Odszukaj plik *Makefile* w innym projekcie utworzonym w języku C i spróbuj zrozumieć sposób jego działania.

ĆWICZENIE 3

Sformatowane dane wyjściowe

Zachowaj w pobliżu plik *Makefile*, ponieważ będzie pomocny w wychwytywaniu błędów. Gdy zajdzie potrzeba automatyzacji kolejnych zadań, umieścimy w nim następne polecenia.

W wielu językach programowania używany jest znany z C sposób formatowania danych wyjściowych. Spróbujmy więc sformatować dane.

Plik *ex3.c*:

```
#include <stdio.h>

int main()
{
    int age = 10;
    int height = 183;

    printf("Mam %d lat.\n", age);
    printf("Mam %d cm wzrostu.\n", height);

    return 0;
}
```

Po wpisaniu kodu źródłowego wydaj standardowe polecenie `make ex3` w celu kompilacji kodu, a następnie uruchom go. Upewnij się o poprawieniu *wszystkich błędów prowadzących do wygenerowania komunikatów z ostrzeżeniami*.

Wprawdzie kod źródłowy przedstawiony w ćwiczeniu jest stosunkowo krótki, ale dużo się w nim dzieje i dlatego poniżej przedstawiam jego dokładne omówienie.

- Na początku dołączamy *plik nagłówkowy* o nazwie *stdio.h*. W ten sposób informujemy kompilator, że będziemy używać standardowych funkcji wejścia-wyjścia. Jedną z tego rodzaju funkcji jest `printf()`.
- Następnie definiujemy zmienną o nazwie `age` i przypisujemy jej wartość `10`.
- Kolejnym krokiem jest zdefiniowanie zmiennej `height` i przypisanie jej wartości `183`.
- Dalej mamy dwa wywołania funkcji `printf()` w celu wyświetlenia informacji o wieku i wzroście najwyższego dziesięciolatka na świecie.
- W wywołaniach funkcji `printf()` zwróć uwagę na obecność ciągu tekstowego formatowania, podobnie jak w wielu innych językach programowania.
- Po ciągu tekstowym formatowania znajduje się nazwa zmiennej, której wartość powinna być przez `printf()` umieszczona w miejscu ciągu tekstowego formatowania.

Wynikiem będzie pobranie przez funkcję `printf()` wartości pewnych zmiennych, utworzenie nowego ciągu tekstowego, a następnie wyświetlenie go w powłoce.

Co powinieneś zobaczyć?

Po przeprowadzeniu kompilacji kodu źródłowego powinieneś otrzymać następujące dane wyjściowe.

Sesja dla ćwiczenia 3.:

```
$ make ex3
cc -Wall -g ex3.c -o ex3
$ ./ex3
Mam 10 lat.
Mam 183 cm wzrostu.
$
```

Już wkrótce przestanę Ci przypominać o konieczności wydania polecenia `make` i nie będę przedstawiał procesu kompilacji, więc upewniaj się, że wszystko działa zgodnie z oczekiwaniami.

Zewnętrzne badania

W poszczególnych ćwiczeniach zawarta jest sekcja „Zadania dodatkowe", w której znajdują się ćwiczenia polegające na samodzielnym wyszukiwaniu informacji i ustalaniu sposobu działania pewnych mechanizmów. To niezwykle ważny aspekt pracy samodzielnego programisty. Jeżeli nieustannie zadajesz pytania innym osobom, zanim spróbujesz samodzielnie uporać się z danym kłopotem, nigdy nie będziesz w stanie samodzielnie rozwiązywać problemów. Dlatego też nigdy nie rozwiniesz własnych umiejętności i zawsze będziesz potrzebował kogoś, kto znajdzie się w pobliżu i pomoże Ci w wykonaniu pracy.

Sposobem na przerwanie złego nawyku jest *zmuszenie* się do próby samodzielnego odpowiadania sobie na pytania, a dopiero później potwierdzania poprawności odpowiedzi. W tym celu zachęcam Cię do psucia kodu, eksperymentowania z udzielanymi odpowiedziami i samodzielnego szukania informacji.

W tym ćwiczeniu proszę Cię, abyś w internecie odnalazł informacje o *wszystkich* kodach i sekwencjach sterujących funkcji `printf()`. Przykładami kodów sterujących są `\n` i `\t`, które powodują wyświetlenie odpowiednio znaku nowego wiersza i tabulatora. Natomiast przykładami sekwencji sterujących są `%s` i `%d`, które wyświetlają odpowiednio ciąg tekstowy i liczbę całkowitą. Odszukaj wszystkie kody i sekwencje sterujące, sprawdź, jak można je modyfikować, a następnie ustal obsługiwane przez nie poziomy „dokładności".

Od teraz tego rodzaju zadania będą pojawiały się w sekcjach „Zadania dodatkowe" i zdecydowanie powinieneś je wykonywać.

Jak to zepsuć?

Spróbuj zepsuć program na kilka z przedstawionych poniżej sposobów, które mogą, choć nie muszą doprowadzić do zawieszenia komputera.

- Usuń zmienną `age` z pierwszego wywołania funkcji `printf()`, a następnie ponownie skompiluj program. Powinno się pojawić kilka komunikatów z ostrzeżeniami.
- Uruchom nowo skompilowany program, który ulegnie awarii lub wyświetli naprawdę dziwne informacje o wieku.
- Przywróć pierwotną postać wywołania `printf()`, a następnie w ogóle nie przypisz wartości początkowej zmiennej `age`. W tym celu zmień polecenie na `int age;`, a później ponownie skompiluj i uruchom program.

Nieprawidłowa sesja dla ćwiczenia 3.:

```
# Edycja pliku ex3.c w celu spowodowania niepowodzenia wywołania printf().
$ make ex3
cc -Wall -g ex3.c -o ex3
ex3.c: In function 'main':
ex3.c:8: warning: too few arguments for format
ex3.c:5: warning: unused variable 'age'
$ ./ex3
Mam -919092456 lat.
Mam 183 cm wzrostu.
# Ponowna edycja ex3.c i przywrócenie pierwotnej postaci printf(),
# ale niezainicjalizowanie zmiennej age.
$ make ex3
cc -Wall -g ex3.c -o ex3
ex3.c: In function 'main':
ex3.c:8: warning: 'age' is used uninitialized in this function
$ ./ex3
Mam 0 lat.
Mam 183 cm wzrostu.
$
```

Zadania dodatkowe

- Znajdź jak najwięcej sposobów na zepsucie programu zdefiniowanego w pliku *ex3.c*.
- Wyświetl podręcznik systemowy `man 3 printf` i wyszukaj informacje o innych sekwencjach sterujących `%`, które możesz wykorzystać. Odszukane sekwencje powinny wyglądać znajomo, jeśli miałeś z nimi styczność w innych językach programowania (są pochodnymi funkcji `printf()`).
- Dodaj `ex3` do sekcji `all` w pliku *Makefile*. Następnie wydaj polecenie `make clean all` w celu kompilacji wszystkich utworzonych dotąd plików kodu źródłowego.
- Dodaj `ex3` także do sekcji `clean` w pliku *Makefile*. Wydaj polecenie `make clean` w celu usunięcia programu, gdy zachodzi potrzeba.

ĆWICZENIE 4

Użycie debugera

Materiał przedstawiony w tym ćwiczeniu opiera się w bardzo dużym stopniu na klipie wideo, w którym pokazuję, jak debuger dostarczany wraz z kompilatorem można wykorzystać do usuwania błędów w programie, wykrywania błędów, a nawet debugowania aktualnie działających procesów. Proszę, obejrzyj wideo przeznaczone dla tego ćwiczenia, aby dowiedzieć się więcej o usuwaniu błędów.

Sztuczki z GDB

Poniżej wymieniłem listę prostych sztuczek, do których można użyć GDB (GNU Debugger):

- **`gdb --args`**. Standardowo debuger pobiera przekazane mu argumenty i zakłada, że są przeznaczone dla niego. Dzięki opcji `--args` argumenty będą przekazane do debugowanego programu.
- **`thread apply all bt`**. Zebranie stosu wywołań dla *wszystkich* wątków. To jest bardzo użyteczna możliwość.
- **`gdb --batch --ex r --ex bt --ex q --args`**. Uruchomienie programu, jeśli jego działanie zakończy się awarią, to otrzymasz informacje o stosie wywołań.

Krótki przewodnik po GDB

W tym wideo pokazuję, jak można korzystać z debugera. Jednak w trakcie pracy będziesz musiał odwoływać się do pewnych poleceń. Poniżej przedstawiłem więc krótką listę poleceń GDB, które wykorzystałem w wideo przeznaczonym dla tego ćwiczenia. Poleceń tych będziemy używać w dalszej części książki.

`run [argumenty]`. Uruchomienie programu wraz z podanymi `argumentami`.

`break [plik:]funkcja`. Zdefiniowanie punktu kontrolnego w miejscu `[plik:]funkcja`. Można również użyć opcji `b`.

`backtrace`. Zapisanie informacji o bieżącym stosie wywołań. Skrótem tej opcji jest `bt`.

`print wyrażenie`. Wyświetlenie wartości `wyrażenia`. Skrótem tej opcji jest `p`.

`continue`. Kontynuacja działania programu. Skrótem tej opcji jest `c`.

`next`. Przejście do kolejnego wiersza, ale *z pominięciem* wywołań funkcji. Skrótem tej opcji jest `n`.

`step`. Przejście do kolejnego wiersza, ale z wejściem *do* wywołań funkcji. Skrótem tej opcji jest `s`.

`quit`. Opuszczenie GDB.

help. Wyświetlenie informacji o dostępnych typach poleceń. Następnie można wyświetlić informacje zarówno o klasie polecenia, jak i o samym poleceniu.

cd, **pwd**, **make**. Działanie tych poleceń jest dokładnie takie samo jak po ich wydaniu w powłoce.

shell. Szybkie uruchomienie powłoki, co pozwala na wykonanie innych zadań.

clear. Usunięcie punktu kontrolnego.

info break, **info watch**. Wyświetlenie informacji o punktach kontrolnych i czujkach.

attach pid. Dołączenie debugera do działającego procesu, co pozwala na jego debugowanie.

detach. Odłączenie debugera od procesu.

list. Wyświetlenie kolejnych dziesięciu wierszy kodu źródłowego. Po dodaniu opcji - można wyświetlić poprzednie dziesięć wierszy.

Krótki przewodnik po LLDB

W systemie OS X nie jest dłużej dostępny debuger GDB i zamiast niego trzeba wykorzystać LLDD. Polecenia są niemalże takie same; poniżej przedstawiłem ich krótką listę.

run [argumenty]. Uruchomienie programu wraz z podanymi `argumentami`.

breakpoint set --nazwa [plik:]funkcja. Zdefiniowanie punktu kontrolnego w miejscu `[plik:]funkcja`. Można również użyć opcji `b`, co jest znacznie łatwiejsze.

thread backtrace. Zapisanie informacji o bieżącym stosie wywołań. Skrótem tej opcji jest `bt`.

print wyrażenie. Wyświetlenie wartości `wyrażenia`. Skrótem tej opcji jest `p`.

continue. Kontynuacja działania programu. Skrótem tej opcji jest `c`.

next. Przejście do kolejnego wiersza, ale *z pominięciem* wywołań funkcji. Skrótem tej opcji jest `n`.

step. Przejście do kolejnego wiersza, ale z wejściem *do* wywołań funkcji. Skrótem tej opcji jest `s`.

quit. Opuszczenie LLDB.

help. Wyświetlenie informacji o dostępnych typach poleceń. Następnie można wyświetlić informacje zarówno o klasie polecenia, jak i o samym poleceniu.

cd, **pwd**, **make**. Działanie tych poleceń jest dokładnie takie samo, jak po ich wydaniu w powłoce.

shell. Szybkie uruchomienie powłoki, co pozwala na wykonanie innych zadań.

clear. Usunięcie punktu kontrolnego.

info break, **info watch**. Wyświetlenie informacji o punktach kontrolnych i czujkach.

`attach -p pid`. Dołączenie debugera do działającego procesu, co pozwala na jego debugowanie.

`detach`. Odłączenie debugera od procesu.

`list`. Wyświetlenie kolejnych dziesięciu wierszy kodu źródłowego. Po dodaniu opcji - można wyświetlić poprzednie dziesięć wierszy.

W internecie znajdziesz zarówno przewodniki, jak i samouczki dotyczące GDB i LLDB.

ĆWICZENIE 5

Nauka na pamięć operatorów w C

Kiedy poznawałeś pierwszy język programowania, prawdopodobnie byłeś zagłębiony w lekturze książki, wpisywałeś nie do końca zrozumiały dla Ciebie kod, a następnie próbowałeś określić, jak on działa. W taki właśnie sposób napisałem większość innych moich książek, a wspomniane podejście doskonale sprawdza się w przypadku początkujących. Gdy dopiero zaczynasz przygodę z programowaniem istnieją pewne skomplikowane zagadnienia, które musisz poznać, zanim zrozumiesz znaczenie wszystkich symboli i słów, a więc to łatwy sposób nauki.

Jednak gdy już opanujesz jeden język programowania, podejście polegające na gmeraniu w składni nie będzie najefektywniejszym sposobem poznawania nowego języka. Wprawdzie sprawdzi się to, ale nowy język opanujesz znacznie szybciej, gdy po prostu zaczniesz z niego korzystać. Taka metoda poznawania języka programowania może wydawać się magią, ale zaufaj mi, że to podejście sprawdza się zaskakująco doskonale.

Mój sposób nauczenia Cię języka C jest następujący: *najpierw* powinieneś poznać oraz zapamiętać wszystkie podstawowe symbole i składnię, a *następnie* wykorzystać tę wiedzę w serii ćwiczeń. Tego rodzaju metoda jest niezwykle podobna do nauki języka obcego, gdzie zaczynasz od uczenia się na pamięć słów i gramatyki, a dopiero później przechodzisz do konwersacji. Na początku musisz podjąć wysiłek polegający na nauczeniu się na pamięć pewnych informacji, w ten sposób zdobywasz podstawową wiedzę, która później ułatwi Ci odczyt i tworzenie kodu w języku C.

OSTRZEŻENIE Niektórzy są przeciwni uczeniu się na pamięć pewnych informacji i twierdzą, że takie rozwiązanie jest niekreatywne i nudne. Jestem dowodem na to, że uczenie się na pamięć pewnych koncepcji nie zabija kreatywności i na pewno nie jest nudne. Maluję, buduję gitary i gram na nich, śpiewam, tworzę kod, piszę książki *i* uczę się na pamięć wielu różnych rzeczy. Wymienione przekonanie jest całkowicie nieuzasadnione i szkodliwe w trakcie procesu nauki. Dlatego też ignoruj próby zniechęcania Cię do uczenia się na pamięć.

Jak uczyć się na pamięć?

Najlepszy sposób nauki na pamięć pewnych informacji to zdumiewająco prosty proces.

1. Przygotuj zestaw kart, na jednej stronie umieść symbol, a na drugiej jego opis. W komputerze możesz do tego celu wykorzystać program o nazwie Anki. Osobiście preferuję ręczne tworzenie fizycznych kart, ponieważ pomaga mi to w zapamiętywaniu informacji.
2. Losowo wybieraj karty i zaczynaj czytać treść umieszczoną na jednej stronie. Postaraj się przypomnieć sobie zawartość drugiej strony, ale bez podglądania.

3. Jeżeli nie możesz sobie przypomnieć tego, co znajduje się na drugiej stronie, odwróć kartę, powtórz odpowiedź i odłóż kartę na oddzielny stos.

Po przejrzeniu wszystkich kart będziesz miał dwa stosy. Pierwszy zawiera karty, na których informacje szybko sobie przypomniałeś. Natomiast drugi to stos kart, na których informacje sprawiły trudność. *Weź ten drugi stos i powtórz informacje zapisane na kartach.*

Na końcu sesji, która zwykle trwa od 15 do 30 minut otrzymujesz zbiór kart z informacjami trudnymi do zapamiętania. Weź te karty ze sobą i ćwicz, gdy tylko masz wolną chwilę.

Wprawdzie istnieje wiele innych sposobów uczenia się na pamięć, ale przedstawiony powyżej sprawdza się najlepiej, gdy szybko chcesz przypomnieć sobie dane, których trzeba natychmiast użyć. Symbole, słowa kluczowe i składnia języka C to dane, które trzeba natychmiast sobie przypominać. Dlatego też podana metoda jest bardzo dobra.

Pamiętaj także o konieczności przyswajania informacji znajdujących się po *obu* stronach kart. Powinieneś być w stanie przeczytać opis i wiedzieć, jakiemu symbolowi odpowiada, a także znać opis dla danego symbolu.

Na koniec warto przypomnieć: *nie przestawaj powtarzać materiału*, gdy opanujesz przedstawione tutaj operatory. Najlepsze podejście polega na połączeniu powtórzeń wraz z ćwiczeniami przedstawionymi w książce, co pozwala na praktyczne wykorzystanie zapamiętanych informacji. Więcej na ten temat znajdziesz w kolejnym ćwiczeniu.

Listy operatorów

Pierwsza grupa to operatory arytmetyczne, które są podobne do stosowanych w praktycznie wszystkich pozostałych językach programowania. Opis na kartach powinien zawierać wyraźne informacje o tym, że to jest operator arytmetyczny, oraz o sposobie jego działania.

Operatory arytmetyczne	
Operator	**Opis**
+	Dodawanie
-	Odejmowanie
*	Mnożenie
/	Dzielenie
%	Modulo
++	Inkrementacja
--	Dekrementacja

Operatory relacji są przeznaczone do sprawdzania wartości i również są powszechnie stosowane w językach programowania.

Operatory relacji	
Operator	**Opis**
==	Równość
!=	Nierówność
>	Większy niż
<	Mniejszy niż
>=	Większy lub równy
<=	Mniejszy lub równy

Operatory logiczne są przeznaczone do przeprowadzania testów logicznych i powinieneś już wiedzieć, do czego służą. Jedynym dziwnym operatorem jest tutaj *logiczny operator trójargumentowy*, do którego powrócimy jeszcze w dalszej części książki.

Operatory logiczne	
Operator	**Opis**
&&	Logiczne I (AND)
||	Logiczne LUB (OR)
!	Logiczne NIE (NOT)
? :	Logiczny operator trójargumentowy

Operatory bitowe wykonują operacje, z którymi nie będziesz się zbyt często spotykał w nowoczesnym kodzie źródłowym. Na różne sposoby zmieniają bity tworzące bajty lub inne typy danych. W tej książce nie będziemy zajmować się operatorami bitowymi, ale musisz wiedzieć, że okazują się pomocne podczas pracy z określonymi typami systemów niskiego poziomu.

Operatory bitowe	
Operator	**Opis**
&	Bitowe I (AND)
|	Bitowe LUB (OR)
^	Bitowe wykluczające LUB (XOR)
~	Odwrócenie znaczenia bitu
<<	Bitowe przesunięcie w lewo
>>	Bitowe przesunięcie w prawo

Operatory przypisania służą po prostu do przypisywania wyrażeń zmiennym, choć język C łączy z przypisaniem ogromną liczbę innych operatorów. Dlatego też „i-równość" oznacza operator *bitowy*, a nie logiczny.

Operatory przypisania	
Operator	**Opis**
=	Przypisanie wartości
+=	Dodanie wartości i przypisanie wyniku
-=	Odjęcie wartości i przypisanie wyniku
*=	Mnożenie wartości i przypisanie wyniku
/=	Dzielenie wartości i przypisanie wyniku
%=	Modulo wartości i przypisanie wyniku
<<=	Przesunięcie w lewo wartości i przypisanie wyniku
>>=	Przesunięcie w prawo wartości i przypisanie wyniku
&=	Bitowe I wartości i przypisanie wyniku
^=	Bitowe XOR wartości i przypisanie wyniku
|=	Bitowe LUB wartości i przypisanie wyniku

Wprawdzie operatory wymienione w poniższej tabeli nazywam *operatorami danych*, ale tak naprawdę współdziałają one z różnymi aspektami wskaźników, dostępem do elementów składowych, a także różnymi elementami struktur danych w języku C.

Operatory danych	
Operator	**Opis**
sizeof()	Pobranie wielkości danego elementu
[]	Notacja tablicy
&	Adres wskazanego elementu
*	Wartość wskazanego elementu
->	Dereferencja do struktury
.	Referencja do struktury

Na końcu mamy kilka różnych symboli, które są często wykorzystywane w różnorodnych rolach (na przykład przecinek) lub trudno je z pewnych powodów zaliczyć do jednej z wcześniejszych kategorii.

Operatory różne	
Operator	**Opis**
,	Przecinek
( )	Nawias okrągły
{ }	Nawias klamrowy
:	Dwukropek
//	Początek komentarza jednowierszowego
/*	Początek komentarza wielowierszowego
*/	Koniec komentarza wielowierszowego

W trakcie lektury książki nie zapominaj o ćwiczeniach z przygotowanymi kartami. Jeżeli przed lekturą poświęcisz 15 – 30 minut na ćwiczenia i podobną ilość czasu przed snem, to wszystkie wymienione tutaj operatory powinieneś zapamiętać w ciągu kilku tygodni.

ĆWICZENIE 6

Nauka na pamięć składni C

Po poznaniu operatorów nadeszła pora na nauczenie się na pamięć słów kluczowych oraz podstawowej składni używanych struktur. Zaufaj mi, gdy twierdzę, że niewielka ilość czasu poświęcona na zapamiętanie wymienionych rzeczy zwróci się wielokrotnie podczas lektury książki.

Jak wspomniałem w ćwiczeniu 5., nie musisz czekać z lekturą książki aż do nauczenia się na pamięć tych rzeczy. Wręcz przeciwnie, powinieneś uczyć się i jednocześnie wykonywać ćwiczenia prezentowane w książce. Przygotowane karty wykorzystaj jako rozgrzewkę przed przystąpieniem do tworzenia kodu. Powtarzaj materiał przez 15 – 30 minut, a następnie wykonuj ćwiczenia programistyczne podane w książce. W trakcie lektury późniejszych ćwiczeń spróbuj potraktować wpisywany kod jako sposób przećwiczenia zapamiętanych dotąd informacji. Dobrze jest przygotować oddzielne stosy kart dla operatorów i słów kluczowych, aby można było łatwo je odróżniać podczas tworzenia kodu. Na koniec dnia przez kolejne 15 – 30 minut powtarzaj informacje zapisane na kartach.

Warto pamiętać, że nauka języka C będzie szybsza i trwalsza, gdy będziesz samodzielnie wprowadzał cały kod aż do chwili zapamiętania informacji.

Słowa kluczowe

W języku programowania *słowa kluczowe* to słowa wspomagające symbole, aby język jeszcze lepiej mógł uwzględnić intencje programisty. Istnieją pewne języki programowania, na przykład APL, które tak naprawdę są pozbawione słów kluczowych. Z kolei inne, takie jak Forth i LISP, składają się praktycznie z samych słów kluczowych. Gdzieś pośrodku znajdują się kolejne języki, na przykład C, Python, Ruby i wiele innych, zawierających połączenie słów kluczowych i symboli, które w ten sposób tworzą podstawy danego języka.

OSTRZEŻENIE Techniczne pojęcie oznaczające przetwarzanie symboli i słów kluczowych w języku programowania to *analiza leksykalna*. Słowo dla jednego z tych symboli lub słów kluczowych jest określane mianem *leksem*.

Słowo kluczowe	
Słowo kluczowe	**Opis**
`auto`	Nadanie lokalnego cyklu życiowego zmiennej lokalnej.
`break`	Opuszczenie polecenia złożonego.
`case`	Gałąź w konstrukcji `switch`.
`char`	Typ danych w postaci znaku.
`const`	Zmienna staje się niemodyfikowalna.

Słowo kluczowe	
Słowo kluczowe	**Opis**
`continue`	Powrót na początek pętli.
`default`	Domyślna gałąź w konstrukcji `switch`.
`do`	Początek pętli `do-while`.
`double`	Typ danych w postaci liczby zmiennoprzecinkowej o podwójnej precyzji.
`else`	Gałąź `else` w konstrukcji `if`.
`enum`	Zdefiniowanie zbioru stałych typu `int`.
`extern`	Zadeklarowanie identyfikatora, który jest zdefiniowany zewnętrznie.
`float`	Typ danych w postaci liczby zmiennoprzecinkowej.
`for`	Początek pętli `for`.
`goto`	Przejście do wskazanej etykiety.
`if`	Początek konstrukcji `if`.
`int`	Typ danych w postaci liczby całkowitej.
`long`	Typ danych w postaci większej liczby całkowitej.
`register`	Zadeklarowanie, że zmienna będzie przechowywana w rejestrze procesora.
`return`	Zwrot wartości z funkcji.
`short`	Typ danych w postaci liczby całkowitej.
`signed`	Typ danych w postaci liczby całkowitej ze znakiem.
`sizeof`	Pozwala na określenie wielkości danych.
`static`	Zachowanie wartości zmiennej po usunięciu jej zakresu.
`struct`	Połączenie zmiennych w pojedynczy rekord.
`switch`	Początek konstrukcji `switch`.
`typedef`	Utworzenie nowego typu.
`union`	Początek konstrukcji `union` (unii).
`unsigned`	Typ danych w postaci liczby całkowitej bez znaku.
`void`	Zadeklarowanie pustego typu danych.
`volatile`	Zadeklarowanie, że zmienna może być modyfikowana wszędzie.
`while`	Początek pętli `while`.

Składnia struktur

Gorąco zachęcam Cię do nauczenia się na pamięć wymienionych powyżej słów kluczowych, a także składni struktur. *Składnia struktury* jest wzorcem symboli tworzących postać kodu źródłowego programu w języku C, na przykład konstrukcji `if` i pętli `while`. Większość tego rodzaju form powinna być Ci znana, ponieważ masz już opanowany przynajmniej jeden inny język programowania. Jedyna trudność polega więc na poznaniu sposobu użycia tych struktur w języku C.

Oto sposób odczytu przedstawionych składni struktur:

1. Wszystko to, co zostało zapisane `WIELKIMI LITERAMI`, jest przeznaczone do zastąpienia.
2. Elementy zapisane [`WIELKIMI LITERAMI W NAWIASIE`] są opcjonalne.
3. Najlepszy sposób na sprawdzenie, jak udało Ci się zapamiętać składnię struktur, polega na tym, że gdy otworzysz edytor tekstu i zobaczysz zwrot typu „konstrukcja…", „polecenie…" itd., spróbuj utworzyć kod po wcześniejszym przeczytaniu opisu przedstawiającego jego działanie.

Konstrukcja `if` to podstawowa struktura pozwalająca na kontrolę logiki programu.

```
if(WARUNEK) {
    KOD;
} else if(WARUNEK) {
    KOD;
} else {
    KOD;
}
```

Konstrukcja `switch` przypomina konstrukcję `if`, ale działa wraz z prostymi stałymi w postaci liczb całkowitych:

```
switch (OPERAND) {
     case STAŁA:
          KOD;
          break;
     default:
          KOD;
}
```

Polecenie `while` tworzy najprostszą postać pętli:

```
while(WARUNEK) {
     KOD;
}
```

W pętli można również użyć polecenia `continue` wznawiającego działanie pętli. Poniższą konstrukcję będziemy na razie określać mianem pętli `while-continue`:

```
while(WARUNEK) {
     if(INNY_WARUNEK) {
          continue;
     }
     KOD;
}
```

Istnieje również możliwość opuszczenia pętli. Poniższą konstrukcję będziemy na razie określać mianem pętli `while-break`:

```
while(WARUNEK) {
     if(INNY_WARUNEK) {
          break;
```

```
        }
        KOD;
}
```

Konstrukcja `do-while` przedstawia odwróconą postać pętli `while`, najpierw jest wykonywany kod, a dopiero *później* sprawdzany jest warunek w celu ustalenia, czy pętla powinna być ponownie wykonana:

```
do {
        KOD;
} while(WARUNEK);
```

Do kontrolowania wykonywania powyższej pętli można również używać słów kluczowych `continue` i `break`.

Konstrukcja `for` to kontrolowana pętla wykonująca ustaloną w liczniku liczbę iteracji:

```
for(INICJALIZACJA; WARUNEK; ZMIANA) {
        KOD;
}
```

Słowo kluczowe `enum` pozwala na utworzenie zbioru stałych w postaci liczb całkowitych:

```
enum { STAŁA1, STAŁA2, STAŁA3 } NAZWA;
```

Polecenie `goto` powoduje przejście do wskazanej etykiety i jest stosowane jedynie w kilku użytecznych sytuacjach, na przykład podczas wykrywania pętli lub opuszczania konstrukcji:

```
if(WARUNEK_BŁĘDU) {
        goto awaria;
}

awaria:
        KOD;
```

Definiowanie funkcji odbywa się następująco:

```
TYP NAZWA(ARGUMENT1, ARGUMENT2, ..) {
        KOD;
        return WARTOŚĆ;
}
```

Ponieważ konstrukcja może być trudna do zapamiętania, wypróbuj poniższy przykład, aby poznać znaczenie `TYP`, `NAZWA`, `ARGUMENT` i `WARTOŚĆ`:

```
int nazwa(argument1, argument2) {
        KOD;
        return 0;
}
```

Słowo kluczowe `typedef` pozwala na zdefiniowanie nowego typu:

```
typedef IDENTYFIKATOR DEFINICJI;
```

Znacznie konkretniejsza postać konstrukcji przedstawia się następująco:

```
typedef unsigned char byte;
```

Niech powyższe polecenie Cię nie zmyli — `unsigned char` to `DEFINICJA`, natomiast `byte` to `IDENTYFIKATOR`.

Tworzona za pomocą słowa kluczowego `struct` struktura pozwala na upakowanie wielu podstawowych typów danych w pojedynczą koncepcję. Struktury są intensywnie wykorzystywane w języku C:

```
struct NAZWA {
     ELEMENTY;
} [NAZWA_ZMIENNEJ];
```

W powyższej strukturze `[NAZWA_ZMIENNEJ]` to element opcjonalny i osobiście proponuję, aby go nie używać, z wyjątkiem kilku sytuacji. Dość często można spotkać połączenie słów kluczowych `typedef` i `struct`, na przykład:

```
typedef struct [NAZWA_STRUKTURY] {
     ELEMENTY;
} IDENTYFIKATOR;
```

Na końcu mamy słowo kluczowe `union` używane do utworzenia konstrukcji przypominającej strukturę, ale o elementach nachodzących się w pamięci. To może być trudne do zrozumienia, więc teraz skoncentruj się jedynie na zapamiętaniu składni:

```
union NAZWA {
     ELEMENTY;
} [NAZWA_ZMIENNEJ];
```

Słowo zachęty

Po przygotowaniu kart zawierających słowa kluczowe i składnię struktur korzystaj z nich w standardowy sposób, czyli najpierw czytaj stronę z nazwą, a później opis na drugiej stronie. W klipie wideo przeznaczonym dla tego ćwiczenia pokazałem, jak użyć oprogramowania Anki do efektywnego przygotowania i stosowania kart. Jednak dokładnie tę samą technikę możesz wykorzystać, przygotowując karty samodzielnie.

Zauważyłem pewną obawę i dyskomfort u osób, które prosiłem o takie właśnie uczenie się na pamięć. Nie jestem pewien, skąd to się bierze. Mimo wszystko zachęcam Cię do zastosowania tej techniki. Potraktuj ją jako możliwość poprawienia własnych umiejętności w zakresie uczenia się. Im więcej będziesz ćwiczył, tym lepszy będziesz się stawał i łatwiej będziesz przyswajał nową wiedzę.

Odczuwanie dyskomfortu i frustracji jest normalne. Być może poświęcisz 15 minut na tego rodzaju ćwiczenie, którego po prostu *nienawidzisz*. To jest zupełnie normalne i wcale nie oznacza, że poniosłeś porażkę. Wytrwałość pomoże Ci pokonać początkową frustrację, a samo ćwiczenie nauczy Cię dwóch rzeczy:

1. Technikę uczenia się na pamięć możesz wykorzystać jako rodzaj samorozwoju prowadzący do poprawy kompetencji. Tylko za pomocą testu pamięci możesz sprawdzić, jak dobrze opanowałeś daną koncepcję.
2. Sposobem na pokonanie trudności jest poruszanie się małymi krokami. Programowanie to doskonały sposób opanowania tej umiejętności, ponieważ zadanie można bardzo łatwo podzielić na małe etapy i skoncentrować się na wykrytych brakach. Wykorzystaj tę możliwość do umocnienia pewności siebie w zakresie podziału dużych zadań na mniejsze etapy.

Słowo ostrzeżenia

Na koniec jeszcze słowo ostrzeżenia dotyczące uczenia się na pamięć. Nauczenie się na pamięć dużej ilości faktów nie oznacza automatycznie, że będziesz mógł je dobrze wykorzystać. Możesz więc nauczyć się na pamięć całego dokumentu opisującego ANSI C i nadal być okropnym programistą. Spotkałem wielu *rzekomych* ekspertów z zakresu C, którzy znali każdy aspekt standardowej gramatyki tego języka, a mimo to nadal tworzyli kod zawierający mnóstwo błędów, dziwaczny, lub nie tworzyli kodu w ogóle.

Nigdy nie myl możliwości nauczenia się na pamięć pewnych faktów z możliwością prawidłowego wykonania zadania. W tym celu musisz zastosować poznane fakty w różnych sytuacjach i wiedzieć, jak je stosować. Pozostała część książki ma Ci w tym pomóc.

ĆWICZENIE 7

Zmienne i typy

Powinieneś już znać strukturę prostego programu w języku C. Przechodzimy więc do kolejnej prostej rzeczy, jaką jest wykorzystanie pewnych zmiennych różnych typów.

Plik *ex7.c*:

```
#include <stdio.h>

int main(intargc, char*argv[])
{
    int distance = 100;
    float power = 2.345f;
    double super_power = 56789.4532;
    char initial = 'A';
    char first_name[] = "Zed";
    char last_name[] = "Shaw";

    printf("Jesteś %d kilometrów stąd.\n", distance);
    printf("Masz %f poziomów mocy.\n", power);
    printf("Masz %f supermocy.\n", super_power);
    printf("Inicjał drugiego imienia to %c.\n", initial);
    printf("Moje imię to %s.\n", first_name);
    printf("Moje nazwisko to %s.\n", last_name);
    printf("Moje pełne imię i nazwisko to %s %c. %s.\n",
        first_name, initial, last_name);

    int bugs = 100;
    double bug_rate = 1.2;

    printf("Masz %d błędów na wyimaginowanym poziomie %f.\n",
        bugs, bug_rate);

    long universe_of_defects = 1L * 1024L * 1024L * 1024L;
    printf("Cały wszechświat zawiera %ld błędów.\n", universe_of_defects);

    double expected_bugs = bugs * bug_rate;
    printf("Powinieneś mieć %f błędów.\n", expected_bugs);

    double part_of_universe = expected_bugs / universe_of_defects;
    printf("To jest jedynie %e błędów we wszechświecie.\n",
        part_of_universe);

    // To jest bez sensu, to jedynie demo czegoś dziwnego.
    char nul_byte = '\0';
    int care_percentage = bugs * nul_byte;
    printf("Powinieneś się więc tym przejmować w %d%%.\n", care_percentage);

    return 0;
}
```

W powyższym programie zadeklarowaliśmy zmienne różnych typów, a następnie wyświetliliśmy ich wartość za pomocą ciągów tekstowych formatowania funkcji `printf()`. Poniżej przedstawiłem dokładne omówienie działania programu.

ex7.c:1 – 4. Standardowy początek programu w języku C.

ex7.c:5 – 7. Zadeklarowanie zmiennych typu `int`, `float` i `double` przeznaczonych dla naszych fikcyjnych obliczeń.

ex7.c:8 – 10. Zadeklarowanie pewnych danych znakowych. Pierwsza jest przeznaczona do przechowywania jednego znaku, kolejne dwie to tablice ciągów tekstowych.

ex7.c:12 – 19. Użycie wywołań funkcji `printf()` do wyświetlenia wartości zadeklarowanych zmiennych.

ex7.c:21 – 22. Zadeklarowanie zmiennych w postaci liczby całkowitej i liczby zmiennoprzecinkowej o podwójnej precyzji. Dzięki temu widzisz, że zmienne nie muszą być deklarowane na początku funkcji.

ex7.c:24 – 25. Wyświetlenie wartości nowych zmiennych — ponownie za pomocą wywołania funkcji `printf()`.

ex7.c:27 – 28. Obliczenie naprawdę wielkiej liczby — całkowitej typu `long`. Zwróć uwagę na użycie notacji `L` do wskazania stałej typu `long` (1L, 1024L).

ex7.c:30 – 35. Wyświetlenie wyniku obliczeń i przeprowadzenie operacji matematycznych na zadeklarowanych zmiennych.

ex7.c:37 – 40. To akurat nie jest najlepsza postać, ale pokazuje przynajmniej, że język C pozwala na użycie zmiennych typu `char` w wyrażeniach matematycznych. Odbywa się to przez pomnożenie wartości `char` przez `int`, a następnie wyświetlany jest otrzymany wynik.

ex7.c:42 – 43. Koniec funkcji `main()`.

Ten plik kodu źródłowego pokazuje jak można przeprowadzać pewne operacje matematyczne z użyciem różnych typów danych i zmiennych. Na końcu programu możesz również poznać funkcjonalność istniejącą w C oraz w kilku innych językach programowania. W języku C *znak* jest po prostu liczbą całkowitą. Wprawdzie to naprawdę bardzo mała liczba całkowita, ale jednak liczba. Masz więc możliwość przeprowadzania operacji matematycznych na zmiennych typu `char`, z czym można spotkać się w wielu programach — na dobre i na złe.

Ostatnia z wymienionych funkcjonalności to jednocześnie Twoje pierwsze zetknięcie z oferowanym przez język C bezpośrednim dostępem do komputera. Do tego tematu jeszcze powrócimy w dalszej części książki.

Co powinieneś zobaczyć?

Jak zwykle poniżej przedstawiam oczekiwane dane wyjściowe.

Sesja dla ćwiczenia 7.:

```
$ make ex7
cc -Wall -g ex7.c -o ex7
$ ./ex7
```

```
Jesteś 100 kilometrów stąd.
Masz 2.345000 poziomów mocy.
Masz 56789.453200 supermocy.
Inicjał drugiego imienia to A.
Moje imię to Zed.
Moje nazwisko to Shaw.
Moje pełne imię i nazwisko to Zed A. Shaw.
Masz 100 błędów na wyimaginowanym poziomie 1.200000.
Cały wszechświat zawiera 1073741824 błędów.
Powinieneś mieć 120.000000 błędów.
To jest jedynie 1.117587e-07 błędów we wszechświecie.
Powinieneś się więc tym przejmować w 0%.
```

Jak to zepsuć?

Przeanalizuj przedstawiony w tym ćwiczeniu program i spróbuj go zepsuć, przekazując wywołaniu funkcji `printf()` nieprawidłowe argumenty. Zobacz, co się stanie, gdy spróbujesz wyświetlić zawartość zmiennej `nul_byte` za pomocą sekwencji `%s` zamiast `%c`. Po zepsuciu programu uruchom go wraz z debugerem i spróbuj określić, co tak naprawdę zepsułeś.

Zadania dodatkowe

- Zmiennej `universe_of_defects` przypisuj różne wielkości aż do chwili, gdy otrzymasz ostrzeżenie od kompilatora.
- Jak faktycznie zostały wyświetlone wielkie liczby?
- Zmień typ zmiennej `universe_of_defects` z `long` na `unsigned long`, a następnie znajdź liczbę, która będzie za duża dla tego typu danych.
- Poszukaj w internecie wiadomości o sposobie działania modyfikatora `unsigned`.

ĆWICZENIE 8

Konstrukcje if, else-if i else

W języku C tak naprawdę nie istnieje typ *boolowski*. Zamiast tego liczba całkowita o wartości `0` jest uznawana za *fałsz*, natomiast wszystkie pozostałe liczby całkowite są uznawane za *prawdę*. W przedstawionym poniżej kodzie źródłowym wyrażenie `argc > 1` w rzeczywistości przyjmuje wartość `1` lub `0`, a nie wyraźnie prawdę (`true`) lub fałsz (`false`), jak ma to miejsce na przykład w Pythonie. To jest kolejny przykład, który pokazuje, że język C działa na warstwie bliższej fizycznemu komputerowi, dla którego wartości `true` to po prostu liczby całkowite.

Jednak w C znajdziemy typową konstrukcję `if`, wykorzystującą ideę numerycznych wartości `true` i `false` do podejmowania decyzji o sposobie wykonywania programu. Działanie tej konstrukcji jest podobne do stosowanych w językach na przykład Python i Ruby, jak możesz zobaczyć w poniższym fragmencie kodu.

Plik *ex8.c*:

```
 1 #include <stdio.h>
 2
 3 int main(int argc, char *argv[])
 4 {
 5     int i = 0;
 6
 7     if (argc == 1) {
 8         printf("Masz tylko jeden argument. Trudno.\n");
 9     } else if (argc > 1 && argc < 4) {
10         printf("Oto argumenty:\n");
11
12         for (i = 0; i < argc; i++) {
13             printf("%s ", argv[i]);
14         }
15         printf("\n");
16     } else {
17         printf("Podałeś zbyt wiele argumentów. Zawiodłeś.\n");
18     }
19
20     return 0;
21 }
```

Składnia konstrukcji `if` przedstawia się następująco:

```
if(WARUNEK) {
    KOD;
} else if(WARUNEK) {
    KOD;
} else {
    KOD;
}
```

Konstrukcja ta jest podobna do spotykanych w większości innych języków programowania, choć z pewnymi różnicami charakterystycznymi dla C.

- Jak wcześniej wspomniałem, `WARUNEK` staje się fałszem, gdy przyjmuje wartość `0`. W pozostałych przypadkach jest prawdą.
- Elementy `WARUNKU` trzeba umieścić w nawiasie, choć w niektórych językach programowania można pominąć ten nawias.
- Nie ma konieczności użycia nawiasów klamrowych do ujęcia kodu, choć ich pomijanie to *bardzo* zła praktyka programistyczna. Nawiasy klamrowe wyraźnie wskazują początek i koniec danej gałęzi kodu. Jeżeli z nich zrezygnujesz, w kodzie mogą pojawiać się dziwne błędy.

Poza wymienionymi aspektami kod działa dokładnie w taki sam sposób, jak w innych językach programowania. Nie ma konieczności użycia bloków `else if` lub `else`.

Co powinieneś zobaczyć?

To jest całkiem prosty program do uruchomienia i wypróbowania.

Sesja dla ćwiczenia 8.:

```
$ make ex8
cc -Wall -g ex8.c -o ex8
$ ./ex8
Podałeś tylko jeden argument. Zawiodłeś.
$ ./ex8 jeden
Oto argumenty:
./ex8 jeden
$ ./ex8 jeden dwa
Oto argumenty:
./ex8 jeden dwa
$ ./ex8 jeden dwa trzy
Podałeś zbyt wiele argumentów. Zawiodłeś.
$
```

Jak to zepsuć?

To nie jest najłatwiejszy kod do zepsucia, co wynika z jego prostoty. Spróbuj jednak nieco namieszać w sprawdzanym warunku polecenia `if`.

- Usuń polecenie `else` na końcu; program nie uwzględni przypadku skrajnego.
- Zmień operator `&&` na `||`, aby użyć logicznego `LUB` zamiast `I`, a następnie sprawdź działanie tak zmodyfikowanego kodu.

Zadania dodatkowe

- Pokrótce poznałeś działanie operatora `&&` odpowiedzialnego za przeprowadzenie porównania. Poszukaj w internecie informacji o innych operatorach boolowskich.
- Utwórz kilka dodatkowych przypadków w programie i sprawdź, jak działają.
- Czy wynik pierwszego testu naprawdę jest poprawny. Dla Ciebie *pierwszy argument* nie jest taki sam, jak pierwszy argument wprowadzony przez użytkownika. Popraw kod źródłowy.

ĆWICZENIE 9

Pętla while i wyrażenia boolowskie

Pierwsza konstrukcja pętli, którą poznasz, to `while`. Zalicza się ona do najprostszych i użytecznych pętli w języku C. Poniżej przedstawiłem przykładowy kod źródłowy wykorzystujący pętlę `while`.

Plik *ex9.c*:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i = 0;
    while (i < 25) {
        printf("%d", i);
        i++;
    }

    // Poniższe polecenie dodaje znak nowego wiersza na końcu.
    printf("\n");

    return 0;
}
```

Na podstawie powyższego fragmentu kodu oraz tego, czego nauczyłeś się podczas zapamiętywania podstawowej składni, możemy stwierdzić, że konstrukcja pętli `while` przedstawia się następująco:

```
while(WARUNEK) {
    KOD;
}
```

Powyższa pętla po prostu wykonuje `KOD`, dopóki `WARUNEK` jest prawdą (`1`). Porównując tę pętlę z działaniem pętli `for`, możemy powiedzieć, że potrzebny jest sposób na inicjalizację i inkrementację zmiennej `i`. Nie zapominaj, że polecenie `i++` inkrementuje zmienną `i` za pomocą operatora postinkrementacji. Jeżeli nie rozpoznajesz operatora, powróć do przygotowanych wcześniej kart.

Co powinieneś zobaczyć?

Dane wyjściowe są praktycznie takie same, a więc wprowadziłem niewielką zmianę, aby pokazać możliwość uruchomienia programu w nieco inny sposób.

Sesja dla ćwiczenia 9.:

```
$ make ex9
cc -Wall -g ex9.c -o ex9
$ ./ex9
01234567891011121314151617181920212223 24
$
```

Jak to zepsuć?

Istnieje kilka powodów, dla których działanie pętli `while` może być inne od oczekiwanego. Dlatego też nie zalecam jej użycia, o ile nie zachodzi konieczność. Poniżej wymieniłem kilka łatwych sposobów na zepsucie pętli `while`.

- Zapomnij o inicjalizacji pierwszej zmiennej `int i;`. W zależności od wartości początkowej `i` pętla w ogóle może nie zostać wykonana lub jej działanie może trwać wyjątkowo długo.
- Zapomnij o inkrementacji `i++` na końcu pętli — otrzymasz *pętlę działającą w nieskończoność*. To jest jeden z koszmarnych problemów, powszechnych w dwóch pierwszych dekadach programowania.

Zadania dodatkowe

- Zmodyfikuj pętlę w taki sposób, aby odliczanie następowało wstecz. Użyj operatora `i--`; pętla powinna odliczać od `25` do `0`.
- Utwórz kilka nieco bardziej skomplikowanych pętli `while`, wykorzystując zdobytą dotąd wiedzę.

ĆWICZENIE 10

Konstrukcja switch

W innych językach programowania, na przykład w Ruby, przygotowujesz konstrukcję `switch`, która może pobrać dowolne wyrażenie. Niektóre języki, takie jak Python, nie mają konstrukcji `switch`, ponieważ polecenie `if` wraz z wyrażeniem boolowskim wykonuje to samo zadanie. W wymienionych językach programowania konstrukcja `switch` może być uznana za alternatywę dla poleceń `if` i wewnętrznie działa dokładnie tak samo.

Z kolei w języku C działanie konstrukcji `switch` jest zupełnie inne i tak naprawdę tworzy *tabelę przeskoków*. Zamiast dowolnie wybranego wyrażenia boolowskiego możesz użyć jedynie tych wyrażeń, których wartością zwrotną jest liczba całkowita. Wspomniana liczba całkowita jest następnie wykorzystywana do obliczenia przeskoku od początku konstrukcji `switch` do bloku zawierającego dopasowaną `wartość`. Poniżej przedstawiłem kod źródłowy pewnego programu w C, który powinien pomóc Ci w zrozumieniu koncepcji tabeli przeskoku.

Plik *ex10.c*:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    if (argc != 2) {
        printf("BŁĄD: Należy podać jeden argument.\n");
        // Poniżej pokazałem polecenie pozwalające na zakończenie działania programu.
        return 1;
    }

    int i = 0;
    for (i = 0; argv[1][i] != '\0'; i++) {
        char letter = argv[1][i];

        switch (letter) {
            case 'a':
            case 'A':
                printf("%d: 'A'\n", i);
                break;

            case 'e':
            case 'E':
                printf("%d: 'E'\n", i);
                break;

            case 'i':
            case 'I':
                printf("%d: 'I'\n", i);
                break;

            case 'o':
```

```
32              case 'O':
33                  printf("%d: 'O'\n", i);
34                  break;
35
36              case 'u':
37              case 'U':
38                  printf("%d: 'U'\n", i);
39                  break;
40
41              case 'y':
42              case 'Y':
43                  if (i > 2) {
44                      // Tylko czasami to będzie Y.
45                      printf("%d: 'Y'\n", i);
46                  }
47                  break;
48
49              default:
50                  printf("%d: %c nie jest samogłoską.\n", i, letter);
51          }
52      }
53
54      return 0;
55  }
```

W powyższym programie pobieramy pojedynczy argument w powłoce, a następnie wyświetlamy wszystkie samogłoski w niezwykle żmudny sposób, demonstrując tym samym przykład zastosowania konstrukcji `switch`. Poniżej przedstawiłem objaśnienie działania tej konstrukcji.

- Kompilator oznacza miejsce w programie, gdzie rozpoczyna się działanie konstrukcji `switch`. Przyjmujemy założenie, że to miejsce nazywamy `Y`.
- Obliczenie wartości wyrażenia `switch(letter)`, aby mogła być wyrażona za pomocą liczby. W omawianym przykładzie będzie to niezmodyfikowany kod ASCII litery wskazywanej przez element `argv[1]`.
- Kompilator przetwarza również wszystkie bloki `case`, na przykład `'A':`, uwzględniając przy tym aktualne położenie programu. Dlatego też w przypadku bloku `case 'A'` położenie w programie możemy określić jako `Y+A`.
- Następnie przeprowadzana jest operacja matematyczna w celu ustalenia, czy przypadek `Y + litera` został uwzględniony w konstrukcji `switch`. Jeżeli działanie programu przeszło zbyt daleko, mamy do czynienia z przypadkiem `Y + default`.
- Po ustaleniu położenia program *przeskakuje* do tego miejsca w kodzie i kontynuuje działanie. Dlatego też w niektórych blokach `case` konieczne jest użycie polecenia `break`, natomiast w innych już nie.
- Po wpisaniu litery `a` nastąpi przejście do `case 'a'`. Ponieważ ten blok nie ma polecenia `break`, działanie jest kontynuowane w bloku `case 'A'`, którego kod zawiera polecenie `break`.
- Po wykonaniu kodu następuje wywołanie polecenia `break` i całkowite opuszczenie konstrukcji `switch`.

Wprawdzie powyżej przedstawiłem dość dokładne wyjaśnienie działania konstrukcji `switch`, ale w praktyce powinieneś pamiętać o kilku prostych regułach.

- Zawsze definiuj blok `default:` przeznaczony do obsługi danych wejściowych nieobsłużonych przez wcześniejsze bloki `case`.
- Nie pozwalaj na przejście do kolejnego bloku `case`, o ile naprawdę tego nie chcesz. Dobrym podejściem jest umieszczanie komentarza na przykład `// Celowe przejście.`, aby poinformować innych o zamierzonym przejściu do kolejnego bloku `case`.
- Zawsze najpierw wpisz polecenia `case` i `break`, a dopiero później kod danego bloku.
- Jeżeli istnieje taka możliwość, spróbuj użyć konstrukcji `if` zamiast `switch`.

Co powinieneś zobaczyć?

Poniżej przedstawiłem zapis przykładowej sesji pracy z omawianym programem oraz różne sposoby przekazywania argumentu.

Sesja dla ćwiczenia 10.:

```
$ make ex10
cc -Wall -gex10.c -o ex10
$ ./ex10
BŁĄD: Należy podać jeden argument.
$
$ ./ex10 Zed
0: Z nie jest samogłoską.
1: 'E'
2: d nie jest samogłoską.
$
$ ./ex10 Zed Shaw
BŁĄD: Należy podać jeden argument.
$
$ ./ex10 "Zed Shaw"
0: Z nie jest samogłoską.
1: 'E'
2: d nie jest samogłoską.
3:   nie jest samogłoską.
4: S nie jest samogłoską.
5: h nie jest samogłoską.
6: 'A'
7: w nie jest samogłoską.
$
```

Nie zapominaj o istnieniu na początku kodu źródłowego konstrukcji `if`, pozwalającej na zakończenie działania programu (polecenie `return 1;`) w przypadku podania nieprawidłowej liczby argumentów. Wartość zwrotna inna niż `0` wskazuje systemowi operacyjnemu na wystąpienie błędu w programie. Istnieje możliwość sprawdzenia w skryptach oraz w innych programach, czy wartość zwrotna jest większa niż `0`, co z kolei pomaga w ustaleniu faktycznego przebiegu zdarzeń.

Jak to zepsuć?

Zepsucie konstrukcji `switch` jest *niewiarygodnie* łatwym zadaniem. Poniżej wymieniłem kilka przykładów, jak możesz namieszać w omawianym programie.

- Zapomnij dodać polecenie `break` — wówczas program wykona dwa lub więcej bloków kodu, czego oczywiście nie chcesz.
- Zapomnij dodać blok `default` — bez jakiegokolwiek komunikatu zostaną zignorowane wartości niedopasowane do istniejących bloków `case`.
- W konstrukcji `switch` przypadkowo umieść zmienną przyjmującą wartość inną niż oczekiwana, na przykład typu `int`, która otrzyma zupełnie niepoprawną wartość.
- Użyj niezainicjalizowanych wartości w konstrukcji `switch`.

Działanie programu można zepsuć także na wiele innych sposobów. Zobacz, czy potrafisz to zrobić.

Zadania dodatkowe

- Utwórz inny program wykorzystujący operacje matematyczne w celu konwersji litery na postać zapisaną małą literą, a następnie za pomocą konstrukcji `switch` usuń wszystkie wielkie litery.
- Wykorzystaj przecinek do inicjalizacji zmiennej `letter` w pętli `for`.
- Upewnij się o obsłużeniu za pomocą innej pętli `for` wszystkich argumentów przekazanych programowi.
- Skonwertuj omawianą konstrukcję `switch` na postać konstrukcji `if`. Która z wersji według Ciebie jest lepsza?
- W przypadku `'Y'` konieczne jest przerwanie działania na zewnątrz konstrukcji `if`. Jaki to ma wpływ na rozwiązanie i co się stanie, jeżeli operację tę przeniesiesz do wewnątrz konstrukcji `if`? Udowodnij sobie, że masz rację.

ĆWICZENIE 11

Tablice i ciągi tekstowe

W tym ćwiczeniu pokażę, że w języku C ciągi tekstowe są przechowywane po prostu w postaci tablicy bajtów i zakończone bajtem NULL, czyli `'\0'`. Prawdopodobnie domyśliłeś się już tego w poprzednim ćwiczeniu, gdy na końcu ciągu tekstowego ręcznie dodaliśmy niezbędny bajt. Poniżej pokazałem dodanie wspomnianego bajta w inny sposób, znacznie przejrzystszy, dzięki wykorzystaniu tablicy liczb.

Plik *ex11.c*:

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5     int numbers[4] = { 0 };
6     char name[4] = { 'a' };
7
8     // Najpierw wyświetlamy niezmodyfikowane liczby.
9     printf("liczby: %d %d %d %d\n",
10         numbers[0], numbers[1], numbers[2], numbers[3]);
11
12     printf("imię (oddzielnie): %c %c %c %c\n",
13         name[0], name[1], name[2], name[3]);
14
15     printf("imię: %s\n", name);
16
17     // Przygotowanie liczb.
18     numbers[0] = 1;
19     numbers[1] = 2;
20     numbers[2] = 3;
21     numbers[3] = 4;
22
23     // Przygotowanie imienia.
24     name[0] = 'Z';
25     name[1] = 'e';
26     name[2] = 'd';
27     name[3] = '\0';
28
29     // Wyświetlenie liczb po zainicjalizowaniu.
30     printf("liczby: %d %d %d %d\n",
31         numbers[0], numbers[1], numbers[2], numbers[3]);
32
33     printf("imię (oddzielne litery): %c %c %c %c\n",
34         name[0], name[1], name[2], name[3]);
35
36     // Wyświetlenie imienia w postaci ciągu tekstowego.
37     printf("imię: %s\n", name);
38
39     // Inny sposób na użycie imienia.
```

```
40     char *another = "Zed";
41
42     printf("inny: %s\n", another);
43
44     printf("inny (oddzielne litery): %c %c %c %c\n",
45         another[0], another[1], another[2], another[3]);
46
47     return 0;
48 }
```

W powyższym fragmencie kodu przygotowaliśmy pewne tablice w żmudny sposób przez przypisanie wartości poszczególnym elementom. Tablica `numbers` zawiera liczby, natomiast w tablicy `name` ręcznie tworzymy ciąg tekstowy.

Co powinieneś zobaczyć?

Po uruchomieniu powyższego fragmentu kodu powinieneś zobaczyć wyświetloną zawartość tablicy najpierw zainicjalizowaną z wartością zero, a później z inną.

Sesja dla ćwiczenia 11.:

```
$ make ex11
cc -Wall -g ex11.c -o ex11
$ ./ex11
liczby: 0 0 0 0
imię (oddzielne litery): a
imię: a
liczby: 1 2 3 4
imię (oddzielne litery): Z e d
imię: Zed
inny: Zed
inny (oddzielne litery): Z e d
$
```

Oto kilka interesujących spostrzeżeń dotyczących omawianego programu.

- Nie podałem wszystkich czterech elementów tablic podczas ich inicjalizacji. W języku C istnieje pewien użyteczny skrót. Jeżeli zdefiniujesz przynajmniej jeden element tablicy, pozostałe będą zainicjalizowane wraz z wartością `0`.
- Podczas wyświetlania zawartości tablicy `numbers` wszystkie elementy to `0`.
- Podczas wyświetlania elementów tablicy `name` dane wyjściowe zawierają tylko pierwszy element `'a'`, ponieważ znak `'\0'` zalicza się do znaków specjalnych i nie jest wyświetlany.
- W trakcie pierwszego wyświetlenia zawartości tablicy `name` dane wyjściowe zawierają tylko literę `a`. Wynika to z faktu wypełnienia tablicy zerami po podaniu pierwszego elementu `'a'` w trakcie inicjalizacji tablicy, a więc ciąg tekstowy jest prawidłowo zakończony znakiem `'\0'`.

- Kolejnym krokiem jest przygotowanie tablic z wykorzystaniem żmudnego, ręcznego przypisywania wartości poszczególnym elementom, a następnie wyświetlenie zawartości tych tablic. Zobacz, w jaki sposób się zmieniły. Teraz liczby są zdefiniowane, ale czy zawartość ciągu tekstowego `name` to poprawnie zapisane moje imię?
- Mamy dwie składnie pozwalające na utworzenie ciągu tekstowego. Pierwsza to `char name[4] = {'a'}` zastosowana w wierszu 6., natomiast druga to `char *another = "name"`, użyta w wierszu 44. Pierwsza z wymienionych jest rzadziej spotykana, natomiast drugą powinieneś wykorzystywać do pracy z literałami znakowymi, jak ma to miejsce w omawianym przykładzie.

Zwróć uwagę na użycie tej samej składni i tego samego stylu kodu do pracy z tablicą zarówno liczb całkowitych, jak i znaków. Jednak funkcja `printf()` traktuje zawartość `name` jako ciąg tekstowy. Wynika to z faktu, o którym warto wspomnieć ponownie: język C nie odróżnia ciągu tekstowego od tablicy znaków.

Kiedy tworzysz literały znakowe, powinieneś zastosować składnię `char *another = "Literał"`. Wprawdzie otrzymany efekt jest dokładnie taki sam, ale to podejście jest bardziej idiomatyczne i łatwiejsze w zapisie.

Jak to zepsuć?

Źródłem niemalże wszystkich błędów w języku C jest zarezerwowanie niewystarczającej ilości miejsca lub nieumieszczenie znaku `'\0'` na końcu ciągu tekstowego. Tak naprawdę trudno jest uniknąć ciągów tekstowych stylu C w większości dobrego kodu źródłowego w C. W dalszej części książki dowiesz się, jak całkowicie unikać ciągów tekstowych C.

Omawiany program można bardzo łatwo zepsuć, po prostu zapominając o umieszczeniu znaku `'\0'` na końcu ciągu tekstowego. Istnieje na to kilka sposobów.

- Pozbycie się inicjalizatora tworzącego `name`.
- Przypadkowe przypisanie `name[3] = 'A';`, co oznacza brak znaku `'\0'` na końcu ciągu tekstowego.
- Ustawienie inicjalizatora jako `{'a','a','a','a'}`, co oznacza zbyt wiele znaków `'a'` i brak miejsca na umieszczenie znaku `'\0'` na końcu ciągu tekstowego.

Spróbuj znaleźć kilka innych sposobów zepsucia omawianego programu. Za każdym razem uruchamiaj program wraz z debugerem, aby dokładnie zobaczyć, co się dzieje i jakie zostały wygenerowane błędy. Czasami popełniasz błędy, których debuger nie jest w stanie wychwycić. Spróbuj pomajstrować przy deklaracji zmiennych i sprawdź, czy to spowoduje wygenerowanie błędu. Oto jeden z aspektów voodoo w języku C: czasami miejsce położenia zmiennej może mieć wpływ na zmianę typu błędu.

Zadania dodatkowe

- Przypisz znaki tablicy `numbers`, a następnie za pomocą wywołania `printf()` wyświetl po jednym znaku jednorazowo. Jakiego rodzaju ostrzeżenie kompilatora zostało wygenerowane?
- Wykonaj odwrotne zadanie względem tablicy `name` — spróbuj ją potraktować jako tablicę liczb całkowitych i wyświetl po jednej liczbie za każdym razem. Jakie pojawiły się komunikaty wygenerowane przez debuger?
- Na ile innych sposobów można wygenerować dane wyjściowe?
- Jeżeli wielkość tablicy znaków wynosi 4 bajty, a liczba całkowita ma 4 bajty długości, to czy wobec tego całą tablicę `name` można potraktować jak po prostu liczbę całkowitą? W jaki sposób można zrealizować tę szaloną sztuczkę?
- Weź kartkę papieru i narysuj wymienione tablice jako rząd kwadratów. Następnie operacje wykonane wcześniej w komputerze rozrysuj na papierze i sprawdź, czy zrobiłeś to prawidłowo.
- Skonwertuj tablicę `name` na styl użyty do zadeklarowania tablicy `another` i sprawdź, czy kod nadal działa prawidłowo.

ĆWICZENIE 12

Wielkość i tablice

W poprzednim ćwiczeniu przeprowadzaliśmy operacje matematyczne, ale z użyciem znaku '\0'. Może się to wydawać dziwne osobom mającym doświadczenie w programowaniu w innym języku, ponieważ próbują one potraktować *ciągi tekstowe* i *tablice bajtów* jako zupełnie odmienne elementy. Język C traktuje ciągi tekstowe jako po prostu tablice bajtów, więc różnica może być wychwycona jedynie przez funkcje wyświetlające zawartość wymienionych elementów.

Zanim przystąpię do wyjaśnienia powyższej koncepcji, najpierw chciałbym wyjaśnić dwie dodatkowe kwestie — wielkość elementu i tablice. Poniżej przedstawiam kod źródłowy, który za chwilę omówimy.

Plik *ex12.c*:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int areas[] = { 10, 12, 13, 14, 20 };
    char name[] = "Zed";
    char full_name[] = {
        'Z', 'e', 'd',
        ' ', 'A', '.', ' ',
        'S', 'h', 'a', 'w', '\0'
    };

    // OSTRZEŻENIE: W niektórych systemach może wystąpić konieczność zmiany w tym
    // kodzie sekwencji %ld na %u, ponieważ są używane liczby całkowite bez znaków.
    printf("Wielkość liczby całkowitej typu int: %ld\n", sizeof(int));
    printf("Wielkość tablicy areas (int[]): %ld\n", sizeof(areas));
    printf("Liczba elementów int w tablicy areas: %ld\n",
        sizeof(areas) / sizeof(int));
    printf("Pierwszy element to %d, drugi to %d.\n", areas[0], areas[1]);

    printf("Wielkość znaku char: %ld\n", sizeof(char));
    printf("Wielkość tablicy name (char[]): %ld\n", sizeof(name));
    printf("Liczba znaków: %ld\n", sizeof(name) / sizeof(char));

    printf("Wielkość tablicy full_name (char[]): %ld\n", sizeof(full_name));
    printf("Liczba znaków: %ld\n",
        sizeof(full_name) / sizeof(char));

    printf("Wartość name=\"%s\", natomiast full_name=\"%s\"\n", name, full_name);

    return 0;
}
```

W powyższym kodzie tworzymy kilka tablic wraz z różnymi typami danych. Ponieważ tablice danych mają tak ważne znaczenie dla działania C, więc istnieje ogromna liczba sposobów, na jakie można je tworzyć. W tym momencie pozostaniemy przy składni `typ nazwa[] = {ini ↪cjalizator};`, a kolejne poznasz w dalszej części książki. Znaczenie przedstawionej składni jest następujące: „Chcę, aby tablica podanego typu została zainicjalizowana wraz z zawartością {…}". Po napotkaniu tego polecenia kompilator C podejmuje następujące kroki:

- Sprawdza typ tablicy — w omawianym przykładzie to `int`.
- Analizuje podany nawias [] i ustala, że nie podano wielkości tablicy.
- Sprawdza inicjalizator `{10, 12, 13, 14, 20}` i ustala, że w tablicy ma zostać umieszczonych pięć liczb całkowitych.
- W pamięci komputera rezerwuje miejsce przeznaczone do przechowywania pięciu liczb całkowitych, jedna po drugiej.
- Pobiera podaną nazwę (tutaj `areas`) i przypisuje jej wspomniany adres w pamięci.

W przypadku zmiennej `areas` następuje utworzenie tablicy pięciu liczb całkowitych przechowującej podane liczby. Polecenie `char name[] = "Zed";` wykonuje to samo zadanie, ale tworzy tablicę trzech znaków i przypisuje ją do nazwy `name`. Ostatnia tablica tworzona w omawianym kodzie to `full_name`, ale w jej przypadku wykorzystujemy irytującą składnię polegającą na podawaniu jednorazowo po jednym znaku. Z perspektywy języka C, `name` i `full_name` to identyczne metody tworzenia tablicy znaków.

W pozostałej części pliku wykorzystujemy słowo kluczowe `sizeof` nakazujące językowi C sprawdzenie wielkości elementu wyrażonej w *bajtach*. W języku C wiele wykonywanych operacji wiąże się z wielkością elementu i położeniem fragmentów pamięci. Aby nieco ułatwić pracę programiście, język udostępnia słowo kluczowe `sizeof`, za pomocą którego można sprawdzić wielkość elementu przed przystąpieniem do pracy z nim.

W tym miejscu sprawy się nieco komplikują, więc najlepiej najpierw uruchomić kod, a dopiero później zapoznać się z objaśnieniem jego działania.

Co powinieneś zobaczyć?

Sesja dla ćwiczenia 12.:

```
 1 $ make ex12
 2 cc -Wall -g ex12.c -o ex12
 3 $ ./ex12
 4 Wielkość liczby całkowitej typu int: 4
 5 Wielkość tablicy areas (int[]): 20
 6 Liczba elementów int w tablicy areas: 5
 7 Pierwszy element to 10, drugi to 12.
 8 Wielkość znaku char: 1
 9 Wielkość tablicy name (char[]): 4
10 Liczba znaków: 4
11 Wielkość tablicy full_name (char[]): 12
```

```
12 Liczba znaków: 12
13 Wartość name="Zed", natomiast full_name="Zed A. Shaw"
14 $
```

Powyżej przedstawiłem dane wyjściowe różnych wywołań funkcji `printf()` i możesz wreszcie zobaczyć, co tak naprawdę C tutaj robi. Muszę w tym miejscu dodać, że otrzymane przez Ciebie dane wyjściowe mogą być zupełnie inne, ponieważ używany komputer może mieć całkiem inną wielkość liczb całkowitych. Poniżej omawiam otrzymane przeze mnie dane wyjściowe.

4. Według mojego komputera wielkość typu `int` wynosi 4 bajty. Twój komputer może stosować inną wielkość — to zależy od wbudowanego procesora (32-bitowy lub 64-bitowy).
5. Tablica `areas` zawiera pięć liczb całkowitych, a więc sensowne jest, że mój komputer wymaga 20 bajtów do przechowywania jej zawartości.
6. Jeżeli wielkość tablicy `areas` podzielimy przez wielkość liczby całkowitej typu `int`, to otrzymamy pięć elementów. Spójrz na kod — odpowiada to ilości liczb podanych w inicjalizatorze.
7. W trakcie operacji uzyskania dostępu do tablicy `areas` używamy poleceń `areas[0]` i `areas[1]`, ponieważ podobnie jak w językach Python i Ruby, tablice mają *indeksy numerowane od zera*.

8 – 10. Tę samą operację powtarzamy dla tablicy `name`. Czy zauważyłeś coś dziwnego w kwestii wielkości tej tablicy? Komunikat sugeruje `4` bajty, podczas gdy podane imię `Zed` składa się z jedynie trzech znaków. Skąd więc wziął się czwarty?

11 – 12. Tę samą operację powtarzamy również dla tablicy `full_name` i widzimy, że jej wielkość została podana prawidłowo.

13. Wyświetlamy zawartość tablic `name` i `full_name`, aby udowodnić, że faktycznie są „ciągami tekstowymi”, jak na to wskazuje funkcja `printf()`.

Upewnij się o dokładnym przeanalizowaniu otrzymanych danych wyjściowych i zrozumieniu, jak zostały wygenerowane. Ta wiedza będzie podstawą do dalszego omawiania tematu tablic i przechowywania danych.

Jak to zepsuć?

Zepsucie omówionego programu nie należy do trudnych zadań. Wypróbuj dowolny z poniższych sposobów.

- Pozbądź się znaku `'\0'` na końcu tablicy `full_name` i zwróć ją. Uruchom program również za pomocą debugera. Następnie przenieś definicję tablicy `full_name` na początek funkcji `main()`, jeszcze przed poleceniem definiującym tablicę `areas`. Spróbuj kilkakrotnie uruchomić program w debugerze i zobaczyć, czy zostaną wygenerowane jakiekolwiek nowe błędy. Czasami może Ci się poszczęścić i nie wystąpią żadne błędy.

- Zmień wywołanie `areas[0]` na `areas[10]`. Sprawdź, jakie komunikaty zostaną wygenerowane przez debuger.
- Wypróbuj jeszcze inne, podobne sposoby zepsucia programu, modyfikując przy tym również tablice `name` i `full_name`.

Zadania dodatkowe

- Spróbuj do tablicy `areas` dodać kolejne elementy za pomocą składni `areas[0] = 100;` i podobnej.
- Spróbuj dodać nowe elementy do tablic `name` i `full_name`.
- Spróbuj zmienić typ jednego elementu tablicy `areas` na znak pochodzący z tablicy `name`.
- Wyszukaj w internecie informacje na temat różnej wielkości liczb całkowitych w zależności od zastosowanego procesora.

ĆWICZENIE 13

Pętla for i tablica ciągów tekstowych

Możesz tworzyć tablice różnych typów, ale pamiętaj, że ciąg tekstowy i tablica bajtów oznacza dokładnie to samo. Naszym kolejnym krokiem jest przygotowanie tablicy zawierającej ciągi tekstowe. Ponadto poznasz pierwszą konstrukcję pętli, `for`, pomagającą w wyświetleniu zawartości tej nowej struktury danych.

Najzabawniejsze jest to, że tablice ciągów tekstowych pojawiały się już w prezentowanych wcześniej programach, choć pozostawały ukryte, na przykład `char *argv[]` w argumentach funkcji `main()`. Poniżej przedstawiam program wyświetlający wszystkie argumenty przekazane mu w powłoce.

Plik *ex13.c*:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i = 0;

    // Przejście przez wszystkie ciągi tekstowe tablicy argv.
    // Dlaczego pomijamy argv[0]?
    for (i = 1; i < argc; i++) {
        printf("argument %d: %s\n", i, argv[i]);
    }

    // Tworzymy własną tablicę ciągów tekstowych.
    char *states[] = {
        "Kalifornia", "Oregon",
        "Washington", "Texas"
    };

    int num_states = 4;

    for (i = 0; i < num_states; i++) {
        printf("stan %d: %s\n", i, states[i]);
    }

    return 0;
}
```

Składnia pętli `for` przedstawia się następująco:

```
for(INICJALIZACJA; WARUNEK; ZMIANA) {
    KOD;
}
```

Oto sposób działania pętli `for`:

- `INICJALIZACJA` to kod wykonywany w celu skonfigurowania pętli. W omawianym przykładzie to po prostu polecenie `i = 0`.
- Następnie mamy `WARUNEK`, czyli sprawdzenie wyrażenia boolowskiego. Jeżeli przyjmie wartość fałsz (`0`), to blok `KOD` zostanie pominięty i pętla nie podejmie żadnych działań.
- Wykonanie bloku `KOD`.
- Po zakończeniu wykonywania bloku `KOD` następuje wykonanie kodu odpowiedzialnego za przeprowadzenie `ZMIANY`, co zwykle oznacza inkrementację pewnego elementu, na przykład `i++`.
- Działanie pętli jest ponownie kontynuowane od kroku 2. i powtarzane, dopóki wyrażenie `WARUNEK` nie przyjmie wartości `0`.

Wykorzystana w omawianym programie pętla `for` przeprowadza iterację przez argumenty powłoki i w następujący sposób używa zmiennej `argc` i tablicy `argv`:

- System operacyjny przekazuje argument powłoki jako ciąg tekstowy w tablicy `argv`. Nazwa programu (tutaj *./ex13*) to argument o indeksie `0`, w kolejnych indeksach są umieszczone pozostałe argumenty.
- System operacyjny przypisuje zmiennej `argc` liczbę argumentów w tablicy `argv`, co pozwala na ich przetworzenie bez obaw o próbę uzyskania dostępu do nieistniejącego elementu tablicy. Pamiętaj, że jeśli podasz tylko jeden argument, to nazwa programu będzie pierwszym elementem tablicy, a wartość zmiennej `argc` będzie wynosić `2`.
- Kodem inicjalizującym pętlę `for` jest przypisanie `i = 1`.
- Następnie za pomocą `i < argc` sprawdzamy, czy wartość `i` jest mniejsza niż podana w `argc`. Ponieważ `$1 < $2`, więc działanie pętli jest kontynuowane.
- Kolejny krok to wykonanie kodu wyświetlającego wartość `i` oraz używającego zmiennej `i` w celu uzyskania dostępu do elementu tablicy `argv`.
- Inkrementacja następuje po wykonaniu polecenia `i++`, które jest wygodnym skrótem dla polecenia `i = i + 1`.
- Procedura jest powtarzana do chwili, aż warunek `i < argc` przyjmie wartość fałsz (`0`). Wówczas działanie pętli zostanie zakończone, a program przejdzie do pierwszego polecenia znajdującego się za pętlą.

Co powinieneś zobaczyć?

Aby poeksperymentować z omawianym programem, powinieneś uruchomić go na dwa sposoby. Pierwszy to przekazanie pewnych argumentów powłoki, co spowoduje przypisanie wartości zmiennej `argc` i tablicy `argv`. Drugi to uruchomienie bez argumentów, co pokaże, że pierwsza pętla `for` nie jest wykonywana, gdy warunek `i < argc` przyjmuje wartość fałsz.

Sesja dla ćwiczenia 13.:

```
$ make ex13
cc -Wall -g ex13.c -o ex13
$ ./ex13 to jest całkiem spora liczba argumentów
argument 1: to
argument 2: jest
argument 3: całkiem
argument 4: spora
argument 5: liczba
argument 6: argumentów
stan 0: Kalifornia
stan 1: Oregon
stan 2: Waszyngton
stan 3: Teksas
$
$ ./ex13
stan 0: Kalifornia
stan 1: Oregon
stan 2: Waszyngton
stan 3: Teksas
$
```

Zrozumienie tablicy ciągów tekstowych

W języku C tworzysz tablicę *ciągów tekstowych* przez połączenie składni `char *str = "blah"` wraz ze składnią `char str[] = {'b','l','a','h'}`, co powoduje powstanie tablicy dwuwymiarowej. Składnia `char *states[] = {...}` w wierszu 14. tworzy tego rodzaju tablicę dwuwymiarową, w której każdy ciąg tekstowy to jeden element, natomiast poszczególne znaki w ciągu tekstowym to drugi.

Czy to wydaje się niezrozumiałe? Koncepcja wielu wymiarów to coś, nad czym większość osób nigdy się nie zastanawia, a więc powinieneś spróbować utworzyć tę tablicę ciągów tekstowych na papierze:

- Narysuj siatkę, po jej lewej stronie umieść indeksy każdego *ciągu tekstowego*.
- Indeksy poszczególnych *znaków* umieść na górze.
- Wypełnij pola pośrodku, przy czym wpisuj tylko jeden znak do każdego pola.
- Po przygotowaniu siatki przeanalizuj omawiany tutaj kod, posługując się tą siatką.

Innym sposobem pomagającym w zrozumieniu tej koncepcji może być zbudowanie tej samej struktury w języku programowania, który znasz lepiej, na przykład Python lub Ruby.

Jak to zepsuć?

- Wykorzystaj inny ulubiony język i użyj go do uruchomienia tego programu, przy czym podaj jak najwięcej argumentów powłoki. Zobacz, czy można zepsuć program, podając zbyt wiele argumentów.

- Zainicjalizuj zmienną `i` wraz z wartością `0` i zobacz, jaki to będzie miało wpływ na działanie programu. Czy musisz dostosować także zmienną `argc`, czy program po prostu działa? Dlaczego indeksowanie od zera sprawdza się w tym przypadku?
- Zmiennej `num_states` przypisz wartość większą niż liczba elementów tablicy, a następnie zobacz, jaki to będzie miało wpływ na działanie programu.

Zadania dodatkowe

- Ustal, jaki rodzaj kodu można umieszczać w poszczególnych fragmentach pętli `for`.
- Poszukaj informacji o tym, jak można wykorzystać przecinek do rozdzielenia wielu poleceń w poszczególnych fragmentach pętli `for`, ale między średnikami.
- Dowiedz się, do czego służy `NULL`, a następnie próbuj wykorzystać tę wartość dla jednego z elementów tablicy `states` i zobacz, jakie dane zostaną wyświetlone.
- Sprawdź, czy możesz przypisać element tablicy `states` tablicy `argv` przed wyświetleniem zawartości obu tablic. Następnie spróbuj zrobić odwrotnie.

ĆWICZENIE 14

Tworzenie i użycie funkcji

Dotąd używaliśmy jedynie funkcji zdefiniowanych w pliku nagłówkowym *stdio.h*. W tym ćwiczeniu utworzymy kilka własnych funkcji i wykorzystamy także parę innych.

Plik *ex14.c*:

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

// Deklaracje wyprzedzające
int can_print_it(char ch);
void print_letters(char arg[]);

void print_arguments(int argc, char *argv[])
{
    int i = 0;

    for (i = 0; i < argc; i++) {
        print_letters(argv[i]);
    }
}

void print_letters(char arg[])
{
    int i = 0;

    for (i = 0; arg[i] != '\0'; i++) {
        char ch = arg[i];

        if (can_print_it(ch)) {
            printf("'%c' == %d ", ch, ch);
        }
    }

    printf("\n");
}

int can_print_it(char ch)
{
    return isalpha(ch) || isblank(ch);
}

int main(int argc, char *argv[])
{
    print_arguments(argc, argv);
    return 0;
}
```

W powyższym kodzie źródłowym utworzyliśmy funkcje przeznaczone do wyświetlania znaków i kodów ASCII dla każdego znaku i spacji. Oto ogólne omówienie sposobu działania programu:

ex14.c:2. Dołączenie nowego pliku nagłówkowego, dzięki któremu uzyskujemy dostęp do wywołań `isalpha()` i `isblank()`.

ex14.c:5 – 6. Informujemy język C, że w dalszej części programu będziemy używać pewnych funkcji, choć teraz nie zostaną zdefiniowane. To jest *deklaracja wyprzedzająca* (ang. *forward declaration*), rozwiązująca problem związany z koniecznością użycia funkcji, zanim zostanie zdefiniowana.

ex14.c:8 – 15. Definiujemy funkcję `print_arguments()`, odpowiedzialną za wyświetlenie tablicy ciągów tekstowych zwykle przekazywanej funkcji `main()`.

ex14.c:17 – 30. Definiujemy kolejną funkcję, `print_letters()`, która jest wywoływana przez `print_arguments()` i służy do wyświetlania poszczególnych znaków oraz ich kodów ASCII.

ex14.c:32 – 35. Definiujemy funkcję `can_print_it()`, przeznaczoną po prostu do sprawdzenia wyniku operacji `isalpha(ch) || isblank(ch)` i jego przekazania funkcji `print_letters()`. Wynikiem może być `0` lub `1`.

ex14.c:38 – 41. Funkcja `main()` wywołuje `print_argument()` i tym samym uruchamia wykonanie całego łańcucha funkcji.

Nie przedstawiam tutaj dokładnego omówienia sposobu działania poszczególnych funkcji, ponieważ wykonują one operacje, które poznałeś już wcześniej. Powinieneś jednak zwrócić uwagę na fakt, że funkcje dodatkowe zostały zdefiniowane dokładnie tak samo, jak definiujemy `main()`. Jedyna różnica polega na udzieleniu pomocy językowi C i wcześniejszemu poinformowaniu o zamiarze użycia funkcji, które jeszcze nie zostały zdefiniowane w pliku. Właśnie do tego służy deklaracja z wyprzedzeniem.

Co powinieneś zobaczyć?

Aby poeksperymentować z omawianym programem, po prostu uruchamiaj go z różnymi argumentami powłoki, które następnie będą przekazywane zdefiniowanym funkcjom. Poniżej przedstawiłem przykładową sesję pracy z programem.

Sesja dla ćwiczenia 14.:

```
$ make ex14
cc -Wall -g ex14.c -o ex14
$ ./ex14
'e' == 101 'x' == 120
$ ./ex14 witaj to jest dobre
'e' == 101 'x' == 120
'w' == 119 'i' == 105 't' == 116 'a' == 97 'j' == 106
't' == 116 'o' == 111
'j' == 106 'e' == 101 's' == 115 't' == 116
'd' == 100 'o' == 111 'b' == 98 'r' == 114 'e' == 101
$ ./ex14 "dodalem tutaj 3 spacje"
```

```
'e' == 101 'x' == 120
'd' == 100 'o' == 111 'd' == 100 'a' == 97 'l' == 108 'e' == 101 'm' ==
↪109 ' ' == 32 't' == 116 'u' == 117 't' == 116 'a' == 97 'j' == 106 ' ' ==
↪32 ' ' == 32 's' == 115 'p' == 112 'a' == 97 'c' == 99 'j' == 106 'e' == 101
$
```

Wywołania `isalpha()` i `isblank()` wykonują całą pracę, określając, czy dany znak jest literą, czy znakiem odstępu. W ostatnim wywołaniu w powyższej sesji wyświetlane są wszystkie znaki poza `3`, ponieważ to jest cyfra.

Jak to zepsuć?

Omówiony tutaj program możemy zepsuć na dwa sposoby:

- Usunięcie deklaracji wyprzedzających wprowadzi zamieszanie w kompilatorze i spowoduje wyświetlenie komunikatów o błędach dotyczących funkcji `can_print_it()` i `print_letters()`.
- Podczas wywoływania `print_arguments()` w funkcji `main()` spróbuj dodać wartość `1` do zmiennej `argc`, aby w ten sposób wymusić wyjście poza granicę tablicy `argv`.

Zadania dodatkowe

- Przeprowadź refaktoryzację funkcji, aby zmniejszyć ich liczbę w programie. Czy na przykład naprawdę potrzebujemy funkcji `can_print_it()`?
- Pozwól funkcji `print_arguments()` na ustalenie wielkości poszczególnych argumentów w postaci ciągów tekstowych. W tym celu wykorzystaj funkcję `strlen()`, a wynik jej działania przekaż do funkcji `print_letters()`. Następnie zmodyfikuj funkcję `print_letters()` w taki sposób, aby przetwarzała jedynie podaną liczbę znaków i nie opierała swojego działania na znaku `'\0'`. Konieczne będzie umieszczenie w kodzie polecenia `#include <string.h>`.
- Wykorzystaj podręcznik systemowy `man` do wyszukania informacji o funkcjach `isalpha()` i `isblank()`. Użyj innych podobnych funkcji do wyświetlenia jedynie cyfr oraz innych znaków.
- Poczytaj o tym, jak inni programiści formatują tworzone funkcje. Nigdy nie korzystaj ze składni K&R (jest przestarzała i wprowadza zamieszanie), ale staraj się zrozumieć działanie kodu utworzonego za jej pomocą.

ĆWICZENIE 15

Wskaźniki, przerażające wskaźniki

Wskaźniki to sławne, mityczne potwory w języku C. Spróbuję je wyjaśnić w tym ćwiczeniu, pokazując składnię przeznaczoną do pracy ze wskaźnikami. Tak naprawdę wskaźniki nie należą do zbyt skomplikowanych koncepcji, ale bardzo często są nadużywane w dziwne sposoby, przez które stają się trudne w wykorzystaniu. Jeżeli będziesz unikać głupich sposobów użycia wskaźników, to przekonasz się, że są całkiem łatwe.

Aby zademonstrować wskaźniki w sposób umożliwiający mi omówienie ich, przygotowałem frywolny program, wyświetlający na trzy różne sposoby wiek grupy osób.

Plik *ex15.c*:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    // Utworzenie dwóch potrzebnych nam tablic danych.
    int ages[] = { 25, 47, 12, 89, 5 };
    char *names[] = {
        "Alan", "Franek",
        "Mariusz", "Janek", "Ludwik"
    };

    // Bezpieczne ustalenie wielkości tablicy ages.
    int count = sizeof(ages) / sizeof(int);
    int i = 0;

    // Pierwszy sposób opiera się na indeksach.
    for (i = 0; i < count; i++) {
        printf("%s przeżył %d lat.\n", names[i], ages[i]);
    }

    printf("---\n");

    // Ustawienie wskaźników tak, aby wskazywały początek tablic.
    int *cur_age = ages;
    char **cur_name = names;

    // Drugi sposób opiera się na wskaźnikach.
    for (i = 0; i < count; i++) {
        printf("%s ma %d lat.\n",
            *(cur_name + i), *(cur_age + i));
    }

    printf("---\n");

    // Trzeci sposób to użycie wskaźników, jakby po prostu były tablicami.
    for (i = 0; i < count; i++) {
```

```
37          printf("%s żyje od %d lat.\n", cur_name[i], cur_age[i]);
38      }
39
40      printf("---\n");
41
42      // Czwarty sposób też opiera się na wskaźnikach,
        // ale użytych w bezsensownie skomplikowany sposób.
43      for (cur_name = names, cur_age = ages;
44              (cur_age - ages) < count;
45              cur_name++, cur_age++) {
46          printf("%s przeżył jak dotąd %d lat.\n", *cur_name, *cur_age);
47      }
48
49      return 0;
50  }
```

Zanim przejdę do wyjaśnienia sposobu działania wskaźników, najpierw dokładnie omówię działanie powyższego programu wiersz po wierszu, abyś dokładnie wiedział, co się w nim dzieje. Czytając to omówienie, staraj się zapisywać na papierze odpowiedzi na swoje pytania, a następnie sprawdź, czy odpowiedzi te pasują do przedstawionego później wyjaśnienia wskaźników.

ex15.c:6 – 10. Utworzenie dwóch tablic. Pierwsza to `ages`, przechowuje dane typu `int`. Natomiast druga to `names` i przechowuje tablicę ciągów tekstowych.

ex15.c:13 – 14. Utworzenie pewnych zmiennych wykorzystywanych później w pętlach `for`.

ex15.c:16 – 19. Iteracja przez obie tablice i wyświetlenie informacji o wieku poszczególnych osób. W tym miejscu wykorzystujemy zmienną `i` w celu użycia indeksu tablicy.

ex15.c:24. Utworzenie wskaźnika prowadzącego do tablicy `ages`. Zwróć uwagę na użycie `int *` podczas tworzenia wskaźnika prowadzącego do wartości typu w postaci liczby całkowitej. To jest podobne do użycia `char *`, czyli wskaźnika prowadzącego do znaku, a ciąg tekstowy jest tablicą znaków. Czy dostrzegasz już podobieństwo?

ex15.c:25. Utworzenie wskaźnika prowadzącego do tablicy `names`. Skoro `char *` jest już wskaźnikiem do znaku, to mamy do czynienia z ciągiem tekstowym. Jednak potrzebujemy dwóch poziomów, ponieważ `names` to tablica dwuwymiarowa, co oznacza konieczność użycia `char **` dla wskaźnika będącego wskaźnikiem do znaku. Przeanalizuj tę koncepcję i spróbuj ją sobie wyjaśnić.

ex15.c:28 – 31. Iteracja przez tablice `ages` i `names`, ale z użyciem wskaźników *plus przesunięcia i*. Zapis `*(cur_name+1)` odpowiada zapisowi `name[i]` i odczytujesz go następująco: „Wartość (wskaźnik cur_name plus wartość i)".

ex15.c:35 – 38. W tych wierszach pokazałem składnię pozwalającą na uzyskanie dostępu do elementu tablicy. Jak widzisz, składnia ta jest taka sama zarówno dla wskaźnika, jak i tablicy.

ex15.c:42 – 47. To jest kolejna celowo wariacka pętla, której działanie jest dokładnie takie samo jak dwóch poprzednich. Jednak tym razem wykorzystujemy różne metody arytmetyczne na wskaźnikach.

ex15.c:43. Inicjalizacja pętli `for` przez przypisanie wskaźnikom `cur_name` i `cur_age` początku tablic, odpowiednio `name` i `ages`.

ex15.c:44. Sprawdzenie warunku w pętli `for` przez porównanie *odległości* wskaźnika `cur_age` od początku tablicy `ages`. Dlaczego takie podejście działa?

ex15.c:45. Inkrementacja w pętli `for`, która powoduje zwiększenie o jeden wartości `cur_name` i `cur_age`, aby wskazywały *następny* element w tablicy, odpowiednio `names` i `ages`.

ex15.c:46. Wskaźniki `cur_name` i `cur_ages` wskazują teraz jeden element w tablicy, z którym aktualnie pracują. Dlatego też możemy go wyświetlić za pomocą `*cur_name` i `*cur_age`, co oznacza „wartość wskazywana przez `cur_name`".

Ten niezwykle prosty program zawiera bardzo dużą ilość informacji. Moim celem jest umożliwienie Ci podjęcie próby samodzielnego określenia sposobu działania wskaźników, zanim zrobię to ja. *Nie kontynuuj lektury, dopóki nie zapiszesz na kartce ustalonego przez Ciebie sposobu działania wskaźników*.

Co powinieneś zobaczyć?

Po uruchomieniu programu przeanalizuj każdy wyświetlony wiersz danych wyjściowych i porównaj go z wierszem kodu źródłowego odpowiedzialnego za wygenerowanie tych danych. Jeżeli zachodzi potrzeba, użyj wywołań `printf()`, aby upewnić się, że patrzysz na odpowiedni wiersz kodu.

Sesja dla ćwiczenia 15.:

```
$ make ex15
cc -Wall -g ex15.c -o ex15
$ ./ex15
Alan przeżył 25 lat.
Franek przeżył 47 lat.
Mariusz przeżył 12 lat.
Janek przeżył 89 lat.
Ludwik przeżył 5 lat.
---
Alan ma 25 lat.
Franek ma 47 lat.
Mariusz ma 12 lat.
Janek ma 89 lat.
Ludwik ma 5 lat.
---
Alan żyje od 25 lat.
Franek żyje od 47 lat.
Mariusz żyje od 12 lat.
Janek żyje od 89 lat.
Ludwik żyje od 5 lat.
---
Alan przeżył jak dotąd 25 lat.
Franek przeżył jak dotąd 47 lat.
```

```
Maria przeżył jak dotąd 12 lat.
Mariusz przeżył jak dotąd 89 lat.
Ludwik przeżył jak dotąd 5 lat.
$
```

Poznajemy wskaźniki

Po wpisaniu polecenia takiego jak `ages[i]` przeprowadzasz *indeksowanie* tablicy `ages`, używając w tym celu liczby przechowywanej przez zmienną `i`. Jeżeli wspomnianą liczbą będzie zero, to wymienione polecenie można zapisać również w postaci `ages[0]`. Liczbę przechowywaną przez zmienną `i` nazywamy *indeksem*, ponieważ wskazuje interesujące nas położenie wewnątrz tablicy `ages`. Równie dobrze można by użyć pojęcia adresu, co oznaczałoby: „Z tablicy `ages` chcę pobrać element znajdujący się pod adresem `i`".

Skoro `i` jest indeksem, to czym jest `ages`? Z perspektywy języka C `ages` to konkretne miejsce w pamięci komputera, w którym znajduje się pierwsza z serii liczb całkowitych. To *również* jest adres, a kompilator C zastąpi każde wystąpienie `ages` adresem pierwszej liczby całkowitej przechowywanej w `ages`. Jeszcze inaczej można powiedzieć, że `ages` to „adres pierwszej liczby całkowitej w `ages`". W tym miejscu pojawia się jednak pewien problem: `ages` to adres w całej *pamięci komputera*. To nie jest jak w przypadku zmiennej `i` będącej po prostu adresem wewnątrz `ages`. Nazwa tablicy `ages` jest w rzeczywistości adresem w pamięci komputera.

Dochodzimy więc do następującego wniosku: język C uznaje cały komputer za jedną, ogromną tablicę bajtów. Oczywiście to nie będzie dla nas zbyt użyteczne, ale na bazie tej ogromnej tablicy bajtów język C umieszcza warstwy koncepcji *typów* i *wielkości* tych typów. Sposób działania wspomnianych koncepcji miałeś okazję zobaczyć już we wcześniejszej części książki. Teraz dowiesz się, jak C stosuje te koncepcje w przypadku tablic.

- Utworzenie bloku pamięci w komputerze.
- Nazwa `ages` *wskazuje* początek tego bloku.
- *Indeksowanie* bloku przez pobranie adresu bazowego `ages`, a następnie pobranie elementu znajdującego się w odległości `i` od wspomnianego adresu bazowego.
- Konwersja adresu wskazywanego przez `ages+i` na postać poprawnej wartości typu `int` oraz o właściwej wielkości, tak aby wartość zwrotna indeksu pokrywała się z tym, czego oczekujesz, czyli z wartością typu `int` przechowywaną w położeniu `i`.

Skoro można pobrać adres bazowy, taki jak `ages`, i dodać do niego inny adres, taki jak `i`, w celu otrzymania nowego adresu, to można tym samym utworzyć coś, co będzie cały czas wskazywało ten nowy adres, czy tak? Odpowiedź brzmi „tak", a wspomnianym „czymś" jest właśnie *wskaźnik*. Na tym polega działanie `cur_age` i `cur_name` w powyższym programie: są to zmienne wskazujące położenie w pamięci komputera, w którym znajdują się `ages` i `names`. Nasz przykładowy program wykonuje następnie pewne operacje matematyczne w celu pobrania wartości z pamięci. W jednym przypadku po prostu dodajemy `i` do `cur_age`, co odpowiada operacji wykonywanej przez program za pomocą polecenia `array[i]`. Jednak w ostatniej pętli `for` wskaźniki te są stosowane bez pomocy zmiennej `i`. Wówczas są traktowane bardziej jak połączenie w jedno tablicy i wartości przesunięcia liczby całkowitej.

Wskaźnik to po prostu adres wskazujący konkretne położenie w pamięci komputera oraz mający przypisany specyfikator typu, który pozwala na prawidłowe określenie wielkości danych. W kontekście omawianego przykładu można powiedzieć, że to przedstawione w postaci pojedynczego typu danych połączenie `ages` oraz `i`. Język C zna adres wskazywany przez wskaźnik, typ danych, wielkość danych podanego typu oraz potrafi pobrać dane znajdujące się pod podanym adresem. Podobnie jak w przypadku zmiennej `i`, także wskaźniki potrafią obsługiwać operacje inkrementacji, dekrementacji, dodawania i odejmowania. Ponadto podobnie jak w przypadku `ages`, masz możliwość pobierania wartości, wstawiania nowej oraz użycia wszystkich operacji dozwolonych dla tablicy.

Celem wskaźnika jest umożliwienie ręcznego indeksowania danych w bloki pamięci, gdy tablica nie wykonuje poprawnie takiego zadania. W praktycznie wszystkich pozostałych sytuacjach powinieneś stosować tablice. Zdarzają się jednak przypadki, gdy *musisz* pracować z niezmodyfikowanymi blokami pamięci i wówczas wskaźniki okazują się niezwykle przydatne. Wskaźnik daje niezmodyfikowany, bezpośredni dostęp do bloku pamięci, z którym możesz pracować.

Trzeba w tym miejscu wspomnieć o jeszcze jednej kwestii: w większości operacji na tablicy lub wskaźniku możesz używać tej samej składni. Na przykład masz wskaźnik prowadzący do czegoś, ale aby uzyskać dostęp, stosujesz składnię tablicy. Można więc wziąć tablicę i wykonać w niej operację arytmetyczną ze wskaźnikiem.

Praktyczne użycie wskaźników

Mamy cztery najważniejsze, użyteczne rzeczy, do których można wykorzystać wskaźniki w kodzie źródłowym utworzonym w języku C:

- Poproszenie system operacyjny o fragment pamięci i użycie wskaźnika do pracy z tym blokiem pamięci. Obejmuje to ciągi tekstowe oraz konstrukcję, której jeszcze nie omówiliśmy w książce, czyli strukturę (`struct`).
- Przekazanie ogromnych bloków pamięci (na przykład dużych struktur) do funkcji za pomocą wskaźników, co zwalnia z konieczności przekazywania funkcji ogromnych danych.
- Pobranie adresu funkcji i jego wykorzystanie jako dynamicznego wywołania zwrotnego.
- Skanowanie skomplikowanych fragmentów pamięci, konwersja bajtów gniazda sieciowego na struktury danych lub pliku możliwe do późniejszego przetworzenia.

Do wszystkich pozostałych zadań powinny być wykorzystywane tablice, choć będziesz spotykał się z użyciem wskaźników w tym celu. Na początku istnienia języka C programiści wykorzystywali wskaźniki, aby przyspieszyć działanie programów, ponieważ kompilatory naprawdę kiepsko radziły sobie z optymalizacją użycia tablic. Obecnie składnia dostępu do tablicy i składnia wskaźnika powodują wygenerowanie tego samego kodu maszynowego i zastosowanie takich samych optymalizacji, więc używanie wskaźników nie jest konieczne. Zamiast tego powinieneś wykorzystywać tablice, gdy tylko istnieje możliwość, a wskaźniki jedynie w charakterze optymalizacji wydajności, o ile jest to absolutnie konieczne.

Leksykon wskaźnika

Przedstawię teraz niewielki leksykon przydatny podczas odczytu i zapisu wskaźników. Jeśli kiedykolwiek będziesz musiał zająć się skomplikowanym poleceniem wskaźnika, po prostu powróć do tego fragmentu i przeanalizuj kod bit po bicie (lub najlepiej nie używaj danego kodu, ponieważ prawdopodobnie nie będzie on dobrym kodem).

`typ *ptr` — wskaźnik o nazwie `ptr` i typie `typ`;

`*ptr` — wartość, do której prowadzi wskaźnik `ptr`;

`*(ptr + i)` — wartość wskaźnika plus przechowywana przez zmienną `i`;

`&coś` — adres w pamięci, gdzie znajduje się `coś`;

`typ *ptr = &coś` — wskaźnik o nazwie `ptr`, typie `typ` i wskazujący `coś`;

`ptr++` — inkrementacja wartości wskazywanej przez `ptr`.

Powyższy prosty leksykon będziemy stosować do rozszyfrowywania wskaźników używanych od tego miejsca w książce.

Wskaźniki nie są tablicami

Niezależnie od wszystkiego nigdy nie powinieneś myśleć, że wskaźniki i tablice oznaczają jedno i to samo. To nie jest to samo, mimo że język C pozwala na pracę z obiema konstrukcjami w taki sam sposób. Na przykład gdy w przedstawionym wcześniej kodzie wykonujemy operację `sizeof(cur_age)`, otrzymujemy wielkość *wskaźnika*, a nie wielkość wskazywanych przez niego danych. Jeżeli chcesz poznać wielkość całej tablicy, musisz użyć jej nazwy, jak to zostało pokazane w wierszu 13. omawianego kodu źródłowego.

Do zrobienia: rozbuduj nieco powyższą koncepcję i wskaż, co nie działa dokładnie tak samo dla wskaźników i tablic.

Jak to zepsuć?

Omawiany w ćwiczeniu program można zepsuć przez wskazanie wskaźnikom nieprawidłowych danych.

- Zmodyfikuj program tak, aby wskaźnik `cur_age` prowadził do `names`. Będziesz musiał wykorzystać rzutowanie w języku C w celu wymuszenia podanej zmiany, a więc poszukaj informacji o tym, jak można to zrobić.
- W ostatniej pętli `for` spróbuj na różne sposoby zepsuć operacje matematyczne.
- Zmodyfikuj pętle w taki sposób, aby pierwsza rozpoczynała się od ostatniego elementu tablicy, a następnie przebiegała do początku tablicy. To zadanie jest trudniejsze, niż się wydaje.

Zadania dodatkowe

- Wszystkie tablice w programie zmień na wskaźniki.
- Wszystkie wskaźniki w programie zmień na tablice.
- Powróć do wcześniejszych programów wykorzystujących tablice i spróbuj zmodyfikować je tak, aby używały wskaźników.
- Przetwórz argumenty powłoki za pomocą jedynie wskaźników, podobnie jak to zostało zrobione dla tablicy `names` w omawianym programie.
- Poeksperymentuj z różnymi wariantami pobierania wartości i adresów pewnych rzeczy.
- Na końcu programu dodaj kolejną pętlę `for` wyświetlającą adres używanego wskaźnika. W tym celu w wywołaniu funkcji `printf()` konieczne jest użycie specyfikatora formatu `%p`.
- Zmodyfikuj program tak, aby używał funkcji dla każdego ze sposobów wyświetlania danych wyjściowych. Spróbuj przekazać wskaźniki tym funkcjom, aby mogły pracować z danymi. Pamiętaj, że możesz zadeklarować funkcję do akceptacji wskaźnika, ale po prostu używać go jak tablicy.
- Pętlę `for` zastąp pętlą `while` i zobacz, który z tych rodzajów pętli będzie się lepiej sprawdzał podczas pracy ze wskaźnikami.

ĆWICZENIE 16

Struktury i prowadzące do nich wskaźniki

W tym ćwiczeniu dowiesz się, jak utworzyć strukturę, wskaźnik prowadzący do niej oraz jak najsensowniej wykorzystać wewnętrzne struktury pamięci. Będziemy tutaj wykorzystywać wiedzę zdobytą w poprzednim ćwiczeniu, a następnie przystąpimy do tworzenia wspomnianych struktur z użyciem wywołania `malloc()` i na podstawie niezmodyfikowanych danych z pamięci.

Jak zwykle zaczynamy od napisania i uruchomienia programu przedstawiającego koncepcje omawiane w ćwiczeniu.

Plik *ex16.c*:

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

struct Person {
    char *name;
    int age;
    int height;
    int weight;
};

struct Person *Person_create(char *name, int age, int height,
        int weight)
{
    struct Person *who = malloc(sizeof(struct Person));
    assert(who != NULL);

    who->name = strdup(name);
    who->age = age;
    who->height = height;
    who->weight = weight;

    return who;
}

void Person_destroy(struct Person *who)
{
    assert(who != NULL);

    free(who->name);
```

```
32      free(who);
33  }
34
35  void Person_print(struct Person *who)
36  {
37      printf("Osoba: %s\n", who->name);
38      printf("\tWiek: %d\n", who->age);
39      printf("\tWzrost: %d\n", who->height);
40      printf("\tWaga: %d\n", who->weight);
41  }
42
43  int main(int argc, char *argv[])
44  {
45      // Utworzenie dwóch struktur przedstawiających osoby.
46      struct Person *joe = Person_create("Joe Alex", 32, 64, 140);
47
48      struct Person *frank = Person_create("Frank Blank", 20, 72, 180);
49
50      // Wyświetlenie informacji o osobach oraz miejscu ich przechowywania w pamięci.
51      printf("Joe znajduje się w pamięci pod adresem %p:\n", joe);
52      Person_print(joe);
53
54      printf("Frank znajduje się w pamięci pod adresem %p:\n", frank);
55      Person_print(frank);
56
57      // Zwiększenie wieku każdej osoby o 20 lat i ponowne wyświetlenie informacji.
58      joe->age += 20;
59      joe->height -= 2;
60      joe->weight += 40;
61      Person_print(joe);
62
63      frank->age += 20;
64      frank->weight += 20;
65      Person_print(frank);
66
67      // Usunięcie obu struktur, aby można było przeprowadzić operacje porządkowe.
68      Person_destroy(joe);
69      Person_destroy(frank);
70
71      return 0;
72  }
```

W celu omówienia powyższego programu zastosuję podejście nieco inne niż wcześniej. Nie zamierzam tutaj wyjaśniać działania programu wiersz po wierszu, tylko zachęcam Ciebie do *napisania* tego. Poniżej ogólnie zaprezentuję funkcjonalność poszczególnych fragmentów programu, a Twoim zadaniem jest opisanie znaczenia wszystkich wierszy.

Polecenia `include`. Na początku kodu importujemy kilka nowych plików nagłówkowych w celu uzyskania dostępu do nowych funkcji. Czy jesteś w stanie określić, jaką funkcjonalność oferują poszczególne pliki nagłówkowe?

Struktura `Person`. Jest to składająca się z czterech elementów struktura przeznaczona do przedstawienia osoby. Ostatecznym wynikiem jest nowy typ złożony pozwalający

na jednoczesne odwołanie się do wszystkich elementów lub do jedynie wybranych, za pomocą ich nazw. Konstrukcja ta przypomina rekord w tabeli bazy danych lub klasę w języku programowania zorientowanego obiektowo.

Funkcja `Person_create()`. Potrzebujemy sposobu pozwalającego na tworzenie struktur osób, stąd konieczność opracowania odpowiedniej funkcji. Oto kilka najważniejszych kwestii dotyczących tej funkcji:

- Używamy wywołania `malloc()` w celu alokacji pamięci i prosimy system operacyjny o dostarczenie fragmentu ciągłej, niezmodyfikowanej pamięci.
- Funkcji `malloc()` przekazujemy wartość w postaci wywołania `sizeof(struct Person)`, aby obliczyć całkowitą wielkość struktury wraz z wszystkimi znajdującymi się w niej elementami.
- Wywołanie `assert()` ma zagwarantować, że na skutek wykonania funkcji `malloc()` otrzymujemy poprawny fragment pamięci. Istnieje stała specjalna o nazwie `NULL`, której znaczenie można określić jako „brak przypisanej wartości lub nieprawidłowy wskaźnik". Działanie funkcji `assert()` polega w zasadzie na sprawdzeniu, czy wartością zwrotną `malloc()` jest `NULL` lub nieprawidłowy wskaźnik.
- Każdy element struktury `Person` inicjalizujemy za pomocą składni `x->y`, co pozwala na wskazanie elementu struktury, któremu ma zostać przypisana wartość.
- Funkcja `strdup()` jest używana do powielenia ciągu tekstowego dla `name`, tylko w celu potwierdzenia, że struktura faktycznie ma wymieniony element. Działanie funkcji `strdup()` w rzeczywistości przypomina wywołanie `malloc()`, a ponadto kopiuje pierwotny ciąg tekstowy do rezerwowanego bloku pamięci.

Funkcja `Person_destroy()`. Skoro mamy funkcję tworzącą strukturę, to potrzebujemy funkcji, która będzie tę strukturę niszczyć. Ponownie wykorzystujemy wywołanie `assert()` w celu upewnienia się o otrzymaniu poprawnych danych wyjściowych. Następnie za pomocą funkcji `free()` zwalniamy pamięć zarezerwowaną wcześniej przez wywołania `malloc()` i `strdup()`. Jeżeli nie zwolnisz tej pamięci, to będziesz miał do czynienia z tak zwanym *wyciekiem pamięci*.

Funkcja `Person_print()`. Potrzebny jest sposób na wyświetlenie informacji o osobach i do tego celu służy wymieniona funkcja. Do pobierania wyświetlanych informacji wykorzystywana jest dokładnie ta sama składnia `x->y` co w przypadku przypisywania wartości poszczególnym elementom.

Funkcja `main()`. W funkcji `main()` używamy wszystkich utworzonych wcześniej funkcji oraz struktury `Person` do wykonania następujących operacji:

- Utworzenie dwóch osób: `joe` i `frank`.
- Wyświetlenie informacji o nich. Zwróć uwagę na użycie specyfikatora formatu `%p` podającego *adres* w pamięci, pod którym struktura została faktycznie umieszczona.
- Zmianę wieku obu osób o 20 lat, a także zmianę innych cech danej osoby.

- Wyświetlenie informacji o osobach po zmianie ich wieku.
- Usunięcie struktur, aby możliwe było poprawne przeprowadzenie operacji porządkujących.

Dokładnie przeanalizuj przedstawiony opis, a następnie wykonaj następujące zadania:

- Wyszukaj informacje o wszystkich nieznanych Ci plikach nagłówkowych. Pamiętaj o możliwości użycia podręcznika systemowego (`man 2 nazwa-funkcji` i `man 3 nazwa-funkcji`) do znalezienia odpowiednich informacji. Oczywiście możesz również poszukać informacji w internecie.
- Nad każdym wierszem kodu umieść *komentarz* opisujący działanie danego polecenia.
- Prześledź każde wywołanie funkcji i zmienne, aby dokładnie zrozumieć sposób działania programu.
- Wyszukaj informacje o wszystkich nieznanych Ci symbolach.

Co powinieneś zobaczyć?

Po umieszczeniu w programie komentarzy wyjaśniających sposób jego działania upewnij się, że faktycznie działa i generuje pokazane poniżej dane wyjściowe.

Sesja dla ćwiczenia 16.:

```
$ make ex16
cc -Wall -g ex16.c -o ex16

$ ./ex16
Joe znajduje się w pamięci pod adresem 0xeba010:
Osoba: Joe Alex
  Wiek: 32
  Wzrost: 64
  Waga: 140
Frank znajduje się w pamięci pod adresem 0xeba050:
Osoba: Frank Blank
  Wiek: 20
  Wzrost: 72
  Waga: 180
Osoba: Joe Alex
  Wiek: 52
  Wzrost: 62
  Waga: 180
Osoba: Frank Blank
  Wiek: 40
  Wzrost: 72
  Waga: 200
```

Poznajemy struktury

Jeżeli wykonałeś powyższe zadania, działanie struktur powinno mieć dla Ciebie sens. Pozwól mi jednak wyjaśnić ich przeznaczenie, aby mieć pewność, że wszystko prawidłowo zrozumiałeś.

Struktura w języku C to kolekcja innych typów danych (zmiennych) przechowywanych w jednym bloku pamięci, a dostęp do poszczególnych zmiennych może odbywać się niezależnie od pozostałych, za pomocą ich nazw. Struktura jest więc podobna do rekordu w tabeli bazy danych lub bardzo prostej klasy w języku programowania zorientowanego obiektowo. Strukturę można przedstawić następująco:

- W powyższym kodzie za pomocą polecenia `struct` tworzymy strukturę zawierającą elementy `name`, `age`, `weight` i `height`.
- Każdy z wymienionych elementów ma typ, na przykład `int`.
- Następnie język C łączy te elementy ze sobą, aby znajdowały się w pojedynczej strukturze.
- W tym momencie struktura `Person` jest *złożonym typem danych*, co oznacza możliwość odwoływania się do niej za pomocą tego samego rodzaju wyrażeń co w przypadku innych typów danych.
- Całą grupę elementów można przekazywać do innych funkcji, podobnie jak to zrobiliśmy w przypadku funkcji `Person_print()`.
- Dostęp do poszczególnych elementów struktury jest możliwy za pomocą ich nazw i składni `x->y`, jeżeli mamy do czynienia ze wskaźnikiem.
- Można utworzyć strukturę niewymagającą wskaźnika, a następnie pracować z nią z wykorzystaniem składni `x.y`. Do tego tematu powrócimy jeszcze w sekcji „Zadania dodatkowe".

Jeżeli nie masz struktury, to musisz ustalić wielkość, format i położenie poszczególnych elementów zawierających takie dane, jak przedstawiłem wcześniej. Tak naprawdę praca w większości wczesnego kodu w asemblerze (a nawet i teraz) odbywa się właśnie w przedstawiony sposób. Język C pozwala na obsługę struktur pamięci za pomocą wspomnianych złożonych typów danych, a programista może się skoncentrować na zadaniach, które chce wykonać na ich podstawie.

Jak to zepsuć?

Sposoby zepsucia omawianego programu opierają się na użyciu wskaźników i wywołań `malloc()`.

- Spróbuj przekazać wartość `NULL` funkcji `Person_destroy()` i zobacz, jaki będzie efekt. Jeżeli działanie programu nie zostało przerwane, to prawdopodobnie nie podałeś opcji `-g` w definicji `CFLAG`, znajdującej się w pliku *Makefile*.
- Zapomnij o wywołaniu funkcji `Person_destroy()` na końcu programu, a następnie uruchom go w debugerze i zobacz komunikat o niezwolnieniu pamięci. Znajdź

opcje, jakie musisz przekazać debugerowi, aby wyświetlić informacje o tym, jak doprowadziłeś do powstania wycieku pamięci.

- Pomiń polecenie `free who->name` w funkcji `Person_destroy()` i otrzymane dane wyjściowe porównaj z wcześniejszymi. Ponownie użyj odpowiednich opcji debugera, aby uzyskać dokładne informacje o tym, co tak naprawdę zepsułeś.
- Tym razem przekaż wartość `NULL` funkcji `Person_print()` i sprawdź, jakie komunikaty zostaną wygenerowane przez debuger. Powinieneś przekonać się, że wartość `NULL` to łatwy sposób na doprowadzenie do awarii programu.

Zadania dodatkowe

W tej sekcji zachęcam Cię do wykonania nieco trudniejszego zadania: skonwertuj omówiony program na postać, która *nie* używa wskaźników i wywołań `malloc()`. To będzie trudne zadanie, więc musisz zebrać informacje dotyczące poniższych kwestii:

- Utworzenie struktury na *stosie*, podobnie jak w przypadku każdej innej zmiennej.
- Inicjalizacja tej struktury za pomocą składni `x.y` zamiast `x->y`.
- Przekazanie tej struktury do innych funkcji bez użycia wskaźnika.

ĆWICZENIE 17

Alokacja pamięci stosu i sterty

W tym ćwiczeniu dokonamy dużego skoku w poziomie trudności i utworzymy pełny, mały program przeznaczony do zarządzania bazą danych. Nasza baza danych nie będzie charakteryzowała się dobrą wydajnością i nie umożliwi przechowywania zbyt wielu danych, ale pozwoli na wykorzystanie większości zdobytej dotąd wiedzy. Ponadto nieco bardziej oficjalnie wprowadzi temat alokacji pamięci, a także rozpoczniemy pracę z plikami. Wykorzystamy pewne funkcje wejścia-wyjścia związane z plikami, ale nie zagłębię się w ten temat i pozwolę Ci na podjęcie próby samodzielnego rozszyfrowania tego zagadnienia.

Jak zwykle zaczniemy od wpisania całego programu i zagwarantowania jego poprawnego działania, a dopiero później przejdziemy do omówienia kodu źródłowego.

Plik *ex17.c*:

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

#define MAX_DATA 512
#define MAX_ROWS 100

struct Address {
    int id;
    int set;
    char name[MAX_DATA];
    char email[MAX_DATA];
};

struct Database {
    struct Address rows[MAX_ROWS];
};

struct Connection {
    FILE *file;
    struct Database *db;
};

void die(const char *message)
{
    if (errno) {
        perror(message);
    } else {
        printf("BŁĄD: %s\n", message);
    }

```

```
34      exit(1);
35  }
36
37  void Address_print(struct Address *addr)
38  {
39      printf("%d %s %s\n", addr->id, addr->name, addr->email);
40  }
41
42  void Database_load(struct Connection *conn)
43  {
44      int rc = fread(conn->db, sizeof(struct Database), 1, conn->file);
45      if (rc != 1)
46          die("Nie udało się wczytać bazy danych.");
47  }
48
49  struct Connection *Database_open(const char *filename, char mode)
50  {
51      struct Connection *conn = malloc(sizeof(struct Connection));
52      if (!conn)
53          die("Błąd pamięci.");
54
55      conn->db = malloc(sizeof(struct Database));
56      if (!conn->db)
57          die("Błąd pamięci.");
58
59      if (mode == 'c') {
60          conn->file = fopen(filename, "w");
61      } else {
62          conn->file = fopen(filename, "r+");
63
64          if (conn->file) {
65              Database_load(conn);
66           }
67      }
68
69      if (!conn->file)
70          die("Nie udało się otworzyć pliku.");
71
72      return conn;
73  }
74
75  void Database_close(struct Connection *conn)
76  {
77      if (conn) {
78          if (conn->file)
79              fclose(conn->file);
80          if (conn->db)
81              free(conn->db);
82          free(conn);
83      }
84  }
85
86  void Database_write(struct Connection *conn)
```

```
87 {
88     rewind(conn->file);
89
90     int rc = fwrite(conn->db, sizeof(struct Database), 1, conn->file);
91     if (rc != 1)
92         die("Nie udało się zapisać w bazie danych.");
93
94     rc = fflush(conn->file);
95     if (rc == -1)
96          die("Nie udało się opróżnić bazy danych.");
97 }
98
99 void Database_create(struct Connection *conn)
100 {
101     int i = 0;
102
103     for (i = 0; i < MAX_ROWS; i++) {
104         // Utworzenie prototypu w celu jego inicjalizacji.
105         struct Address addr = {.id = i,.set = 0 };
106         // Przypisanie danych prototypowi.
107         conn->db->rows[i] = addr;
108     }
109 }
110
111 void Database_set(struct Connection *conn, int id, const char *name,
112         const char *email)
113 {
114     struct Address *addr = &conn->db->rows[id];
115     if (addr->set)
116         die("Dane już istnieją, najpierw je usuń.");
117
118     addr->set = 1;
119     // OSTRZEŻENIE: Błąd, przeczytaj sekcję "Jak to zepsuć?", aby dowiedzieć się,
        // jak usunąć ten błąd.
120     char *res = strncpy(addr->name, name, MAX_DATA);
121     // Zademonstrowanie błędu wywołania strncpy().
122     if (!res)
123         die("Nie udało się skopiować imienia.");
124
125     res = strncpy(addr->email, email, MAX_DATA);
126     if (!res)
127         die("Nie udało się skopiować adresu e-mail.");
128 }
129
130 void Database_get(struct Connection *conn, int id)
131 {
132     struct Address *addr = &conn->db->rows[id];
133
134     if (addr->set) {
135         Address_print(addr);
136     } else {
137         die("Identyfikator nie został zdefiniowany.");
138     }
```

```
139 }
140
141 void Database_delete(struct Connection *conn, int id)
142 {
143     struct Address addr = {.id = id,.set = 0 };
144     conn->db->rows[id] = addr;
145 }
146
147 void Database_list(struct Connection *conn)
148 {
149     int i = 0;
150     struct Database *db = conn->db;
151
152     for (i = 0; i < MAX_ROWS; i++) {
153         struct Address *cur = &db->rows[i];
154
155         if (cur->set) {
156             Address_print(cur);
157         }
158     }
159 }
160
161 int main(int argc, char *argv[])
162 {
163     if (argc < 3)
164         die("UŻYCIE: ex17 <baza> <akcja> [parametry akcji]");
165
166     char *filename = argv[1];
167     char action = argv[2][0];
168     struct Connection *conn = Database_open(filename, action);
169     int id = 0;
170
171     if (argc > 3) id = atoi(argv[3]);
172     if (id >= MAX_ROWS) die("Nie ma aż tylu rekordów.");
173
174     switch (action) {
175         case 'c':
176             Database_create(conn);
177             Database_write(conn);
178             break;
179
180         case 'g':
181             if (argc != 4)
182                 die("Trzeba podać identyfikator.");
183
184             Database_get(conn, id);
185             break;
186
187         case 's':
188             if (argc != 6)
189                 die("Trzeba podać identyfikator, imię i adres e-mail.");
190
191             Database_set(conn, id, argv[4], argv[5]);
```

```
192             Database_write(conn);
193             break;
194
195         case 'd':
196             if (argc != 4)
197             die("Aby usunąć rekord, trzeba podać jego identyfikator.");
198
199             Database_delete(conn, id);
200             Database_write(conn);
201             break;
202
203         case 'l':
204             Database_list(conn);
205             break;
206         default:
207             die("Nieprawidłowa akcja: c=utwórz, g=pobierz, s=ustaw, d=usuń,
                ↪l=wyświetl");
208     }
209
210     Database_close(conn);
211
212     return 0;
213 }
```

W powyższym programie wykorzystujemy zbiór struktur w celu utworzenia prostej bazy danych dla książki adresowej. Znajdziesz tutaj pewne elementy, które wcześniej nie były używane. Dlatego też powinieneś przeanalizować ten kod źródłowy wiersz po wierszu, wyjaśnić sobie działanie każdego z nich i odszukać nieznane Ci funkcje. Istnieje kilka kluczowych kwestii, na które powinieneś zwrócić szczególną uwagę:

Polecenia `#define` dla stałych. Wykorzystamy inną funkcję preprocesora C (CPP) w celu utworzenia stałych `MAX_DATA` i `MAX_ROWS`. Do tematu preprocesora C jeszcze wrócimy w dalszej części książki, a teraz wykorzystamy jego możliwości do utworzenia niezawodnie działających stałych. Wprawdzie istnieją jeszcze inne sposoby, ale nie mają zastosowania w określonych sytuacjach.

Struktury o wielkości ustalonej na stałe. Struktura `Address` używa wspomnianych wcześniej stałych do utworzenia fragmentów danych o stałej wielkości. Tego rodzaju dane charakteryzują się mniejszą efektywnością, choć jednocześnie są łatwiejsze do przechowywania i odczytu. Struktura `Database` również ma na stałe ustaloną wielkość, ponieważ jest o stałej wielkości tablicą struktur `Address`. W ten sposób wszystkie dane można zapisać na dysku i później je przenieść.

Funkcja `die()` powodująca przerwanie programu i generująca komunikat o błędzie. W małym programie, takim jak tutaj omawiany, tworzymy pojedynczą funkcję, która kończy działanie programu i generuje komunikat o błędzie, gdy wystąpi problem. Nadałem tej funkcji nazwę `die()`; wykorzystujemy ją do zakończenia działania programu i wygenerowania błędu, gdy wywołanie jakiejkolwiek funkcji zakończy się niepowodzeniem lub program otrzyma nieprawidłowe dane wejściowe.

Zmienna `errno` i funkcja `perror()` przeznaczone do zgłaszania błędów. Kiedy wynikiem działania funkcji jest błąd, zwykle następuje ustawienie zewnętrznej zmiennej

o nazwie `errno`, dostarczającej dokładne informacje o przyczynie niepowodzenia. Ponieważ to są jedynie liczby, trzeba użyć funkcji `perror()` w celu wyświetlenia komunikatu o błędzie.

Funkcje struktury `FILE`. Do pracy z plikami korzystamy z nowych funkcji, takich jak `fopen()`, `fread()`, `fclose()` i `rewind()`. Wszystkie wymienione funkcje działają ze strukturą `FILE`, która jest podobna do pozostałych struktur, z wyjątkiem tego, że została zdefiniowana przez bibliotekę standardową C.

Zagnieżdżone struktury wskaźników. Powinieneś dokładnie przeanalizować użycie zagnieżdżonych struktur oraz pobierania adresów elementów tablicy. Szczególną uwagę zwróć na kod taki jak `&conn->db->rows[i]`, oznaczający: „Pobierz element `i` znajdujący się w tablicy `rows`, która z kolei znajduje się w strukturze `db`, a ta w strukturze `conn`, a następnie pobierz jego adres (`&`)".

Kopiowanie prototypów struktur. Najlepiej to widać w funkcji `Database_delete()`. Używamy tymczasowej, lokalnej struktury `Address`, inicjalizujemy jej elementy `id` i `set`, a następnie kopiujemy ją do tablicy `rows` przez przypisanie jej do wybranego elementu. Dzięki takiemu podejściu mamy pewność, że wszystkie elementy poza `set` i `id` zostaną zainicjalizowane wraz z zerami, co w rzeczywistości ułatwia zapis danych. Nawiasem mówiąc, nie należy używać funkcji `malloc()` do przeprowadzania tego rodzaju operacji kopiowania struktury. Nowoczesny język C pozwala po prostu na przypisanie jednej struktury do innej i automatycznie zajmuje się obsługą jej skopiowania.

Przetwarzanie skomplikowanych argumentów. W programie mamy całkiem skomplikowane operacje przetwarzania argumentów, choć tak naprawdę to nie jest najlepszy sposób ich zademonstrowania. W dalszej części książki poznasz lepsze rozwiązania w tym zakresie.

Konwersja ciągów tekstowych na dane typu `int`. Funkcji `atoi()` używamy do pobrania ciągu tekstowego z identyfikatora w powłoce i jego konwersji na postać zmiennej typu `int`. Więcej informacji na temat tej i podobnych funkcji znajdziesz w dalszej części książki.

Alokacja ogromnych danych na stercie. Celem omawianego programu jest pokazanie użycia wywołania `malloc()` do zarezerwowania przez system operacyjny ogromnej ilości pamięci podczas tworzenia bazy danych. Więcej informacji na ten temat znajdziesz w dalszej części książki.

Wartość `NULL` wynosi `0`, więc operacje boolowskie działają. W wielu operacjach sprawdzamy, czy wskaźnik nie przyjmuje wartości `NULL`. Odbywa się to za pomocą wywołania `if(!ptr) die("Niepowodzenie!")`, ponieważ wartość `NULL` oznacza fałsz. Oczywiście można w tych miejscach użyć wyraźnego wywołania `if(ptr == NULL) die("Niepowodzenie!")`. W niektórych rzadziej spotykanych systemach wartość `NULL` będzie w komputerze używana do przechowywania (przedstawienia) czegoś innego niż `0`. Jednak zgodnie ze standardem języka C powinieneś być w stanie tworzyć kod, jeśli `NULL` oznacza wartość `0`. Na razie przyjmujemy założenie, że kiedy mówię „`NULL` oznacza `0`", to mam na myśli jej wartość z punktu widzenia osoby przesadnie pedantycznej.

Co powinieneś zobaczyć?

Powinieneś poświęcić sporo czasu na przetestowanie działania programu, uruchomienie go w debugerze w celu potwierdzenia, że użycie pamięci przez program odbywa się prawidłowo. Poniżej przedstawiłem zapis sesji normalnego testowania programu, a następnie użycia debugera do sprawdzenia operacji.

Sesja dla ćwiczenia 17.:

```
$ make ex17
cc -Wall -g ex17.c -o ex17
$ ./ex17 db.dat c
$ ./ex17 db.dat s 1 zed zed@zedshaw.com
$ ./ex17 db.dat s 2 frank frank@zedshaw.com
$ ./ex17 db.dat s 3 joe joe@zedshaw.com
$
$ ./ex17 db.dat l
1 zed zed@zedshaw.com
2 frank frank@zedshaw.com
3 joe joe@zedshaw.com
$ ./ex17 db.dat d 3
$ ./ex17 db.dat l
1 zed zed@zedshaw.com
2 frank frank@zedshaw.com
$ ./ex17 db.dat g 2
2 frank frank@zedshaw.com
```

Alokacja stosu kontra sterty

Programiści mają teraz wygodę. Mogą eksperymentować z językami takimi jak Ruby i Python, po prostu tworząc obiekty oraz zmienne bez przejmowania się ich cyklem życiowym. Nie trzeba zastanawiać się, czy dany element znajduje się na *stosie*, a także nad tym, co jest na stercie. Może tego nie wiesz, ale istnieje znaczne prawdopodobieństwo, że używany przez Ciebie język programowania w ogóle nie umieszcza zmiennych na stosie. Wszystko odbywa się na stercie, choć pewnie nawet *nie wiesz*, co to jest.

Pod tym względem język C jest inny, ponieważ do wykonania pracy używa rzeczywistego procesora, co oznacza zarezerwowanie pewnego fragmentu pamięci RAM, nazywanego stosem, oraz kolejnego, nazywanego stertą. Na czym polega różnica? Wszystko zależy od tego, gdzie umieszczasz dane.

Sterta jest łatwiejsza do wyjaśnienia, ponieważ oznacza po prostu całą pozostałą pamięć w komputerze. Aby uzyskać do niej dostęp, używasz funkcji `malloc()`. W trakcie każdego wywołania `malloc()` system operacyjny wykorzystuje funkcje wewnętrzne do zarejestrowania fragmentu pamięci dla Twojego programu, a następnie przekazuje wskaźnik do tego fragmentu. Po zakończeniu pracy należy użyć funkcji `free()` w celu zwrócenia systemowi operacyjnemu wcześniej zarezerwowanej pamięci, która dzięki temu będzie mogła być wykorzy-

stana przez inne programy. Jeżeli pamięć nie zostanie zwrócona, mamy wówczas do czynienia z tak zwanym *wyciekiem* pamięci, ale na szczęście narzędzie Valgrind może pomóc w wyśledzeniu tego.

Natomiast stos to specjalny fragment pamięci przeznaczony do przechowywania zmiennych tymczasowych, które każda funkcja tworzy jako zmienne lokalne dla danej funkcji. Każdy argument funkcji jest *umieszczany* na stosie, a następnie używany wewnątrz tej funkcji. To naprawdę jest struktura danych w postaci stosu, więc ostatnia umieszczona na nim rzecz jest pierwszą do pobrania. To dotyczy również wszystkich zmiennych lokalnych, takich jak `char action` i `int id` w `main()`. Zaletą użycia stosu jest to, że po zakończeniu działania funkcji kompilator C wyrzuca te zmienne ze stosu. To jest proste rozwiązanie — umieszczanie zmiennych na stosie chroni przed powstaniem wycieku pamięci.

Najłatwiejszym podejściem jest trzymanie się następującej mantry: jeżeli zmienna lub funkcja nie powstała na skutek użycia wywołania `malloc()`, to znajduje się na stosie.

Mamy trzy najważniejsze problemy związane ze stosem i stertą:

- Jeżeli na skutek wywołania `malloc()` otrzymasz blok pamięci, a wskaźnik do niego zostanie umieszczony na stosie, to po zakończeniu działania funkcji ten wskaźnik zostanie usunięty ze stosu i trwale utracony.
- Jeżeli umieścisz zbyt wiele danych na stosie (na przykład ogromne struktury danych i tablice), to możesz doprowadzić do tak zwanego *przepełnienia stosu* i działanie programu zostanie przerwane. W takim przypadku należy korzystać ze sterty i wywołań `malloc()`.
- Jeżeli masz wskaźnik prowadzący do czegoś na stosie, a następnie przekażesz go lub zwrócisz z funkcji, to funkcja otrzymująca ten wskaźnik spowoduje błąd naruszenia ochrony pamięci (ang. *segmentation fault*, inaczej *segfault*), ponieważ rzeczywiste dane zostały usunięte ze stosu i trwale utracone. W tym momencie wskaźnik prowadzi do niezarezerwowanego miejsca w pamięci.

Dlatego też w omawianym programie mamy funkcję `Database_open()`, odpowiedzialną za alokację pamięci lub zakończenie działania programu, oraz `Database_close()`, która zwalnia całą zarezerwowaną wcześniej pamięć. Jeżeli utworzysz funkcję wykonującą wszystkie operacje lub żadną, a następnie funkcję bezpiecznie zwalniającą całą zarezerwowaną wcześniej pamięć, to łatwiej nad wszystkim zapanować.

Kiedy program kończy działanie, system operacyjny zwalnia wszystkie zarezerwowane przez niego zasoby, choć czasami nie odbywa się to natychmiast. Często spotykane rozwiązanie (zastosowałem je także w omawianym programie) polega na przerwaniu działania programu i zezwoleniu systemowi operacyjnemu na przeprowadzenie operacji porządkujących po wystąpieniu błędu.

Jak to zepsuć?

Omawiany tutaj program ma wiele miejsc, w których można go zepsuć. Wypróbuj kilka z nich, a także postaraj się znaleźć inne.

- Klasycznym sposobem jest usunięcie operacji sprawdzenia, co pozwoli na przekazywanie dowolnych danych. Na przykład usuń przeprowadzaną w wierszu 172. operację sprawdzenia, która uniemożliwia przekazanie dowolnego numeru rekordu.
- Możesz również spróbować uszkodzić plik danych. Otwórz go w edytorze tekstu, zmień losowo wybrane bajty, a następnie zapisz plik.
- Możesz także znaleźć sposoby na przekazanie nieprawidłowych argumentów podczas uruchamiania programu. Na przykład podanie od końca nazwy pliku i akcji spowoduje utworzenie pliku o nazwie akcji, a następnie wykonanie akcji na podstawie pierwszego znaku.
- W omawianym programie istnieje błąd związany z kiepskim projektem funkcji `strcpy()`. Przeczytaj nieco o wymienionej funkcji, a następnie sprawdź, co się stanie, gdy wielkość podanego imienia lub adresu przekroczy 512 bajtów. Popraw program przez wymuszenie użycia `'\0'` jako ostatniego znaku, aby zawsze był ustawiony niezależnie od wszystkiego (takie powinno być działanie funkcji `strcpy()`).
- W kolejnej sekcji zachęcam Cię do modyfikacji programu w taki sposób, aby tworzył bazy danych o dowolnej wielkości. Zobacz, jaką największą bazę danych może utworzyć ten program, zanim dojdzie do jego awarii ze względu na brak pamięci do zalokowania przez wywołanie `malloc()`.

Zadania dodatkowe

- Funkcja `die()` wymaga modyfikacji, aby możliwe było przekazanie zmiennej `conn`, co z kolei pozwoli na zamknięcie programu i przeprowadzenie operacji porządkujących.
- Zmień kod w taki sposób, aby akceptował parametry dla stałych `MAX_DATA` i `MAX_ROWS`, przechowywał je w strukturze `Database` i zapisywał w pliku, co umożliwi utworzenie bazy danych o dowolnej wielkości.
- Dodaj kolejne operacje możliwe do przeprowadzenia w bazie danych, na przykład `find`.
- Wyszukaj informacje o tym, jak język C przeprowadza pakowanie struktur, a następnie spróbuj ustalić, dlaczego plik ma taką, a nie inną wielkość. Sprawdź, czy możesz określić nową wielkość po dodaniu kolejnych elementów.
- Dodaj kolejne elementy do struktury `Address`, aby ułatwić przeprowadzanie operacji wyszukiwania.
- Utwórz skrypt powłoki przeznaczony do automatycznego testowania programu przez wydawanie poleceń w odpowiedniej kolejności. Podpowiedź: użyj `set -e` na początku skryptu powłoki, aby przerwać jego wykonywanie, gdy jakiekolwiek polecenie zawiera błąd.
- Spróbuj zmodyfikować program tak, aby używał pojedynczego elementu globalnego do obsługi połączenia z bazą danych. Jak wypada ta nowa wersja programu w porównaniu z poprzednią?
- Wyszukaj informacje o strukturze danych sterty i utwórz ją w ulubionym języku programowania. Następnie spróbuj to samo zrobić w C.

ĆWICZENIE 18

Wskaźniki do funkcji

W języku C funkcje są tak naprawdę wskaźnikami prowadzącymi do miejsca w kodzie, w którym istnieje pewien kod. Podobnie jak tworzyłeś wskaźniki do struktur, ciągów tekstowych i tablic, możesz utworzyć wskaźnik prowadzący do funkcji. Podstawowym przeznaczeniem dla tych wskaźników jest przekazywanie wywołań zwrotnych do innych funkcji lub też symulowanie klas i obiektów. W tym ćwiczeniu przygotujemy kilka wywołań zwrotnych, natomiast w kolejnym opracujemy prosty system obiektowy.

Format wskaźnika funkcji przedstawia się następująco:

```
int (*NAZWA_WSKAŹNIKA)(int a, int b)
```

Oto sposób pozwalający na zapamiętanie powyższego formatu:

- Utwórz standardową deklarację funkcji — `int dowolna_nazwa(int a, int b)`
- Opakuj nazwę funkcji składnią wskaźnika — `int (*dowolna_nazwa)(int a, int b)`
- Zmień nazwę na nazwę wskaźnika — `int (*nazwa_docelowa)(int a, int b)`

Kluczową kwestią do zapamiętania jest to, że po wykonaniu powyższych kroków nazwą *zmiennej* dla wskaźnika będzie *nazwa_docelowa* i możesz jej używać tak, jakby była funkcją. To jest bardzo podobne do sytuacji, w której wskaźniki do tablic mogą być używane jak wskazywane przez nie tablice. Wskaźniki do funkcji można więc wykorzystywać jak wskazywane przez nie funkcje, ale o innej nazwie.

```
int (*tester)(int a, int b) = sorted_order;
printf("TEST: %d oznacza to samo co %d\n", tester(2, 3), sorted_order(2, 3));
```

Takie podejście działa, nawet jeśli wskaźnik funkcji zwraca wskaźnik do czegoś innego:

- Zapisz — `char *cos_wspanialego(int niesamowity_poziom)`
- Opakuj — `char *(*cos_wspanialego)(int niesamowity_poziom)`
- Zmień nazwę — `char *(*wspaniala_funkcjonalnosc)(int niesamowity_poziom)`

Kolejnym problemem do rozwiązania dotyczącym wskaźników funkcji jest to, że trudno podać je jako parametry funkcji, na przykład gdy chcesz przekazać wywołanie zwrotne funkcji do innej funkcji. W takim przypadku rozwiązaniem będzie użycie `typedef`, czyli słowa kluczowego w C przeznaczonego do tworzenia nowych nazw dla innych, bardziej skomplikowanych typów.

Musisz jedynie umieścić słowo kluczowe `typedef` przed tą samą składnią wskaźnika funkcji, a następnie możesz wykorzystywać nazwę tak jak typ. Tego rodzaju podejście zostało zaprezentowane w poniższym fragmencie kodu.

Plik *ex18.c*:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

/** Nasz stary przyjaciel, czyli funkcja die() z ćwiczenia 17. */
void die(const char *message)
{
    if (errno) {
        perror(message);
    } else {
        printf("BŁĄD: %s\n", message);
    }

    exit(1);
}

// Polecenie typedef tworzy nieprawdziwy typ.
// W omawianym programie to typ dla wskaźnika funkcji.
typedef int (*compare_cb) (int a, int b);

/**
* Klasyczna funkcja sortowania bąbelkowego, która
* używa compare_cb do przeprowadzenia sortowania.
*/
int *bubble_sort(int *numbers, int count, compare_cb cmp)
{
    int temp = 0;
    int i = 0;
    int j = 0;
    int *target = malloc(count * sizeof(int));

    if (!target)
        die("Błąd pamięci.");

    memcpy(target, numbers, count * sizeof(int));

    for (i = 0; i < count; i++) {
        for (j = 0; j < count - 1; j++) {
            if (cmp(target[j], target[j + 1]) > 0) {
                temp = target[j + 1];
                target[j + 1] = target[j];
                target[j] = temp;
            }
        }
    }

    return target;
}

```

```
51 int sorted_order(int a, int b)
52 {
53     return a - b;
54 }
55
56 int reverse_order(int a, int b)
57 {
58     return b - a;
59 }
60
61 int strange_order(int a, int b)
62 {
63     if (a == 0 || b == 0) {
64         return 0;
65     } else {
66         return a % b;
67     }
68 }
69
70 /**
71 * Funkcja używana do sprawdzenia, czy elementy są sortowane prawidłowo.
72 * Odbywa się to przez posortowanie i wyświetlenie elementów.
73 */
74 void test_sorting(int *numbers, int count, compare_cb cmp)
75 {
76     int i = 0;
77     int *sorted = bubble_sort(numbers, count, cmp);
78
79     if (!sorted)
80         die("Nie udało się przeprowadzić sortowania.");
81
82     for (i = 0; i < count; i++) {
83         printf("%d ", sorted[i]);
84     }
85     printf("\n");
86
87     free(sorted);
88 }
89
90 int main(int argc, char *argv[])
91 {
92     if (argc < 2) die("UŻYCIE: ex18 4 3 1 5 6");
93
94     int count = argc - 1;
95     int i = 0;
96     char **inputs = argv + 1;
97
98     int *numbers = malloc(count * sizeof(int));
99     if (!numbers) die("Błąd pamięci.");
100
101     for (i = 0; i < count; i++) {
102         numbers[i] = atoi(inputs[i]);
103     }
```

```
104
105     test_sorting(numbers, count, sorted_order);
106     test_sorting(numbers, count, reverse_order);
107     test_sorting(numbers, count, strange_order);
108
109     free(numbers);
110
111     return 0;
112 }
```

W powyższym programie tworzymy algorytm sortowania dynamicznego, który może posortować tablicę liczb całkowitych za pomocą wywołania zwrotnego przeznaczonego do przeprowadzania porównania elementów. Poniżej przedstawiam dokładne omówienie działania programu.

ex18.c:1 – 4. Standardowe polecenia dołączające pliki nagłówkowe wymagane do wywołania wszystkich funkcji w programie.

ex18.c:7 – 16. Funkcja `die()` z poprzedniego ćwiczenia; będziemy ją wykorzystywać do sprawdzenia pod kątem błędów.

ex18.c:20. W tym miejscu użyliśmy słowa kluczowego `typedef`. Później użyjemy `compare_cb` jako jej typu, podobnie jak używamy `int` lub `char` w `bubble_sort()` i `test_sorting()`.

ex18.c:26 – 49. To jest implementacja sortowania bąbelkowego, która jest bardzo nieefektywnym sposobem na sortowanie liczb całkowitych. Oto omówienie zastosowanej implementacji:

ex18.c:26. Użyliśmy `typedef` dla `compare_cb` jako ostatniego parametru `cmp`. Teraz to jest funkcja zwracająca wynik porównania dwóch liczb całkowitych, potrzebny do przeprowadzenia sortowania.

ex18.c:28 – 34. Standardowy sposób utworzenia zmiennych na stosie oraz utworzenie w stercie nowej tablicy liczb całkowitych za pomocą wywołania `malloc()`. Upewnij się, że rozumiesz znaczenie polecenia `count *sizeof(int)`.

ex18.c:38. Zewnętrzna pętla sortowania bąbelkowego.

ex18.c:39. Wewnętrzna pętla sortowania bąbelkowego.

ex18.c:40. Teraz już można użyć wywołania zwrotnego `cmp()` podobnie jak zwykłej funkcji zamiast nazwy czegoś zdefiniowanego przez nas; jest to po prostu wskaźnik. Dzięki temu komponent wywołujący może przekazać cokolwiek, o ile będzie to dopasowane do sygnatury `compare_cb typedef`.

ex18.c:41 – 43. Rzeczywista operacja zamiany kolejności elementów, gdzie sortowanie bąbelkowe robi to, co powinno.

ex18.c:48. Zwrot nowo utworzonej i posortowanej tablicy wynikowej `target`.

ex18.c:51 – 68. Trzy różne wersje funkcji typu `compare_cb`, które muszą mieć dokładnie taką samą definicję jak utworzony wcześniej `typedef`. W przypadku niedopasowania definicji kompilator C wygeneruje komunikat o błędzie.

ex18.c:74 – 88. To jest sprawdzenie poprawności działania funkcji `bubble_sort()`. Możesz zobaczyć również, jak używamy `compare_cb` w celu przekazania do `bubble_sort()`, co demonstruje możliwość przekazywania tych wskaźników podobnie jak każdych innych.

ex18.c:90 – 103. Prosta funkcja `main()` przeprowadzająca konfigurację tablicy na podstawie liczb całkowitych przekazanych w powłoce. Następnie wywoływana jest funkcja `test_sorting()`.

ex18.c:105 – 107. Wreszcie możesz zobaczyć, jak używany jest wskaźnik funkcji `compare_cb`. W tym celu po prostu wywołujemy funkcję `test_sorting()`, ale przekazujemy nazwy `sorted_order`, `reverse_order` oraz `strange_order` jako funkcje do użycia. Następnie kompilator C odszukuje adresy wymienionych funkcji i konwertuje je na wskaźniki przeznaczone do użycia przez funkcję `test_sorting()`. Jeżeli spojrzysz na kod funkcji `test_sorting()`, to zobaczysz, że przekazuje ona wymienione wskaźniki do `bubble_sort()` i tak naprawdę nie wie, do czego są wykorzystywane. Kompilator jedynie ustala, że są dopasowane do prototypu `compare_cb` i powinny działać.

ex18.c:109. Ostatnim zadaniem jest zwolnienie pamięci zarezerwowanej dla utworzonej tablicy liczb.

Co powinieneś zobaczyć?

Uruchomienie programu jest proste, choć powinieneś wypróbować różne kombinacje liczb, a nawet innych znaków, aby zobaczyć, jaki będzie wynik działania programu.

Sesja dla ćwiczenia 18.:

```
$ make ex18
cc -Wall -g ex18.c -o ex18
$ ./ex18 4 1 7 3 2 0 8
0 1 2 3 4 7 8
8 7 4 3 2 1 0
3 4 2 7 1 0 8
$
```

Jak to zepsuć?

Zamierzam zachęcić Cię do zrobienia czegoś dziwnego w celu zepsucia omówionego programu. Ponieważ użyte wskaźniki są takie same jak każde inne, więc prowadzą do pewnych bloków pamięci. Język C pozwala na konwersję wskaźnika na inny, co umożliwia przetwarzanie danych na różne sposoby. Wprawdzie to z reguły nie jest konieczne, ale w celu pokazania, jak można doprowadzić do awarii programu, proszę Cię o dodanie poniższego kodu na końcu funkcji `test_sorting()`:

```
unsigned char *data = (unsigned char *)cmp;

for(i = 0; i < 25; i++) {
```

```
        printf("%02x:", data[i]);
    }

    printf("\n");
```

Powyższa pętla to rodzaj konwersji funkcji na ciąg tekstowy, a następnie wyświetlenie jego zawartości. Nie spowoduje to awarii programu, o ile procesor i system operacyjny nie mają problemów z wykonaniem tego rodzaju zadania. Po wyświetleniu zawartości posortowanej tablicy zobaczysz ciąg tekstowy składający się z liczb szesnastkowych, na przykład:

```
55:48:89:e5:89:7d:fc:89:75:f8:8b:55:fc:8b:45:
```

To powinien być niezmodyfikowany kod bajtowy samej funkcji. Powinieneś zobaczyć, że początek jest taki sam, a inna może być jedynie końcówka ciągu tekstowego. Istnieje prawdopodobieństwo, że powyższa pętla nie pobierze całej funkcji lub też pobierze zbyt wiele danych i zahaczy o fragment innego programu. Bez dokładniejszej analizy pozostaje to niewiadomą.

Zadania dodatkowe

- Przejdź do edytora szesnastkowego, otwórz w nim program *ex18*, a następnie odszukaj sekwencję liczb szesnastkowych będących początkiem wymienionej wcześniej funkcji. W ten sposób przekonasz się, czy możesz znaleźć funkcję w niezmodyfikowanym kodzie programu.
- Losowo wybierz inny element w edytorze szesnastkowym, a następnie zmień go. Ponownie uruchom program i zobacz, jaki będzie efekt wprowadzonej zmiany. Ciągi tekstowe są najłatwiejszymi elementami do zmiany.
- Przekaż nieprawidłową funkcję dla `compare_cb`, a następnie sprawdź, jakie komunikaty będą wyświetlone przez kompilator C.
- Przekaż wartość `NULL` i dokładnie obserwuj zachowanie programu. Następnie uruchom debuger i sprawdź, jakie będą wyświetlone komunikaty.
- Utwórz inny algorytm sortowania, a następnie zmień funkcję `test_sorting()` w taki sposób, aby pobierała *zarówno* nową funkcję sortowania, jak i wywołanie zwrotne porównania funkcji sortowania. Wykorzystaj funkcję `test_sorting()` do przetestowania obu algorytmów.

ĆWICZENIE 19

Opracowane przez Zeda wspaniałe makra debugowania

W języku C mamy do czynienia z regularnie pojawiającym się problemem, który zamierzam rozwiązać za pomocą zestawu samodzielnie opracowanych makr. Możesz mi podziękować później, gdy już przekonasz się, jak niezwykle fantastyczne są te makra. W tej chwili nie dostrzeżesz ich wyjątkowości i po prostu będziesz je wykorzystywał. Jednak pewnego dnia przyjdziesz do mnie i powiesz: „Zed, te makra debugowania są rewelacyjne. Zawdzięczam ci mojego pierworodnego, ponieważ oszczędziłeś mi dekadę bólu głowy i wiele razy uchroniłeś przed zabiciem się. Dziękuję ci, dobry człowieku, oto milion dolarów i oryginalny prototyp Snakeheada Telecastera podpisany przez Leo Fendera".

Tak, wspomniane makra są naprawdę wspaniałe.

Problem obsługi błędów w C

Obsługa błędów jest trudnym zadaniem w praktycznie każdym języku programowania. Istnieją także całe języki programowania, w których podjęto ogromny wysiłek mający na celu nawet uniknięcie koncepcji błędu. Z kolei w innych językach opracowano skomplikowane struktury kontrolne, takie jak wyjątki, przeznaczone do przekazywania błędów. Problem istnieje, ponieważ większość programistów przyjmuje założenie, że błędy się nie zdarzają, a ten optymizm ma wpływ na typy tworzonych i używanych języków.

Język C stawia czoła temu problemowi przez zwrot kodu błędu i przypisania zmiennej globalnej `errno` wartości, którą można później odczytać. W ten sposób powstaje pole dla skomplikowanego kodu, który istnieje tylko po to, aby sprawdzić, czy w innym kodzie utworzonym przez Ciebie istnieje błąd. Im więcej będziesz tworzyć kodu w języku C, tym częściej zaczniesz stosować wymieniony poniżej wzorzec:

- Wywołanie funkcji.
- Sprawdzenie, czy wartość zwrotna wskazuje na wystąpienie błędu (tę operację trzeba wykonać za każdym razem).
- Usunięcie wszystkich utworzonych dotąd zasobów.
- Wyświetlenie komunikatu o błędzie w nadziei, że okaże się pomocny.

Zastosowanie powyższego wzorca oznacza, że każde wywołanie funkcji (dokładnie tak, *każdej* funkcji) zawiera trzy lub cztery dodatkowe wiersze kodu tylko po to, aby upewnić się o jej działaniu. Oczywiście nie obejmuje to problemu usunięcia wszystkich wygenerowanych dotąd śmieci. Jeżeli program zawiera 10 różnych struktur, 3 pliki i połączenie z bazą danych, to masz 14 kolejnych wierszy kodu, w których może wystąpić błąd.

W przeszłości nie stanowiło to problemu, ponieważ tworzone programy w C robiły to, do czego były przeznaczone, natomiast po wystąpieniu błędu po prostu kończyły działanie.

Nie trzeba było więc zajmować się zwalnianiem zasobów, skoro system operacyjny zajmował się tym automatycznie po przerwaniu działania programu. Obecnie programy w C są przeznaczone do nieprzerwanego działania tygodniami, miesiącami, a nawet latami i muszą elegancko obsługiwać błędy pojawiające się z różnych źródeł. Serwer WWW po prostu nie może nagle zakończyć działania przy nawet najdrobniejszym błędzie. Ponadto nie możesz sobie pozwolić na wystąpienie sytuacji, w której na przykład utworzona przez Ciebie biblioteka prowadzi do awarii programu, w którym została zastosowana. To jest po prostu niewłaściwe.

W innych językach programowania problem próbuje się rozwiązać za pomocą wyjątków, ale one same mogą powodować problemy w C (podobnie jak w innych językach programowania). W języku C można mieć tylko jedną wartość zwrotną, natomiast wątki tworzą całe stosy wartości zwrotnych wraz z dowolnymi wartościami. Próba uporządkowania stosu wyjątku w C jest trudna, a na dodatek nie będzie on mógł być używany przez inne biblioteki.

Makra debugowania

Rozwiązaniem stosowanym przeze mnie od lat jest mały zestaw makr debugowania, w których zaimplementowałem podstawową logikę debugowania i system obsługi błędów dla C. Ten system jest niezwykle łatwy do opanowania, współpracuje z każdą biblioteką i powoduje, że kod w języku C jest znacznie bardziej niezawodny oraz przejrzysty.

Cel osiągnąłem przez wykorzystanie następującej konwencji: po wystąpieniu błędu funkcja przechodzi do etykiety `error:`, czyli fragmentu funkcji odpowiedzialnego za zwolnienie zasobów i zwrot kodu błędu. Można również użyć makra o nazwie `check()` do sprawdzenia kodu błędu, wyświetlenia komunikatu o błędzie, a następnie do przejścia do sekcji zajmującej się operacjami porządkującymi. Oczywiście wymienione powyżej operacje można połączyć z funkcjami zbierającymi dane, na których podstawie będą później wyświetlane użyteczne komunikaty dotyczące debugowania.

Poniżej przedstawiam pełną zawartość najbardziej fantastycznego zbioru błyskotliwości, jaki kiedykolwiek widziałeś.

Plik *dbg.h*:

```
#ifndef __dbg_h__
#define __dbg_h__

#include <stdio.h>
#include <errno.h>
#include <string.h>

#ifdef NDEBUG
#define debug(M, ...)
#else
#define debug(M, ...) fprintf(stderr, "DEBUG %s:%d: " M "\n",\
    __FILE__, __LINE__, ##__VA_ARGS__)
#endif

```

```
15 #define clean_errno() (errno == 0 ? "brak" : strerror(errno))
16
17 #define log_err(M, ...) fprintf(stderr,\
18     "[ERROR] (%s:%d: errno: %s) " M "\n", __FILE__, __LINE__,\
19     clean_errno(), ##__VA_ARGS__)
20
21 #define log_warn(M, ...) fprintf(stderr,\
22     "[WARN] (%s:%d: errno: %s) " M "\n",\
23     __FILE__, __LINE__, clean_errno(), ##__VA_ARGS__)
24
25 #define log_info(M, ...) fprintf(stderr, "[INFO] (%s:%d) " M "\n",\
26     __FILE__, __LINE__, ##__VA_ARGS__)
27
28 #define check(A, M, ...) if(!(A)) {\
29     log_err(M, ##__VA_ARGS__); errno=0; goto error; }
30 #define sentinel(M, ...) { log_err(M, ##__VA_ARGS__);\
31     errno=0; goto error; }
32
33 #define check_mem(A) check((A), "Brak pamięci.")
34
35 #define check_debug(A, M, ...) if(!(A)) { debug(M, ##__VA_ARGS__);\
36     errno=0; goto error; }
37
38 #endif
```

Tak, to jest ten wspaniały kod. Oto jego dokładne omówienie, niemalże wiersz po wierszu:

dbg.h:1 – 2. Standardowy sposób ochrony przed przypadkowym dwukrotnym dołączeniem pliku. Z takim podejściem spotkałeś się już w poprzednim ćwiczeniu.

dbg.h:4 – 6. Polecenia `include` dla funkcji wymaganych do działania makr.

dbg.h:8. Początek bloku `#ifdef` pozwalającego na ponowną kompilację programu, aby wszystkie komunikaty dotyczące debugowania zostały usunięte.

dbg.h:9. W przypadku kompilacji wraz ze zdefiniowaną opcją `NODEBUG` komunikaty pozostaną. Możesz zobaczyć, że w takim przypadku wywołanie `#define debug()` nie będzie niczym zastąpione (prawa strona pozostaje pusta).

dbg.h:10. Konstrukcja `#else` dopasowana do `#ifdef`.

dbg.h:11. Alternatywna sekcja `#define debug()`, która przekształca każde użycie `debug("format", arg1, arg2)` na wywołanie `fprintf()` do `stderr`. Wielu programistów C nie wie o możliwości utworzenia makr, które będą działały podobnie jak wywołanie `printf()` i pobierały zmienną liczbę argumentów. Pewne kompilatory C (tak naprawdę CPP) nie obsługują takiego podejścia, natomiast pozostałe tak. Najbardziej interesujące jest tutaj użycie `##__VA_ARGS__`, oznaczające: „Umieść w tym miejscu wszystko, co jest przeznaczone dla argumentów dodatkowych". Ponadto zwróć uwagę na użycie `__FILE__` i `__LINE__` w celu pobrania aktualnej nazwy pliku i wiersza do wykorzystania w komunikacie dotyczącym debugowania. To jest *niezwykle* użyteczne.

dbg.h:13. W tym miejscu kończy się sekcja `#ifdef`.

dbg.h:15. To jest makro `clean_errno()` używane w innych makrach w celu otrzymania bezpiecznej, czytelnej wersji zmiennej `errno`. Dziwna składnia w środku to operator trójargumentowy, o którego działaniu nieco więcej dowiesz się w dalszej części książki.

dbg.h:17 – 26. Makra `log_err()`, `log_warn()` i `log_info()` służące do rejestracji komunikatów przeznaczonych dla użytkownika końcowego. Ich działanie jest podobne do `debug()`, ale nie mogą być skompilowane.

dbg.h:28. To jest najlepsze z makr, `check()`. Służy do sprawdzenia, czy warunek A jest prawdziwy. Jeśli nie jest, to rejestruje błąd `M` (wraz ze zmienną liczbą argumentów dla `log_err()`), a następnie przechodzi do sekcji `error:`, funkcji wykorzystywanej w celu przeprowadzenia operacji porządkujących.

dbg.h:31. Drugie z najlepszych makr, `sentinel()`. Służy do umieszczenia w dowolnej części funkcji, która nie powinna być wykonana. Jeżeli jednak będzie, to wyświetla komunikat o błędzie i powoduje przejście do sekcji `error:`. Wykorzystujemy to makro w konstrukcjach `if` i `switch` do przechwycenia przypadków, które nie powinny się zdarzyć, na przykład `default:`.

dbg.h:34. Makro skrótu o nazwie `check_mem()`, które sprawdza poprawność wskaźnika. Jeżeli wskaźnik jest nieprawidłowy, makro generuje komunikat o błędzie: „Brak pamięci".

dbg.h:36. Alternatywne makro o nazwie `check_debug()`, które sprawdza i obsługuje błąd. Jeżeli jednak błąd należy do powszechnie występujących, to makro nie będzie o nim informować. Do wyświetlenia komunikatu tutaj używamy makra `debug()` zamiast `log_error()`. Dlatego też po zdefiniowaniu `NDEBUG` operacja sprawdzenia nadal będzie przeprowadzana, przejścia do sekcji błędu nie będzie i komunikat nie zostanie wyświetlony.

Użycie dbg.h

Poniżej przedstawiłem przykład użycia pliku *dbg.h* w małym programie, który w rzeczywistości nie robi nic użytecznego i ma na celu jedynie zademonstrowanie przykładu zastosowania omawianych makr. Ponieważ makra będziemy od teraz wykorzystywać we wszystkich tworzonych programach, upewnij się, że dobrze rozumiesz sposób ich użycia.

Plik *ex19.c*:

```
1  #include "dbg.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  void test_debug()
6  {
7      // Zwróć uwagę na brak konieczności użycia \n.
8      debug("Mam brązowe włosy.");
9
10     // Przekazanie argumentów, podobnie jak w wywołaniu printf().
```

```
11      debug("Mam %d lat.", 37);
12  }
13
14  void test_log_err()
15  {
16      log_err("Jestem przekonany, że wszystko jest zepsute.");
17      log_err("Mamy %d problemów w %s.", 0, "kosmosiie.");
18  }
19
20  void test_log_warn()
21  {
22      log_warn("Możesz to bezpiecznie zignorować.");
23      log_warn("Rozważ przeanalizowanie: %s.", "/etc/passwd");
24  }
25
26  void test_log_info()
27  {
28      log_info("Cóż, zrobiłem coś prozaicznego.");
29      log_info("To zdarza się %f razy dziennie.", 1.3f);
30  }
31
32  int test_check(char *file_name)
33  {
34      FILE *input = NULL;
35      char *block = NULL;
36
37      block = malloc(100);
38      check_mem(block); // To powinno działać.
39
40      input = fopen(file_name, "r");
41      check(input, "Nie udało się otworzyć pliku %s.", file_name);
42
43      free(block);
44      fclose(input);
45      return 0;
46
47  error:
48      if (block) free(block);
49      if (input) fclose(input);
50      return -1;
51  }
52
53  int test_sentinel(int code)
54  {
55      char *temp = malloc(100);
56      check_mem(temp);
57
58      switch (code) {
59          case 1:
60              log_info("To działa.");
61              break;
62          default:
63              sentinel("To nie powinno być uruchomione.");
```

```
64      }
65
66      free(temp);
67      return 0;
68
69  error:
70      if (temp)
71          free(temp);
72      return -1;
73  }
74
75  int test_check_mem()
76  {
77      char *test = NULL;
78      check_mem(test);
79
80      free(test);
81      return 1;
82
83  error:
84      return -1;
85  }
86
87  int test_check_debug()
88  {
89      int i = 0;
90      check_debug(i != 0, "Ups, mam wartość 0.");
91
92      return 0;
93  error:
94      return -1;
95  }
96
97  int main(int argc, char *argv[])
98  {
99      check(argc == 2, "Wymagany jest argument.");
100
101     test_debug();
102     test_log_err();
103     test_log_warn();
104     test_log_info();
105
106     check(test_check("ex19.c") == 0, "nie udało się z powodu ex19.c");
107     check(test_check(argv[1]) == -1, "nie udało się z powodu argv");
108     check(test_sentinel(1) == 0, "niepowodzenie makra test_sentinel.");
109     check(test_sentinel(100) == -1, "niepowodzenie makra test_sentinel.");
110     check(test_check_mem() == -1, "niepowodzenie makra test_check_mem.");
111     check(test_check_debug() == -1, "niepowodzenie makra test_check_debug.");
112
113     return 0;
114
115 error:
116     return 1;
117 }
```

Zwróć uwagę na sposób użycia makra `check()`. W przypadku wartości `false` następuje przejście do sekcji `error:` w celu przeprowadzenia operacji porządkujących. Te wiersze można odczytać następująco: „Sprawdź, czy A jest prawdą; jeśli nie, powiedz `M` i przejdź do wskazanego miejsca".

Co powinieneś zobaczyć?

Po uruchomieniu programu podaj dowolny fikcyjny pierwszy argument, a otrzymasz pokazane poniżej dane wyjściowe.

Sesja dla ćwiczenia 19.:

```
$ make ex19
cc -Wall -g -DNDEBUG ex19.c -o ex19
$ ./ex19 test
[ERROR] (ex19.c:16: errno: brak) Jestem przekonany, że wszystko jest zepsute.
[ERROR] (ex19.c:17: errno: brak) Mamy 0 problemów w kosmosie.
[WARN] (ex19.c:22: errno: brak) Możesz to bezpiecznie zignorować.
[WARN] (ex19.c:23: errno: brak) Rozważ przeanalizowanie: /etc/passwd.
[INFO] (ex19.c:28) Cóż, zrobiłem coś prozaicznego.
[INFO] (ex19.c:29) To zdarza się 1.300000 razy dziennie.
[ERROR] (ex19.c:38: errno: brak pliku lub katalogu) Nie udało się otworzyć pliku test.
[INFO] (ex19.c:57) To działa.
[ERROR] (ex19.c:60: errno: brak) To nie powinno być uruchomione.
[ERROR] (ex19.c:74: errno: brak) Brak pamięci.
```

Czy widzisz, że makra podają dokładny numer wiersza, w którym wystąpiło niepowodzenie? W ten sposób możesz uniknąć konieczności późniejszego poświęcenia godzin na znalezienie źródła błędu. Ponadto zwróć uwagę na wyświetlenie komunikatu o błędzie po przypisaniu wartości zmiennej `errno`. To również może zaoszczędzić Ci wiele czasu podczas debugowania.

W jaki sposób CPP obsługuje makra?

Przechodzimy teraz do krótkiego wprowadzenia, pokazującego, jak faktycznie można używać makr w kompilatorze CPP. W tym celu podzielę na fragmenty najbardziej skomplikowane makro w pliku *dbg.h* i przedstawię przykład jego uruchomienia w CPP, abyś mógł zobaczyć, jak naprawdę wygląda cały proces.

Wyobraź sobie, że mamy funkcję o nazwie `dosomething()`, która zwraca wartość `0` w przypadku sukcesu oraz `-1` w przypadku niepowodzenia. W trakcie każdego wywołania funkcji `dosomething()` trzeba przeprowadzić sprawdzenie pod kątem kodu błędu, a więc tworzę kod w postaci:

```
int rc = dosomething();

if(rc != 0) {
    fprintf(stderr, "Wystąpił błąd: %s\n", strerror());
    goto error;
}
```

Moim celem jest to, aby kompilator CPP hermetyzował tę konstrukcję `if` w znacznie czytelniejszym i łatwiejszym do zapamiętania wierszu kodu. Można powiedzieć, że chcę skorzystać z podejścia opartego na makrze `check()` przedstawionym w pliku *dbg.h*:

```
int rc = dosomething();
check(rc == 0, "Wystąpił błąd.");
```

To jest *znacznie* czytelniejsze rozwiązanie i dokładnie przedstawia zachodzące zdarzenia: sprawdzenie, czy funkcja została wykonana zgodnie z oczekiwaniami; jeśli nie, to następuje zgłoszenie błędu. Zastosowanie takiego podejścia wymaga pewnych sztuczek w CPP, które czynią z tego kompilatora użyteczne narzędzie generowania kodu. Ponownie spójrz na makra `check()` i `log_err()`:

```
#define log_err(M, ...) fprintf(stderr,\
    "[ERROR] (%s:%d: errno: %s) " M "\n", __FILE__, __LINE__,\
    clean_errno(), ##__VA_ARGS__)
#define check(A, M, ...) if(!(A)) {\
    log_err(M, ##__VA_ARGS__); errno=0; goto error; }
```

Pierwsze makro, `log_err()`, jest prostsze. Jego zadanie polega na zastąpieniu omawianego makra wywołaniem funkcji `fprintf()` do `stderr`. Jedyna sztuczka w tym makrze polega na użyciu `...` w definicji `log_err(M, ...)`. Dzięki temu możesz przekazać zmienną liczbę argumentów do makra, tak więc możesz przekazać przeznaczone dla wywołania `fprintf()`. Powstaje tutaj pytanie, w jaki sposób te argumenty zostaną wstrzyknięte do wywołania `fprintf()`. Spójrz na koniec `##__VA_ARGS__`, gdzie kompilator CPP otrzymuje polecenia pobrania argumentów podanych w miejscu `...`, a następnie wstrzyknięcia ich jako części wywołania `fprintf()`. Możesz więc wydać polecenie podobne do poniższego:

```
log_err("Wiek: %d, imię: %s", age, name);
```

Argumenty `age` i `name` to fragment definicji `...`. W trakcie wykonywania makra zostaną wstrzyknięte do danych wyjściowych funkcji `fprintf()`:

```
fprintf(stderr, "[ERROR] (%s:%d: errno: %s) Wiek %d: imię %d\n",
    __FILE__, __LINE__, clean_errno(), age, name);
```

Czy widzisz argumenty `age` i `name` na końcu? W taki właśnie sposób `...` i `##__VA_ARGS__` współpracują ze sobą, co doskonale sprawdza się w makrach wywołujących inne makra ze zmienną liczbą argumentów. Spójrz teraz na makro `check()` i zobacz, jak wywołuje `log_err()`, choć makro `check()` używa *również* `...` i `##__VA_ARGS__` do przeprowadzenia tego wywołania. W ten sposób można przekazać do makra `check()` pełne ciągi tekstowe formatowania w stylu `printf()`, które następnie zostaną przekazane do `log_err()` — oba makra działają jak `printf()`.

Kolejną rzeczą do przeanalizowania jest przygotowanie przez makro `check()` konstrukcji `if` przeznaczonej do sprawdzenia pod kątem błędu. Jeżeli pozbędziemy się fragmentu odpowiedzialnego za użycie `log_err()`, to otrzymamy:

```
if(!(A)) { errno=0; goto error; }
```

Powyższe polecenie ma następujące znaczenie: „Jeśli A przyjmuje wartość fałsz, to usuń zawartość zmiennej `errno` i przejdź do sekcji `error`". Makro `check()` zostaje zastąpione konstrukcją `if`, więc jeśli ręcznie rozwiniemy makro `check(rc == 0, "Wystąpił błąd.")`, to otrzymamy poniższy kod:

```
if(!(rc == 0)) {
    log_err("Wystąpił błąd.");
    errno=0;
    goto error;
}
```

Dzięki wykonaniu zawartości tych dwóch makr kompilator CPP zastępuje poszczególne makro rozszerzoną wersją jego definicji, a na dodatek będzie to robić *rekurencyjnie*, aż do rozwinięcia wszystkich makr w makrach. W ten sposób kompilator CPP można uznać za oparty na rekurencji system szablonów, o czym wspomniałem już wcześniej. Potęga omówionego rozwiązania wynika z możliwości generowania całych bloków parametryzowanego kodu, a więc staje się naprawdę użytecznym narzędziem przeznaczonym do generowania kodu.

W tym miejscu może pojawić się pytanie, dlaczego zamiast przedstawionego rozwiązania nie można używać funkcji takiej jak `die()`. Powód jest prosty: chcemy otrzymać informacje o pliku i numerze wiersza, którego wykonanie zakończyło się niepowodzeniem, a następnie przejść do operacji zajmującej się obsługą danego błędu. Jeżeli zrobisz to wewnątrz funkcji, nie otrzymasz numeru wiersza, w którym błąd faktycznie wystąpił, a przejście do procedury obsługi stanie się znacznie bardziej skomplikowane.

Kolejnym powodem jest pozostająca konieczność utworzenia konstrukcji `if`, która będzie przypominała wszystkie pozostałe konstrukcje `if` w kodzie i nie będzie jasno wskazywała, że odpowiada za sprawdzenie pod kątem błędu. Dzięki opakowaniu konstrukcji `if` makrem o nazwie `check()` przeznaczenie tego fragmentu kodu staje się wyraźne i jednocześnie nie jest częścią głównego przepływu działania programu.

Warto jeszcze wspomnieć o oferowanej przez CPP możliwości *warunkowej kompilacji* fragmentów kodu. Dlatego też możesz mieć kod istniejący tylko podczas kompilacji wersji programu przeznaczonej do debugowania. Tę możliwość wyraźnie widać w pliku *dbg.h*, gdzie makro `debug()` ma uwzględnioną zawartość tylko wtedy, gdy kompilator o nią poprosi. Bez takiej możliwości mielibyśmy zmarnowaną konstrukcję `if`, sprawdzającą, czy program działa w trybie debugowania, a ponadto marnowalibyśmy zasoby procesora na przeprowadzanie operacji sprawdzenia, gdy nie ma żadnej wartości do przeanalizowania.

Zadania dodatkowe

- Na początku pliku umieść polecenie `#define NDEBUG` i sprawdź, czy wszystkie komunikaty z procesu debugowania faktycznie zniknęły.
- Wycofaj poprzednią zmianę, a następnie dodaj `-DNDEBUG` do `CFLAGS` na początku pliku *Makefile* i ponownie skompiluj ten sam program.
- Zmodyfikuj operację rejestracji danych, aby oprócz nazwy pliku i numeru wiersza zawierały także nazwę funkcji, w której wystąpił błąd.

ĆWICZENIE 20

Zaawansowane techniki debugowania

W poprzednim ćwiczeniu poznałeś opracowane przeze mnie doskonałe makra debugowania i zobaczyłeś, jak można je wykorzystywać. Podczas debugowania kodu makra `debug()` używam niemal wyłącznie do przeanalizowania danej sytuacji i odszukania źródła problemu. W tym ćwiczeniu zaprezentuję podstawy zastosowania GDB do przeanalizowania prostego programu, który pozostaje uruchomiony i nie kończy działania. Dowiesz się, jak wykorzystać GDB w celu dołączenia debugera do uruchomionego procesu, jak zatrzymać debuger i jak zobaczyć, co się dzieje w trakcie działania programu. Na koniec przedstawię kilka drobnych podpowiedzi i sztuczek, które można wykorzystać podczas pracy z GDB.

Klip przygotowany dla tego ćwiczenia to kolejne wideo, w którym koncentruję się na pokazaniu zaawansowanych sztuczek w stosowanej przeze mnie technice debugowania. Przedstawione poniżej informacje stanowią rozwinięcie materiału pokazanego w wideo, więc najpierw obejrzyj klip. Debugowanie będzie znacznie łatwiejsze do opanowania, gdy zaczniesz od obejrzenia materiału wideo.

Użycie makra debug() kontra GDB

Debugowanie przeprowadzam głównie w stylu „metody naukowej": zaczynam od możliwych przyczyn problemu, a następnie eliminuję je lub udowadniam, że faktycznie mogą być źródłem problemu. Problem, jaki wielu programistów ma z tego rodzaju podejściem, polega na błędnym przekonaniu, że może ono spowolnić ich pracę. Panikują i rzucają się do działań mających na celu usunięcie błędu, ale w tym pośpiechu zupełnie nie dostrzegają faktu, że podejmują chaotyczne działania i nie zbierają naprawdę użytecznych informacji. Zauważyłem, że rejestrowanie danych (i wyświetlanie komunikatów debugera) zmusza mnie do naukowego podejścia do rozwiązania problemów i w większości sytuacji po prostu ułatwia zebranie odpowiednich informacji.

Ponadto istnieją jeszcze wymienione poniżej powody, dla których komunikaty debugera wykorzystuję jako podstawowe informacje w trakcie usuwania błędów:

- Możliwość wyświetlenia całego stosu wywołań programu wraz z wartościami zmiennych debugowania, co pozwala ustalić, co tak naprawdę poszło niezgodnie z oczekiwaniami. W przypadku GDB musisz umieścić polecenia `watch()` i `debug()` we wszystkich miejscach, z których chcesz otrzymać informacje. Trudno jest uzyskać niezawodne informacje, wykonując program w taki sposób.
- Komentarze wyświetlane w trakcie debugowania mogą pozostać w kodzie i kiedy zachodzi potrzeba ich wyświetlenia, wystarczy po prostu ponownie skompilować program. W przypadku GDB trzeba te same informacje skonfigurować unikatowo dla każdego szukanego problemu.

- Znacznie łatwiej jest włączyć rejestrowanie danych w serwerze, który nie działa prawidłowo, a następnie przeanalizować dzienniki zdarzeń i sprawdzić, co się dzieje. Administratorzy systemów wiedzą, jak zajmować się obsługą rejestracji danych, ale zwykle nie wiedzą, jak używać GDB.
- Wyświetlanie informacji jest po prostu łatwiejsze. Debugery zawsze są ograniczone i cudaczne w obrębie ich własnych dziwacznych interfejsów i niespójności. Nie ma nic skomplikowanego w debugowaniu za pomocą wywołania `debug("Czy w po ↪rządku? %d", my_stuff);`.
- Kiedy przygotowujesz komunikaty debugowania w celu wykrycia problemu, tak naprawdę jesteś zmuszony do analizy kodu i wykorzystania metody naukowej. Debugowanie możesz więc potraktować następująco: „Przyjmuję hipotezę, że tutaj znajduje się źródło problemu". Następnie podczas przeprowadzania debugowania sprawdzasz tę hipotezę i jeśli kod działa prawidłowo, przechodzisz do kolejnego fragmentu, który może powodować błąd. Może się wydawać, że tego rodzaju podejście zabiera dużo czasu, ale w rzeczywistości będzie szybsze, ponieważ przeprowadzasz proces zróżnicowanej diagnostyki i eliminujesz potencjalne źródła błędów aż do chwili znalezienia faktycznego błędu.
- Wyświetlanie informacji podczas debugowania sprawdza się lepiej w przypadku stosowania testów jednostkowych. Wywołania `debug()` można skompilować i kiedy test jednostkowy kończy się niepowodzeniem, w dowolnej chwili można przeanalizować dzienniki zdarzeń. W przypadku GDB konieczne jest ponowne wykonanie testu w GDB, a następnie za pomocą narzędzi GDB ustalenie, co tak naprawdę się dzieje.

Pomimo wszystkich zalet makra `debug()` i jego przewagi nad GDB nadal używam GDB w kilku sytuacjach. Uważam, że należy stosować wszelkie narzędzia, które pomogą w wykonaniu danej pracy. Czasami wystarczy tylko podłączyć się do nieprawidłowo działającego programu i nieco poeksperymentować. Być może masz serwer, który ciągle ulega awarii i tylko za pomocą jego podstawowych plików można poznać przyczynę kłopotów. W takich oraz w kilku innych sytuacjach użycie GDB jest odpowiednim podejściem. Zawsze warto mieć pod ręką jak najwięcej narzędzi, które pomogą w rozwiązaniu problemów.

Poniżej przedstawiłem krótkie omówienie, kiedy używam poszczególnych narzędzi debugowania: GDB, Valgrind i makra `debug()`.

- Narzędzie Valgrind wykorzystuję do wychwytywania wszystkich błędów pamięci. W tym zakresie korzystam z GDB tylko wtedy, gdy Valgrind sprawia problemy lub jeśli jego użycie spowodowałoby zbytnie spowolnienie działania programu.
- Makro `debug()` i generowane przez nie komunikaty wykorzystuję do diagnozowania i usuwania problemów związanych z logiką bądź z użyciem programu. To będzie około 90% problemów po rozpoczęciu pracy narzędzia Valgrind.
- GDB wykorzystuję w pozostałych, dziwnych przypadkach, a także w sytuacjach kryzysowych w celu zebrania informacji. Jeżeli narzędzie Valgrind nic nie wykrywa i nie mogę nawet wyświetlić potrzebnych mi informacji, to uruchamiam GDB i zaczynam szukać. W takich przypadkach użycie GDB sprowadza się całkowicie

do zbierania informacji. Gdy już mam pewną koncepcję dotyczącą problemu, przystępuję do utworzenia testu jednostkowego dla przyczyny błędu, a następnie generuję komunikaty, aby dowiedzieć się, dlaczego dany błąd w ogóle wystąpił.

Strategia debugowania

Ten proces sprawdza się z praktycznie każdą używaną techniką debugowania. Zamierzam omówić ją w kategoriach wykorzystania GDB, ponieważ programiści mają tendencję do pomijania tego procesu podczas użycia debugerów. Zastosuj ten proces dla każdego błędu, poza tymi najtrudniejszymi.

- Pracę rozpocznij od niewielkiego pliku tekstowego o nazwie *notes.txt*, który możesz wykorzystywać na notatki dotyczące idei, błędów, problemów itd.
- Zanim zaczniesz używać GDB, zapisz informacje o błędzie, który zamierzasz usunąć, oraz o jego ewentualnych przyczynach.
- Dla każdego błędu zapisz nazwy plików i funkcji, o których sądzisz, że mogą być źródłem problemu. Ewentualnie zapisz, że nie potrafisz ustalić źródła problemu.
- Teraz rozpocznij sesję z GDB, wybierz pierwszą potencjalną przyczynę błędu z podanymi nazwami pliku i funkcji, a następnie ustaw tam punkt kontrolny.
- Użyj GDB do uruchomienia programu i spróbuj potwierdzić, czy wskazana przyczyna faktycznie jest źródłem błędu. Najlepszym rozwiązaniem jest próba użycia polecenia `set` do łatwego poprawienia programu lub natychmiastowego wywołania błędu.
- Jeżeli to nie jest rzeczywista przyczyna błędu, zanotuj to w pliku *notes.txt* i nie zapomnij dodać, dlaczego wyeliminowałeś daną możliwość. Przejdź do kolejnej przyczyny najłatwiejszej do zidentyfikowania podczas debugowania i dodawaj kolejne informacje.

Jeżeli jeszcze tego nie zauważyłeś, to podpowiadam, że to w zasadzie jest przykład metody naukowej. Przygotowałeś zbiór hipotez, a później za pomocą debugowania potwierdziłeś lub wykluczyłeś je. W ten sposób zyskujesz głębsze spojrzenie na możliwe przyczyny błędu i wreszcie je odnajdziesz. Tego rodzaju proces może pomóc w uniknięciu wielokrotnej analizy tych samych przyczyn po ich wcześniejszym wykluczeniu.

Przedstawiony proces można zastosować także w połączeniu z wyświetlaniem informacji dotyczących debugowania, ale jedyna różnica polega wówczas na zapisywaniu hipotez w kodzie źródłowym, a nie w pliku *notes.txt*. W ten sposób zostajesz zmuszony do zastosowania metody naukowej podczas eliminowania błędów, ponieważ hipotezy musisz zapisać w postaci poleceń wyświetlających komunikaty debugera.

Zadania dodatkowe

- Znajdź debuger graficzny, a następnie porównaj jego użycie ze zwykłym GDB. Debugery graficzne bywają użyteczne, gdy analizowany program znajduje się na komputerze lokalnym. Natomiast okazują się bezużyteczne, jeśli trzeba przeprowadzić debugowanie programu na serwerze.
- W systemie operacyjnym można włączyć opcję tworzenia zrzutów zawartości pamięci w przypadku awarii programu. W ten sposób otrzymujesz plik (tak zwany *core dump*) z zapisem stanu programu w chwili jego awarii. Dzięki analizie pliku dowiesz się, co się zdarzyło w chwili awarii i jaka była jej przyczyna. Zmodyfikuj program w pliku *ex18.c* tak, aby ulegał awarii po kilku iteracjach, a następnie spróbuj utworzyć plik typu *core dump* i przeanalizować go.

ĆWICZENIE 21

Zaawansowane typy danych i kontrola przepływu

To ćwiczenie można potraktować jako pełne kompendium wiedzy o oferowanych przez język C i dostępnych do użycia typach danych oraz strukturach kontroli przepływu. To jest rodzaj przewodnika i nie musisz wprowadzać żadnego kodu. Zachęcam Cię do zapamiętania pewnych informacji przez utworzenie kart, co pozwoli Ci na przyswojenie najważniejszych koncepcji.

Aby materiał przedstawiony w ćwiczeniu był użyteczny, powinieneś poświęcić przynajmniej tydzień na analizę treści i uzupełnienie wszystkich zagadnień, które mogły zostać tutaj pominięte. Zapiszesz znaczenie poszczególnych elementów, a następnie utworzysz program potwierdzający Twoje przypuszczenia.

Dostępne typy danych

Typ	Opis
`int`	Przechowuje liczbę całkowitą, domyślnie o wielkości 32 bitów.
`double`	Przechowuje dużą liczbę zmiennoprzecinkową.
`float`	Przechowuje mniejszą liczbę zmiennoprzecinkową.
`char`	Przechowuje jednobajtowy znak.
`void`	Wskazuje na „brak typu" i jest używany do określenia, że funkcja nic nie zwraca. Innymi słowy wskaźnik nie ma typu, na przykład `void *cokolwiek`.
`enum`	Typ wyliczeniowy, który działa jako liczba całkowita, jest konwertowany na jej postać, ale pozwala na nadanie nazw symbolicznych zbiorom. Niektóre kompilatory mogą generować ostrzeżenia, gdy w konstrukcji `switch` nie zostaną pokryte wszystkie elementy typu wyliczeniowego.

Modyfikatory typu

Modyfikator	Opis
`unsigned`	Zmienia typ w taki sposób, aby nie były używane liczby ujemne. Otrzymujesz większy zakres liczb dodatnich, ale żadna wartość nie może być mniejsza niż `0`.
`signed`	Otrzymujesz zakres liczb dodatnich i ujemnych. Oznacza to zmniejszenie o połowę zakresu liczb dodatnich w celu otrzymania takiego samego zakresu liczb ujemnych.
`long`	Pozwala na użycie większego magazynu dla typu, aby mogły być przechowywane większe liczby. To zwykle oznacza podwojenie aktualnej wielkości typu.
`short`	Pozwala na użycie mniejszego magazynu dla typu. Oznacza to przechowywanie mniejszych liczb, które zajmują połowę aktualnej wielkości typu.

Kwalifikatory typów

Kwalifikator	Opis
`const`	Wskazuje, że zmienna nie ulega zmianie po inicjalizacji.
`volatile`	Wskazuje, że kompilator nie powinien próbować niczego odgadywać i nie powinien przeprowadzać żadnych przekombinowanych optymalizacji. Z kwalifikatora tego musisz skorzystać z reguły tylko wtedy, gdy naprawdę wykonujesz dziwne operacje z użyciem zmiennych.
`register`	Wymusza na kompilatorze przechowywanie zmiennej w rejestrze, choć kompilator może to zignorować. Obecne kompilatory znacznie lepiej radzą sobie z ustaleniem, gdzie najlepiej umieścić zmienne. Dlatego też używaj tego kwalifikatora tylko wtedy, gdy chcesz sprawdzić rzeczywistą poprawę wydajności działania programu.

Konwersja typu

Język C używa pewnego rodzaju mechanizmu promowania — analizuje dwa operandy znajdujące się po którejkolwiek stronie wyrażenia i promuje mniejszy, aby został dopasowany do większego przed przeprowadzeniem operacji. Jeżeli po jednej stronie wyrażenia mamy listę, to przed wykonaniem operacji druga również zostanie skonwertowana na listę. Promowanie odbywa się w następującej kolejności:

1. `long double`
2. `double`
3. `float`
4. `int` (jedynie `char` i `short int`)
5. `long`

Jeżeli samodzielnie chcesz ustalić, jak w danych wyrażeniu zostanie przeprowadzona konwersja, to nie pozostawiaj tego zadania kompilatorowi. Wykorzystaj wyraźne operacje rzutowania, aby otrzymać dokładnie taki wynik, jakiego oczekujesz. Na przykład jeśli masz wyrażenie:

```
long + char - int * double
```

to zamiast sprawdzać, czy będzie poprawnie skonwertowane na typ `double`, lepiej użyj rzutowania:

```
(double)long - (double)char - (double)int * double
```

Umieszczenie żądanego typu w nawiasie przed nazwą zmiennej pozwala na wymuszenie zastosowania wskazanego typu. Jednak trzeba koniecznie pamiętać o pewnej zasadzie: *zawsze promuj typ, nigdy nie degraduj*. Dlatego też nie rzutuj typu `long` na `char`, o ile nie wiesz, co robisz.

Wielkość typu

Plik nagłówkowy *stdint.h* definiuje zbiór zarówno `typedef` dla dokładnych wielkości liczb całkowitych, jak i makr dla wielkości wszystkich typów. Z powodu zachowanej spójności takie rozwiązanie jest znacznie łatwiejsze w użyciu niż wcześniej stosowany plik *limits.h*. Poniżej wymieniłem zdefiniowane typy.

Typ	Opis
`int8_t`	8-bitowa liczba całkowita ze znakiem
`uint8_t`	8-bitowa liczba całkowita bez znaku
`int16_t`	16-bitowa liczba całkowita ze znakiem
`uint16_t`	16-bitowa liczba całkowita bez znaku
`int32_t`	32-bitowa liczba całkowita ze znakiem
`uint32_t`	32-bitowa liczba całkowita bez znaku
`int64_t`	64-bitowa liczba całkowita ze znakiem
`uint64_t`	64-bitowa liczba całkowita bez znaku

Zastosowany został wzorzec w postaci `(u)int(`*`BITY`*`)_t`, gdzie *u* oznacza `unsigned` (bez znaku), a *BITY* wskazuje ilość bitów dla danej liczby. Ten wzorzec jest następnie powtarzany w makrach zwracających wartości maksymalne dla poszczególnych typów:

- `INT(WN)_MAX`. Maksymalna wartość dodatniej liczby całkowitej ze znakiem o podanej liczbie bitów (*N*), na przykład `INT16_MAX`.
- `INT(N)_MIN`. Minimalna wartość ujemnej liczby całkowitej ze znakiem o podanej liczbie bitów (*N*).
- `UINT(N)_MAX`. Maksymalna wartość dodatniej liczby całkowitej bez znaku o podanej liczbie bitów (*N*). Ponieważ jest to liczba bez znaku, więc minimum wynosi `0` i nie może być liczbą ujemną.

OSTRZEŻENIE Zachowaj ostrożność! Nie szukaj dosłownej definicji `INT(N)_MAX` w żadnym pliku nagłówkowym. W tym miejscu wykorzystałem `(N)` w charakterze miejsca zarezerwowanego dla dowolnej, aktualnie obsługiwanej liczby bitów przez Twoją platformę. Dlatego też `(N)` może przyjąć wartość `8`, `16`, `32`, `64`, a być może nawet `128`. Zdecydowałem się na taki zapis w ćwiczeniu, aby uniknąć konieczności wypisania każdej możliwej kombinacji.

Plik *stdint.h* zawiera także makra dotyczące wielkości typu `size_t`, liczb całkowitych wystarczająco dużych do przechowywania wskaźników, a także inne makra definiujące przydatne wielkości. Kompilatory powinny posiadać przynajmniej wymienione typy, mogą również pozwalać na obsługę innych, znacznie większych typów.

Poniżej wymieniłem pełną listę typów, która powinna znajdować się w pliku *stdint.h*.

Typ	Opis
`int_least(N)_t`	Przechowuje co najmniej (*N*) bitów.
`uint_least(N)_t`	Przechowuje co najmniej (*N*) bitów bez znaku.
`INT_LEAST(N)_MAX`	Maksymalna wartość dopasowanego typu o co najmniej podanej liczbie bitów (*N*).
`INT_LEAST(N)_MIN`	Minimalna wartość dopasowanego typu o co najmniej podanej liczbie bitów (*N*).
`UINT_LEAST(N)_MAX`	Maksymalna wartość dopasowanego typu bez znaku o co najmniej podanej liczbie bitów (*N*).
`int_fast(N)_t`	Podobny do `int_least*N*_`, ale dotyczy „najszybszego" typu o co najmniej podanej precyzji.
`uint_fast(N)_t`	Najszybsza liczba całkowita bez znaku o co najmniej podanej precyzji.
`INT_FAST(N)_MAX`	Maksymalna wartość dopasowanego najszybszego typu o podanej liczbie bitów (*N*).
`INT_FAST(N)_MIN`	Minimalna wartość dopasowanego najszybszego typu o podanej liczbie bitów (*N*).
`UINT_FAST(N)_MAX`	Maksymalna wartość dopasowanego najszybszego typu bez znaku o podanej liczbie bitów (*N*).
`intptr_t`	Liczba całkowita *ze znakiem* o wielkości wystarczającej do przechowywania wskaźnika.
`uintptr_t`	Liczba całkowita *bez znaku* o wielkości wystarczającej do przechowywania wskaźnika.
`INTPTR_MAX`	Maksymalna wartość `intptr_t`.
`INTPTR_MIN`	Minimalna wartość `intptr_t`.
`UINTPTR_MAX`	Maksymalna wartość `uintptr_t` bez znaku.
`intmax_t`	Największa liczba obsługiwana w tym systemie.
`uintmax_t`	Największa liczba bez znaku obsługiwana w tym systemie.
`INTMAX_MAX`	Największa wartość dla największej liczby ze znakiem.
`INTMAX_MIN`	Najmniejsza wartość dla największej liczby ze znakiem.
`UINTMAX_MAX`	Największa wartość dla największej liczby bez znaku.
`PTRDIFF_MIN`	Minimalna wartość `ptrdiff_t`.
`PTRDIFF_MAX`	Maksymalna wartość `ptrdiff_t`.
`SIZE_MAX`	Maksymalna wartość `size_t`.

Dostępne operatory

Poniżej przedstawiłem obszerną listę wszystkich operatorów istniejących w języku C. Na liście tej przyjąłem następujące założenia:

Operator	Opis
(dwuargumentowy)	Operator ma operandy lewy i prawy, na przykład `X + Y`.
(jednoargumentowy)	Operator samodzielny, na przykład `-X`.
(prefiks)	Operator pojawia się przed zmienną, na przykład `++X`.
(postfiks)	Z reguły tak samo jak w przypadku wersji (prefiks), ale umieszczenie operatora daje inne znaczenie, na przykład `X++`.
(trójargumentowy)	Mamy tylko jeden operator tego rodzaju i nosi on nazwę operatora trójargumentowego, a jego znaczenie to „trzy operandy", na przykład `X ? Y : Z`.

Operatory matematyczne

Tego rodzaju operatory są przeznaczone do przeprowadzania podstawowych operacji matematycznych. Umieściłem tutaj także zwykły nawias okrągły (), ponieważ oznacza on wywołanie funkcji, co jest bliskie operacji matematycznej.

Operator	Opis
()	Wywołanie funkcji
* (dwuargumentowy)	Mnożenie
/	Dzielenie
+ (dwuargumentowy)	Dodawanie
+ (jednoargumentowy)	Liczba dodatnia
++ (postfiks)	Odczytywanie, następnie inkrementacja
++ (prefiks)	Inkrementacja, następnie odczytywanie
-- (postfiks)	Odczytywanie, następnie dekrementacja
-- (prefiks)	Dekrementacja, następnie odczytywanie
- (dwuargumentowy)	Odejmowanie
- (jednoargumentowy)	Liczba ujemna

Operatory danych

Tego rodzaju operatory są używane w celu uzyskania dostępu do danych na różne sposoby oraz pod różnymi postaciami.

Operator	Opis
->	Uzyskanie dostępu do struktury wskaźnika
.	Uzyskanie dostępu do wartości struktury
[]	Indeks tablicy
sizeof	Wielkość typu lub zmiennej
& (jednoargumentowy)	Adres wskazanego elementu
* (jednoargumentowy)	Wartość wskazanego elementu

Operatory logiczne

Tego rodzaju operatory są przeznaczone do sprawdzania równości i nierówności zmiennych.

Operator	Definicja
!=	Nierówność
<	Mniejszy niż
<=	Mniejszy niż lub równy
==	Równość (to nie jest przypisanie)
>	Większy niż
>=	Większy niż lub równy

Operatory bitowe

To znacznie bardziej zaawansowane operatory, przeznaczone do przesuwania i modyfikacji bitów w liczbach całkowitych.

Operator	Opis
& (bitowy)	Bitowe I (AND)
<<	Przesunięcie w lewo
>>	Przesunięcie w prawo
^	Bitowe XOR (wyłączające LUB)
|	Bitowe LUB (OR)
~	Odwrócenie wszystkich bitów

Operatory boolowskie

Tego rodzaju operatory są przeznaczone do sprawdzenia prawdy. Dokładnie zapoznaj się z operatorem trójargumentowym, ponieważ będzie on niezwykle użyteczny.

Operator	Opis
!	Nie
&&	I (AND)
||	LUB (OR)
?:	Sprawdzenie prawdy za pomocą operatora trójargumentowego. Zapis X ? Y : Z należy odczytać jako „jeśli X, wtedy Y, w przeciwnym razie Z".

Operatory przypisania

Poniżej wymieniłem złożone operatory przypisania, które pozwalają na przypisanie wartości i (lub) jednoczesne przeprowadzenie operacji. Większość wymienionych tutaj operatorów można łączyć w bardziej złożone operatory przypisania.

Operator	Opis
=	Przypisanie wartości
%=	Modulo wartości i przypisanie wyniku
&=	Bitowe I wartości i przypisanie wyniku
*=	Mnożenie wartości i przypisanie wyniku
+=	Dodanie wartości i przypisanie wyniku
-=	Odjęcie wartości i przypisanie wyniku
/=	Dzielenie wartości i przypisanie wyniku
<<=	Przesunięcie w lewo wartości i przypisanie wyniku
>>=	Przesunięcie w prawo wartości i przypisanie wyniku
^=	Bitowe XOR wartości i przypisanie wyniku
|=	Bitowe LUB wartości i przypisanie wyniku

Dostępne struktury kontroli

Mamy kilka struktur kontroli, z którymi jeszcze nie miałeś okazji się spotkać.

- `do-while`. Pętla `do { ... } while(X);`. Najpierw wykonuje kod w bloku, a następnie sprawdza wartość wyrażenia `X` przed zakończeniem działania.
- `break`. Powoduje przerwanie pętli, czyli jej wcześniejsze zakończenie.
- `continue`. Zatrzymuje wykonywanie kodu w bloku pętli i powraca do sprawdzenia warunku, aby móc kontynuować działanie.
- `goto`. Przechodzi do podanego miejsca w kodzie, gdzie znajduje się podana `etykieta:`. Przykład użycia widziałeś w pliku *dbg.h*, w którym makro powodowało przejście do sekcji `error:`.

Zadania dodatkowe

- Przejrzyj zawartość pliku nagłówkowego *stdint.h* lub jego opis, a następnie wypisz wszystkie dostępne identyfikatory dotyczące wielkości.
- Przejrzyj każdy wypisany element, a następnie zapisz jego znaczenie w kodzie. Aby poprawnie określić działanie poszczególnych elementów, będziesz musiał poszukać informacji w internecie.
- Zapamiętaj zdobyte informacje przez zapisanie ich na kartach i powtarzanie przez 15 minut dziennie.
- Utwórz program wyświetlający przykładowe dane w poszczególnych typach, a następnie sprawdź, czy wcześniej prawidłowo określiłeś działanie tych typów.

ĆWICZENIE 22

Stos, zakres i elementy globalne

Koncepcja zakresu wydaje się skomplikowana dla osób stawiających dopiero pierwsze kroki w programowaniu. Wywodzi się z użycia stosu systemowego (został omówiony nieco wcześniej) oraz sposobu jego wykorzystania do przechowywania zmiennych tymczasowych. W tym ćwiczeniu dowiesz się, czym jest zakres na podstawie analizy działania struktur danych stosu. Następnie zobaczysz, jakie ta koncepcja ma zastosowanie w nowoczesnym języku C.

Jednak rzeczywistym celem tego ćwiczenia jest pokazanie niebezpieczeństw związanych ze stosem i zakresem w C. Kiedy programista nie rozumie koncepcji zakresu, to prawie zawsze ma problem ze zrozumieniem tego, jak tworzone są zmienne, gdzie istnieją oraz jak i kiedy są usuwane. Gdy poznasz wymienione zagadnienia, koncepcja zakresu stanie się łatwiejsza do opanowania.

W tym ćwiczeniu będziemy korzystać z trzech plików:

ex22.h. To jest plik nagłówkowy zawierający definicję pewnych zmiennych zewnętrznych oraz funkcji.

ex22.c. To nie jest plik w znanej dotąd postaci z funkcją `main()`, ale plik kodu źródłowego, który po kompilacji stanie się plikiem obiektowym *ex22.o*, zawierającym pewne funkcje i zmienne zdefiniowane w pliku nagłówkowym *ex22.h*.

ex22_main.c. Ten plik zawiera funkcję `main()` wraz z odniesieniami do dwóch wyżej wymienionych plików. Dzięki temu będzie można zaprezentować różne koncepcje zakresu.

Pliki ex22.h i ex22.c

Pracę zaczynamy od utworzenia pliku nagłówkowego o nazwie *ex22.h* zawierającego definicje funkcji i zmiennych zewnętrznych.

Plik *ex22.h*:

```
#ifndef _ex22_h
#define _ex22_h

// Zmienna THE_SIZE z pliku ex22.c staje się dostępna w innych plikach .c.
extern int THE_SIZE;

// Pobieranie i ustawienie jako wewnętrznej zmiennej statycznej w pliku ex22.c.
int get_age();
void set_age(int age);

// Uaktualnienie zmiennej statycznej znajdującej się wewnątrz update_ratio().
double update_ratio(double ratio);
void print_size();

#endif
```

Koniecznie zwróć uwagę na użycie w powyższym pliku polecenia `extern int THE_SIZE;`, którego znaczenie wyjaśnię po przygotowaniu pliku *ex22.c*, odpowiadającego utworzonemu wcześniej plikowi nagłówkowemu.

Plik *ex22.c.*:

```
 1 #include <stdio.h>
 2 #include "ex22.h"
 3 #include "dbg.h"
 4
 5 int THE_SIZE = 1000;
 6
 7 static int THE_AGE = 37;
 8
 9 int get_age()
10 {
11     return THE_AGE;
12 }
13
14 void set_age(int age)
15 {
16     THE_AGE = age;
17 }
18
19 double update_ratio(double new_ratio)
20 {
21     static double ratio = 1.0;
22
23     double old_ratio = ratio;
24     ratio = new_ratio;
25
26     return old_ratio;
27 }
28
29 void print_size()
30 {
31     log_info("Wydaje się, że wielkość wynosi: %d", THE_SIZE);
32 }
```

W dwóch przedstawionych powyżej plikach wprowadziliśmy pewne nowe rodzaje magazynu danych dla zmiennych:

`extern`. Można powiedzieć, że to słowo kluczowe oznajmia kompilatorowi: „Zmienna istnieje, ale w innym położeniu zewnętrznym". Zwykle oznacza to, że jeden z plików *.c* będzie używał zmiennej zdefiniowanej w innym pliku *.c*. W omawianym przypadku wskazujemy, że plik *ex22.c* zawiera zmienną `THE_SIZE`, do której dostęp będzie odbywał się z poziomu pliku *ex22_main.c*.

`static (plik)`. Znaczenie tego słowa kluczowego można uznać za odwrotne dla `extern` — dana zmienna jest używana jedynie w bieżącym pliku *.c*, nie powinna być dostępna w innych fragmentach programu. Musisz koniecznie zapamiętać, że polecenie `static` na poziomie pliku (podobnie jak tutaj `THE_AGE`) ma inne znaczenie niż w pozostałych miejscach.

`static (funkcja)`. Jeżeli zadeklarujesz zmienną w funkcji określonej jako `static`, to ta zmienna działa jak zdefiniowana jako `static` w pliku, ale będzie dostępna jedynie dla danej funkcji. Jest to sposób na utworzenie stałej dla funkcji, choć w rzeczywistości ta możliwość jest naprawdę *rzadko* wykorzystywana w nowoczesnym programowaniu w C, ponieważ trudno ją stosować wraz z wątkami.

W tych dwóch plikach powinieneś poznać następujące zmienne i funkcje:

`THE_SIZE`. Zmienna zadeklarowana jako `extern`, będziemy jej używać z poziomu pliku *ex22_main.c*.

`get_age()` i **`set_age()`**. Te funkcje pobierają zmienną statyczną `THE_AGE` i udostępniają ją innym fragmentom programu. Tylko wymienione funkcje mogą uzyskać bezpośredni dostęp do zmiennej `THE_AGE`.

`update_ratio()`. Ta funkcja pobiera nową wartość `ratio` i zwraca starą. Do śledzenia bieżącej wartości wykorzystuje zmienną statyczną na poziomie funkcji.

`print_size()`. Ta funkcja wyświetla wartość, jaką według kodu w pliku *ex22.c* ma zmienna `THE_SIZE`.

Plik ex22_main.c

Po utworzeniu wymienionych powyżej plików możemy przystąpić do przygotowania funkcji `main()` wykorzystującej te pliki oraz prezentującej pewne koncepcje dotyczące zakresu.

Plik *ex22_main.c*:

```
 1 #include "ex22.h"
 2 #include "dbg.h"
 3
 4 const char *MY_NAME = "Zed A. Shaw";
 5
 6 void scope_demo(int count)
 7 {
 8     log_info("Licznik wynosi: %d", count);
 9
10     if (count > 10) {
11         int count = 100; // ŹLE! BŁĄD!
12
13         log_info("Licznik w tym zakresie wynosi %d", count);
14     }
15
16     log_info("Licznik na wyjściu wynosi: %d", count);
17
18     count = 3000;
19
20     log_info("Licznik po przypisaniu wynosi: %d", count);
21 }
22
23 int main(int argc, char *argv[])
24 {
```

```
25      // Sprawdzenie działania akcesorów THE_AGE.
26      log_info("Nazywam się: %s, age: %d", MY_NAME, get_age());
27
28      set_age(100);
29
30      log_info("Mój obecny wiek: %d", get_age());
31
32      // Sprawdzenie działania extern THE_SIZE.
33      log_info("Wartość THE_SIZE wynosi: %d", THE_SIZE);
34      print_size();
35
36      THE_SIZE = 9;
37
38      log_info("Obecnie wartość THE_SIZE wynosi: %d", THE_SIZE);
39      print_size();
40
41      // Sprawdzenie działania funkcji statycznej.
42      log_info("Wartość początkowa: %f", update_ratio(2.0));
43      log_info("Wartość ponownie: %f", update_ratio(10.0));
44      log_info("Wartość raz jeszcze: %f", update_ratio(300.0));
45
46      // Sprawdzenie działania przykładowego zakresu.
47      int count = 4;
48      scope_demo(count);
49      scope_demo(count * 20);
50
51      log_info("Licznik po wywołaniu scope_demo(): %d", count);
52
53      return 0;
54  }
```

Poniżej przedstawiam omówienie kodu niemalże wiersz po wierszu. W trakcie czytania powinieneś wyszukiwać poszczególne zmienne i miejsca ich występowania.

ex22_main.c:4. Słowo kluczowe `const` oznacza stałą, to alternatywne podejście do użycia `define` w celu utworzenia stałej.

ex22_main.c:6. Prosta funkcja demonstrująca więcej kwestii dotyczących zakresu w funkcji.

ex22_main.c:8. Wyświetlenie wartości licznika, jaką ma on na początku wykonywania funkcji.

ex22_main.c:10. Konstrukcja `if` rozpoczynająca nowy *blok zakresu* wraz z następną zmienną licznika (`count`). Tak naprawdę ta wersja `count` to zupełnie nowa zmienna. Można by powiedzieć, że konstrukcja `if` uruchomiła nową minifunkcję.

ex22_main.c:11. Ta zmienna `count` pozostaje lokalna dla aktualnego bloku, jest zupełnie inna od zmiennej o tej samej nazwie podanej na liście parametrów funkcji.

ex22_main.c:13. Polecenie wyświetla wartość licznika (100), to nie jest wartość przekazana wywołaniu funkcji `scope_demo()`.

ex22_main.c:16. Docieramy do trudniejszego fragmentu. Zmienną `count` mamy w dwóch miejscach: w parametrach funkcji oraz w konstrukcji `if`. Ponieważ

konstrukcja `if` utworzyła nowy blok, więc zmienna `count` w wierszu 11. *nie ma wpływu na parametr o tej samej nazwie*. Po wykonaniu polecenia w tym wierszu zobaczysz, że wyświetlona jest wartość parametru, a nie `100`.

ex22_main.c:18 – 20. Parametrowi `count` przypisujemy wartość `3000` i wyświetlamy ją, co pokazuje możliwość zmiany parametrów funkcji. Wspomniana zmiana nie ma wpływu na wersję zmiennej używanej przez komponent wywołujący tę funkcję.

Upewnij się, że dokładnie prześledziłeś zdarzenia w trakcie wykonywania funkcji, choć nie powinieneś uważać, że już zrozumiałeś koncepcję zakresu. Po prostu zapamiętaj, że po umieszczeniu zmiennej wewnątrz bloku (na przykład utworzonego za pomocą konstrukcji `if` lub `while`), powstają zupełnie *nowe* zmienne, istniejące jedynie w tym bloku. Zrozumienie tej koncepcji ma znaczenie kluczowe, a brak jej zrozumienia *jest źródłem wielu błędów*. Wkrótce dowiesz się, dlaczego nie powinieneś tworzyć zmiennych w bloku.

Pozostała część pliku *ex22_main.c* pokazuje różne sposoby, na jakie można operować zmiennymi i wyświetlać ich wartości.

ex22_main.c:26. Wyświetlenie bieżącej wartości `MY_NAME` i pobranie wartości zmiennej `THE_AGE` w pliku *ex22.c* za pomocą funkcji akcesora `get_age()`.

ex22_main.c:27 – 30. Użycie funkcji akcesora `set_age()` w pliku *ex22.c* w celu zmiany wartości zmiennej `THE_AGE` i wyświetlenie jej wartości.

ex22_main.c:33 – 39. Przeprowadzenie tej samej operacji względem zmiennej `THE_SIZE` w pliku *ex22.c*, ale tym razem przez uzyskanie bezpośredniego dostępu. W tym miejscu pokazuję również rzeczywistą zmianę w pliku i wyświetlenie wartości za pomocą funkcji `print_size()`.

ex22_main.c:42 – 44. Przykład użycia zmiennej statycznej `ratio` wewnątrz funkcji `update_ratio()`. Wartość wymienionej zmiennej jest zachowywana między dwoma wywołaniami funkcji.

ex22_main.c:46 – 51. Kilkakrotne wywołanie funkcji `scope_demo()`, aby pokazać zakres w działaniu. Warto w tym miejscu zwrócić uwagę, że wartość lokalnej zmiennej `count` pozostaje bez zmian. *Musisz* zrozumieć, że przekazanie takiej zmiennej nie pozwala na zmianę jej wartości w funkcji. Jeżeli chcesz przeprowadzić zmianę, musisz skorzystać z pomocy starego, dobrego znajomego, czyli wskaźnika. W przypadku przekazania wskaźnika do zmiennej `count` wywołana funkcja otrzyma adres tej zmiennej i będzie mogła zmienić jej wartość.

Powyższe wyjaśnienie dokładnie przedstawia sposób działania programu, ale i tak powinieneś monitorować zdarzenia zachodzące w plikach oraz upewnić się, że wszystko dokładnie zrozumiałeś.

Co powinieneś zobaczyć?

Tym razem zamiast do użycia pliku *Makefile* zachęcam do samodzielnej kompilacji obu plików, co pozwoli Ci zobaczyć, jak kompilator w rzeczywistości je połączy. Poniżej przedstawiłem przykładowe dane wyjściowe sesji.

Sesja dla ćwiczenia 22.:

```
$ cc -Wall -g -DNDEBUG -c -o ex22.o ex22.c
$ cc -Wall -g -DNDEBUG ex22_main.c ex22.o -o ex22_main
$ ./ex22_main
[INFO] (ex22_main.c:26) Nazywam się: Zed A. Shaw, age: 37
[INFO] (ex22_main.c:30) Mój obecny wiek: 100
[INFO] (ex22_main.c:33) Wartość THE_SIZE wynosi: 1000
[INFO] (ex22.c:32) Wydaje się, że wielkość wynosi: 1000
[INFO] (ex22_main.c:38) Obecnie wartość THE_SIZE wynosi: 9
[INFO] (ex22.c:32) Wydaje się, że wielkość wynosi: 9
[INFO] (ex22_main.c:42) Wartość początkowa: 1.000000
[INFO] (ex22_main.c:43) Wartość ponownie: 2.000000
[INFO] (ex22_main.c:44) Wartość raz jeszcze: 10.000000
[INFO] (ex22_main.c:8) Licznik wynosi: 4
[INFO] (ex22_main.c:16) Licznik na wyjściu wynosi: 4
[INFO] (ex22_main.c:20) Licznik po przypisaniu wynosi: 3000
[INFO] (ex22_main.c:8) Licznik wynosi: 80
[INFO] (ex22_main.c:13) Licznik w tym zakresie wynosi 100
[INFO] (ex22_main.c:16) Licznik na wyjściu wynosi: 80
[INFO] (ex22_main.c:20) Licznik po przypisaniu wynosi: 3000
[INFO] (ex22_main.c:51) Licznik po wywołaniu scope_demo(): 4
```

Uważnie monitoruj zmiany zachodzące w poszczególnych zmiennych i dopasowanie tych operacji do wierszy generujących dane wyjściowe. W przykładzie wykorzystałem funkcję `log_info()` z makr zdefiniowanych w pliku *dbg.h*, więc otrzymujesz dokładny numer wiersza, w którym wyświetlana jest wartość zmiennej; możesz ją wyszukać w plikach w celu dalszego monitorowania.

Zakres, stos i błędy

Jeżeli wszystko zrobiłeś prawidłowo, powinieneś teraz znać kilka różnych sposobów na umieszczanie zmiennych w kodzie źródłowym C. Dostępne sposoby to między innymi użycie słowa kluczowego `extern` i funkcji akcesorów, takich jak `get_age()`, w celu utworzenia zmiennych globalnych. Możesz tworzyć nowe zmienne wewnątrz bloków, ich wartości pozostaną aż do chwili zakończenia działania bloku, a znajdujące się na zewnątrz bloku zmienne o takich samych nazwach będą nietknięte. Istnieje możliwość przekazania wartości do funkcji i możliwość zmiany parametru, ale bez zmiany wersji znajdującej się w komponencie wywołującym tę funkcję.

Najważniejsze jest zapamiętanie, że wszystkie wymienione działania mogą spowodować powstanie błędów. Możliwość umieszczania przez C elementów w różnych miejscach pamięci komputera, a następnie udzielenie do nich dostępu oznacza większe prawdopodobieństwo pomyłki dotyczącej miejsca przechowywania danego elementu. Jeżeli nie wiesz, gdzie znajduje się dany element, to prawdopodobnie nie obsłużysz go prawidłowo.

Mając to wszystko na uwadze, poniżej przedstawiłem kilka reguł, których przestrzeganie podczas tworzenia kodu źródłowego w C pomaga uniknąć błędów związanych ze stosem.

- Nie przesłaniaj zmiennej, jak zostało to zrobione w omawianym programie w przypadku zmiennej `count` w funkcji `scope_demo()`. W ten sposób można doprowadzić do powstania ukrytych i trudnych do wychwycenia błędów, gdy *jesteś przekonany* o zmianie wartości, choć tak naprawdę wartość pozostaje niezmieniona.
- Unikaj stosowania zbyt wielu zmiennych globalnych, zwłaszcza między wieloma plikami. Jeżeli musisz z nich korzystać, zdecyduj się na funkcje akcesora, tak jak w przypadku pokazanej w programie `get_age()`. To nie dotyczy stałych, ponieważ są tylko do odczytu. Mam tutaj na myśli jedynie zmienne, takie jak `THE_SIZE`. Jeżeli chcesz umożliwić innym modyfikowanie lub ustawienie tego rodzaju zmiennych, utwórz dla nich funkcje akcesorów.
- Kiedy masz wątpliwości, umieść zmienną na stercie. Nie opieraj się na semantyce stosu lub specjalizowanych lokalizacjach. Po prostu twórz elementy za pomocą wywołań `malloc()`.
- Nie używaj statycznych zmiennych funkcji, jak to zrobiłem w przypadku funkcji `update_ratio()`. One rzadko bywają użyteczne, a najczęściej sprawiają poważne problemy, gdy kod ma działać równolegle w wątkach. Ponadto są niezwykle trudne do wyszukania w porównaniu z doskonale zdefiniowanymi zmiennymi globalnymi.
- Unikaj ponownego użycia parametrów funkcji. To wprowadza niepotrzebne zamieszanie, ponieważ nie wiadomo, czy mamy do czynienia z po prostu ponownym użyciem, czy jednak ze zmianą wersji znajdującej się w komponencie wywołującym tę funkcję.

Podobnie jak w wielu innych przypadkach, także powyższe reguły można złamać, jeśli będzie to praktyczne. Tak naprawdę gwarantuję Ci, że będziesz miał do czynienia z kodem, który pomimo złamania wszystkich powyższych reguł nadal działa doskonale. Ograniczenia powodowane przez różne platformy czasami nawet wręcz wymagają łamania reguł.

Jak to zepsuć?

W ramach ćwiczenia tym razem spróbuj uzyskać dostęp do pewnych elementów lub je zmienić, ale w taki sposób, aby nie spowodować awarii programu.

- Spróbuj z poziomu pliku *ex22_main.c* uzyskać bezpośredni dostęp do zdefiniowanych w pliku *ex22.c* zmiennych, do których, jak sądzisz, nie masz dostępu. Czy na przykład w funkcji `update_ratio()` możesz pobrać wartość zmiennej `ratio`? Co się stanie, jeśli będziesz mieć wskaźnik prowadzący do tej zmiennej?
- Pozbądź się deklaracji `extern` w pliku *ex22.c* i zobacz, jakie otrzymasz komunikaty o błędach lub z ostrzeżeniami.
- Dodaj specyfikatory `static` lub `const` do różnych zmiennych, a następnie spróbuj je zmodyfikować.

Zadania dodatkowe

- Poszukaj informacji dodatkowych o koncepcji przekazywania przez wartość oraz przekazywania przez referencję. Utwórz przykłady wykorzystujące oba rodzaje przekazywania.
- Użyj wskaźników, aby uzyskać dostęp do elementów, do których, jak sądzisz, nie masz dostępu.
- Użyj debugera, aby sprawdzić, jak przedstawia się kwestia dostępu, gdy próbujesz go uzyskać w niepoprawny sposób.
- Utwórz funkcję rekurencyjną powodującą przepełnienie stosu. Nie wiesz, co to jest funkcja rekurencyjna? Spróbuj umieścić wywołanie funkcji `scope_demo()` na końcu kodu tej funkcji, aby w ten sposób zapętlić jej wywołanie.
- Zmodyfikuj plik *Makefile* w taki sposób, aby umożliwić kompilację programu omówionego w ćwiczeniu.

ĆWICZENIE 23

Poznaj mechanizm Duffa

To ćwiczenie może być niezłą łamigłówką dla mózgu, ponieważ zamierzam wprowadzić jedną z najsławniejszych sztuczek stosowanych w języku C, czyli mechanizm Duffa. Nazwa pochodzi od nazwiska jego wynalazcy, Toma Duffa. Ten niewielki fragment doskonałego (diabelskiego?) kodu pozwala na zapakowanie w mały pakiet wszystkiego, czego się dotąd nauczyłeś. Ponadto określenie sposobu działania omawianego tutaj mechanizmu to również przykład dobrej, logicznej zabawy.

> **OSTRZEŻENIE** Możliwość zastosowania tego rodzaju szalonych sztuczek w języku C zarówno przynosi frajdę podczas programowania, jak i powoduje, że C jest irytujący w użyciu. Dobrze jest znać takie sztuczki, ponieważ dzięki temu jeszcze lepiej poznajesz język i komputer. Jednak nigdy nie powinieneś stosować tego rodzaju kombinacji. Zamiast tego należy dążyć do tworzenia kodu łatwego w odczycie.

Odkryty przez Toma Duffa mechanizm to związana z kompilatorem C sztuczka, która tak naprawdę nie powinna działać. W tym momencie jeszcze nie zdradzę Ci jej przeznaczenia — to łamigłówka dla Ciebie, którą powinieneś spróbować rozwiązać. Zacznij od wprowadzenia przedstawionego poniżej kodu, a następnie spróbuj ustalić jego przeznaczenie i odpowiedzieć na pytanie, *dlaczego* działa właśnie w taki sposób.

Plik *ex23.c*:

```
#include <stdio.h>
#include <string.h>
#include "dbg.h"

int normal_copy(char *from, char *to, int count)
{
    int i = 0;

    for (i = 0; i < count; i++) {
        to[i] = from[i];
}

    return i;
}

int duffs_device(char *from, char *to, int count)
{
    {
        int n = (count + 7) / 8;

        switch (count % 8) {
            case 0:
```

```
23                  do {
24                      *to++ = *from++;
25                      case 7:
26                      *to++ = *from++;
27                      case 6:
28                      *to++ = *from++;
29                      case 5:
30                      *to++ = *from++;
31                      case 4:
32                      *to++ = *from++;
33                      case 3:
34                      *to++ = *from++;
35                      case 2:
36                      *to++ = *from++;
37                      case 1:
38                      *to++ = *from++;
39                  } while (--n > 0);
40          }
41      }
42
43      return count;
44  }
45
46  int zeds_device(char *from, char *to, int count)
47  {
48      {
49          int n = (count + 7) / 8;
50
51          switch (count % 8) {
52              case 0:
53  again: *to++ = *from++;
54
55              case 7:
56              *to++ = *from++;
57              case 6:
58              *to++ = *from++;
59              case 5:
60              *to++ = *from++;
61              case 4:
62              *to++ = *from++;
63              case 3:
64              *to++ = *from++;
65              case 2:
66              *to++ = *from++;
67              case 1:
68              *to++ = *from++;
69              if (--n > 0)
70                  goto again;
71          }
72      }
73
74      return count;
```

```
}

int valid_copy(char *data, int count, char expects)
{
    int i = 0;
    for (i = 0; i < count; i++) {
        if (data[i] != expects) {
            log_err("[%d] %c != %c", i, data[i], expects);
            return 0;
        }
    }

    return 1;
}

int main(int argc, char *argv[])
{
    char from[1000] = { 'a' };
    char to[1000] = { 'c' };
    int rc = 0;

    // Konfiguracja.
    memset(from, 'x', 1000);
    // Zdefiniowanie trybu awarii.
    memset(to, 'y', 1000);
    check(valid_copy(to, 1000, 'y'), "Nie zainicjalizowano prawidłowo.");

    // Użycie zwykłego kopiowania.
    rc = normal_copy(from, to, 1000);
    check(rc == 1000, "Zwykłe kopiowanie nie udało się: %d", rc);
    check(valid_copy(to, 1000, 'x'), "Zwykłe kopiowanie nie udało się.");

    // Zerowanie.
    memset(to, 'y', 1000);

    // Mechanizm Duffa.
    rc = duffs_device(from, to, 1000);
    check(rc == 1000, "Mechanizm Duffa uległ awarii: %d", rc);
    check(valid_copy(to, 1000, 'x'), "Mechanizm Duffa uległ awarii.");

    // Zerowanie.
    memset(to, 'y', 1000);

    // Moja wersja.
    rc = zeds_device(from, to, 1000);
    check(rc == 1000, "Mechanizm Zeda uległ awarii: %d", rc);
    check(valid_copy(to, 1000, 'x'), "Mechanizm Zeda uległ awarii.");

    return 0;
error:
    return 1;
}
```

W powyższym kodzie źródłowym znalazły się trzy wersje funkcji kopiującej:

`normal_copy()`. Ta funkcja jest oparta na zwykłej pętli `for` kopiującej znaki z jednej tablicy do drugiej.

`duffs_device()`. W tej funkcji mamy implementację tak zwanego mechanizmu Duffa.

`zeds_device()`. Z kolei w tej funkcji mamy inną implementację mechanizmu Duffa, używającą polecenia `goto`, dzięki któremu otrzymujesz wskazówkę, co tak naprawdę dzieje się w dziwnej pętli `do-while`, wykorzystanej w funkcji `duffs_device()`.

Zanim będziesz kontynuować lekturę ćwiczenia, dokładnie przeanalizuj kod wszystkich trzech funkcji. Spróbuj samodzielnie odpowiedzieć na pytanie, co tak naprawdę się w nich dzieje.

Co powinieneś zobaczyć?

Ten program nie generuje danych wyjściowych, po prostu jest wykonywany i kończy działanie. Uruchom go za pomocą debugera, aby zobaczyć, czy jesteś w stanie wychwycić jakiekolwiek inne błędy. Spróbuj samemu wprowadzić pewne błędy, jak pokazałem w ćwiczeniu 4.

Rozwiązanie łamigłówki

Pierwsza kwestia, o której należy pamiętać, to istnienie w języku C dość luźnego podejścia do pewnych fragmentów składni. Dlatego też możesz umieścić pół pętli `do-while` w jednym fragmencie konstrukcji `switch` i pół w innym miejscu, a mimo to kod nadal będzie działał. Jeżeli ponownie spojrzysz na przygotowaną przeze mnie wersję z poleceniem `goto`, znacznie wyraźniej dostrzeżesz to, co się dzieje w kodzie. Upewnij się, że zrozumiałeś sposób działania tej części kodu.

Druga kwestia wiąże się z tym, jak domyślna semantyka przejścia przez bloki `case` konstrukcji `switch` pozwala na przeskok do wybranego bloku `case`, a następnie po prostu działanie kodu aż do końca konstrukcji `switch`.

Ostatnią wskazówką jest polecenie `count % 8` i obliczenie `n` na początku.

Teraz spróbuj określić sposób działania tych funkcji. W tym celu wykonaj poniższe działania:

- Wydrukuj omawiany fragment kodu, aby móc dokonywać pewnych zapisków na papierze.
- Wypisz wszystkie zmienne w tabeli w postaci zainicjalizowanej przed rozpoczęciem wykonywania konstrukcji `switch`.
- Przeanalizuj logikę konstrukcji `switch` i przechodź do odpowiednich bloków `case`.
- Uaktualniaj informacje o zmiennych (`to` i `from`) oraz wskazywanych przez nie tablicach.
- Kiedy dotrzesz do pętli `while` lub polecenia `goto` w mojej wersji funkcji, sprawdź zmienne, a następnie podążaj za logiką na początek pętli `do-while` lub do miejsca położenia etykiety `again`.

- Kontynuuj ręczne monitorowanie działania programu i uaktualnianie zmiennych, aż do chwili, gdy dokładnie zrozumiesz, jak ten program działa.

Dlaczego w ogóle mam się tak męczyć?

Kiedy ustalisz rzeczywisty sposób działania powyższego fragmentu kodu, ostatnie pytanie brzmi: „Czy kiedykolwiek będę używał takiego rozwiązania?". Celem pokazanej sztuczki jest ręczne rozwijanie pętli. Ogromne, długie pętle mogą być wykonywane powoli, więc jednym ze sposobów na ich przyspieszenie jest odszukanie pewnego stałego fragmentu w pętli, a następnie po prostu wielokrotne, sekwencyjne powielenie w niej kodu. Jeżeli na przykład wiadomo, że pętla jest wykonywana minimum 20 razy, to zawartość pętli można umieścić 20 razy w kodzie źródłowym.

Działanie mechanizmu Duffa w zasadzie polega na zautomatyzowaniu powyższego przez podział pętli na 8 fragmentów iteracji. Rozwiązanie to jest sprytne i faktycznie działa, ale obecnie mamy do dyspozycji dobre kompilatory, które podejmują takie działania za programistę. Nigdy nie powinieneś stosować mechanizmu Duffa, z wyjątkiem naprawdę rzadkich przypadków, w których *dowiodłeś*, że użycie tego mechanizmu przekłada się na wzrost wydajności działania programu.

Zadania dodatkowe

- Nigdy więcej nie używaj mechanizmu Duffa.
- Przeczytaj zamieszczony w Wikipedii artykuł dotyczący mechanizmu Duffa i zobacz, czy jesteś w stanie wychwycić błąd. Przeczytaj wymieniony artykuł, porównaj zamieszczony tam kod z wersją przedstawioną w ćwiczeniu i spróbuj zrozumieć, dlaczego kod z Wikipedii nie działa u Ciebie, ale działał u Toma Duffa.
- Napisz zestaw makr pozwalających na utworzenie tego rodzaju mechanizmu o dowolnej wielkości. Na przykład chcesz mieć 32 bloki `case`, ale nie masz ochoty na samodzielne utworzenie ich wszystkich. Czy potrafisz przygotować makro, które jednorazowo umieści 8 bloków `case`?
- Zmodyfikuj funkcję `main()` w celu przeprowadzenia testów szybkości i sprawdź, która wersja charakteryzuje się większą wydajnością.
- Poszukaj informacji o funkcjach `memcpy()`, `memmove()` i `memset()`, a następnie porównaj wydajność ich działania.
- Nigdy więcej nie używaj mechanizmu Duffa!

ĆWICZENIE 24

Dane wejściowe, dane wyjściowe i pliki

Potrafisz wykorzystać funkcję `printf()` do wyświetlania informacji. Wprawdzie to doskonała funkcjonalność, ale tak naprawdę potrzebujemy nieco więcej możliwości. W tym ćwiczeniu będziemy używać funkcji `fscanf()` i `fgets()` w celu umieszczenia w strukturze informacji o danej osobie. Po tym prostym wprowadzeniu dotyczącym danych wejściowych poznasz pełną listę funkcji, jakie język C ma do obsługi operacji wejścia-wyjścia. Niektóre z nich już spotkałeś i wykorzystywałeś, więc otrzymasz kolejną możliwość utrwalenia informacji.

Plik *ex24.c*:

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

#define MAX_DATA 100

typedef enum EyeColor {
    BLUE_EYES, GREEN_EYES, BROWN_EYES,
    BLACK_EYES, OTHER_EYES
} EyeColor;

const char *EYE_COLOR_NAMES[] = {
    "niebieski", "zielony", "brązowy", "czarny", "inny"
};

typedef struct Person {
    int age;
    char first_name[MAX_DATA];
    char last_name[MAX_DATA];
    EyeColor eyes;
    float income;
} Person;

int main(int argc, char *argv[])
{
    Person you = {.age = 0 };
    int i = 0;
    char *in = NULL;

    printf("Jak masz na imię? ");
    in = fgets(you.first_name, MAX_DATA - 1, stdin);
    check(in != NULL, "Nie udało się odczytać imienia.");

    printf("Jak masz na nazwisko? ");
    in = fgets(you.last_name, MAX_DATA - 1, stdin);
    check(in != NULL, "Nie udało się odczytać nazwiska.");
```

```
36
37      printf("Ile masz lat? ");
38      int rc = fscanf(stdin, "%d", &you.age);
39      check(rc > 0, "Musisz podać liczbę.");
40
41      printf("Jaki masz kolor oczu:\n");
42      for (i = 0; i <= OTHER_EYES; i++) {
43          printf("%d) %s\n", i + 1, EYE_COLOR_NAMES[i]);
44      }
45      printf("> ");
46
47      int eyes = -1;
48      rc = fscanf(stdin, "%d", &eyes);
49      check(rc > 0, "Musisz podać liczbę.");
50
51      you.eyes = eyes - 1;
52      check(you.eyes <= OTHER_EYES
53          && you.eyes >= 0, "Zrób to dobrze, nie ma takiej opcji.");
54
55      printf("Ile zarabiasz na godzinę? ");
56      rc = fscanf(stdin, "%f", &you.income);
57      check(rc > 0, "Podaj liczbę zmiennoprzecinkową.");
58
59      printf("----- WYNIK -----\n");
60
61      printf("Imię: %s", you.first_name);
62      printf("Nazwisko: %s", you.last_name);
63      printf("Wiek: %d\n", you.age);
64      printf("Kolor oczu: %s\n", EYE_COLOR_NAMES[you.eyes]);
65      printf("Dochód: %f\n", you.income);
66
67      return 0;
68      error:
69
70      return -1;
71  }
```

Powyższy kod źródłowy jest niezwykle prosty i wprowadza użycie funkcji `fscanf()`, która jest przeznaczoną do obsługi plików odmianą funkcji `scanf()`. Rodzina funkcji `scanf()` ma działanie odwrotne do wersji `printf()`. Podczas gdy `printf()` to funkcje przeznaczone do wyświetlania danych na podstawie formatu, `scanf()` odczytuje (inaczej skanuje) dane wejściowe na podstawie formatu.

Na początku pliku nie znajdują się żadne nietypowe polecenia, więc poniżej przedstawiłem omówienie działania funkcji `main()` programu.

ex24.c:25 – 27. Przygotowanie niezbędnych zmiennych.

ex24.c:29 – 31. Pobranie imienia za pomocą funkcji `fgets()`, która odczytuje ciąg tekstowy z danych wejściowych (w omawianym przypadku to standardowe wejście — `stdin`), ale gwarantuje, że dany bufor nie będzie przepełniony.

ex24.c:33 – 35. To samo, ale dotyczy nazwiska. Ponownie używamy funkcji `fgets()`.

ex24.c:38 – 39. Wykorzystanie funkcji `fscanf()` w celu odczytania liczby całkowitej ze standardowego wejścia i umieszczenie jej w zmiennej `age`. Możesz zauważyć użycie tego samego ciągu tekstowego formatowania, jak w przypadku funkcji `printf()` wyświetlającej liczbę całkowitą. Zwróć uwagę na konieczność podania *adresu* zmiennej `age`; funkcja `fscanf()` ma wskaźnik prowadzący do tej zmiennej i może ją modyfikować. To jest doskonały przykład użycia w charakterze parametru wskaźnika prowadzącego do pewnych danych.

ex24.c:41 – 45. Wyświetlenie wszystkich opcji związanych z kolorem oczu wraz z dopasowaniem liczby do przygotowanego wcześniej typu wyliczeniowego `EyeColor`.

ex24.c:47 – 49. Ponowne użycie funkcji `fscanf()` w celu pobrania wieku i umieszczenia go w zmiennej `age`. W trakcie operacji upewniamy się o poprawności danych wejściowych. To jest bardzo ważny aspekt, ponieważ użytkownik może podać wartość spoza tablicy `EYE_COLOR_NAMES` i tym samym spowodować wystąpienie błędu naruszenia ochrony pamięci.

ex24.c:56 – 57. Pobranie informacji o zarobkach użytkownika i umieszczenie tych danych w zmiennej `income`.

ex24.c:61 – 65. Wyświetlenie wszystkich zebranych informacji, co pozwala na ich sprawdzenie. Zwróć uwagę na wykorzystanie tablicy `EYE_COLOR_NAMES` do wyświetlenia wartości wskazywanej przez `EyeColor`.

Co powinieneś zobaczyć?

Po uruchomieniu programu powinieneś zobaczyć, że wprowadzone dane wejściowe zostały prawidłowo skonwertowane. Spróbuj podać także nieprawidłowe dane, co pozwoli Ci się przekonać, jak program potrafi się przed nimi chronić.

Sesja dla ćwiczenia 24.:

```
$ make ex24
cc -Wall -g -DNDEBUG ex24.c -o ex24
$ ./ex24
Jak masz na imię? Zed
Jak masz na nazwisko? Shaw
Ile masz lat? 37
Jaki masz kolor oczu:
1) niebieski
2) zielony
3) brązowy
4) czarny
5) inny
> 1
Ile zarabiasz na godzinę? 1.2345
----- WYNIK -----
Imię: Zed
Nazwisko: Shaw
Wiek: 37
Kolor oczu: niebieski
Dochód: 1.234500
```

Jak to zepsuć?

Wszystko dobrze, ale tak naprawdę najważniejszym aspektem tego ćwiczenia jest rzeczywisty sposób działania funkcji `scanf()`. Sprawdza się ona doskonale podczas prostej konwersji liczby, natomiast okazuje się niedostateczna podczas konwersji ciągu tekstowego, ponieważ trudno jest wskazać prawidłową wielkość bufora przed rozpoczęciem odczytu danych. Mamy także problem z funkcją `gets()` (nie pomyl jej z funkcją `fgets()`) i dlatego unikamy jej użycia. Funkcja `gets()` zupełnie nic nie wie o wielkości bufora danych wejściowych i spowoduje awarię programu.

Aby zademonstrować problemy funkcji `fscanf()` z ciągiem tekstowym, zmień wiersze, w których używamy `fgets()`, aby przyjęły postać `fscanf(stdin, "%50s", you.first_name)`, a następnie ponownie spróbuj uruchomić program. Czy zauważyłeś pobranie zbyt wielu danych i przechwycenie klawisza *Enter*? Nie takiego zachowania oczekujemy i dlatego zamiast zajmować się rozwiązywaniem dziwnych problemów dotyczących `scanf()`, należy po prostu używać `fgets()`.

Następnie zmień wywołania `fgets()` na `gets()` i uruchom program *ex24* za pomocą debugera. Teraz wydaj poniższe polecenie:

```
"run << /dev/urandom"
```

W ten sposób program otrzyma losowo wygenerowane dane. Tego rodzaju operacja nosi nazwę *fuzzingu* i stanowi dobry sposób wyszukiwania wszelkich błędów związanych z danymi wejściowymi. W omawianym przypadku dostarczamy programowi danych z pliku (urządzenia) */dev/urandom*, a następnie obserwujemy jego awarię. Na niektórych platformach operację trzeba będzie przeprowadzić kilkakrotnie lub nawet zmienić wartość `MAX_DATA`, aby była odpowiednio mała.

Funkcja `gets()` jest uznawana za tak złą, że na niektórych platformach generowane jest nawet ostrzeżenie podczas próby uruchomienia *programu* zawierającego jej wywołanie. Nigdy nie powinieneś używać funkcji `gets()`.

Na koniec przejdź do kodu obsługującego `you.eyes` i usuń polecenia odpowiedzialne za sprawdzenie, czy podana liczba znajduje się w odpowiednim zakresie. Następnie podaj nieprawidłową liczbę, taką jak `-1` lub `1000`. Czy w debugerze możesz zobaczyć, co się wówczas dzieje w programie?

Funkcje wejścia-wyjścia

Poniżej przedstawiłem krótką listę różnych funkcji wejścia-wyjścia, na które powinieneś zwrócić uwagę. Przygotuj sobie karty zawierające nazwę funkcji i wszystkie warianty podobne do niej.

- `fscanf()`
- `fgets()`
- `fopen()`

- `freopen()`
- `fdopen()`
- `fclose()`
- `fcloseall()`
- `fgetpos()`
- `fseek()`
- `ftell()`
- `rewind()`
- `fprintf()`
- `fwrite()`
- `fread()`

Przeanalizuj wymienione funkcje, zapamiętaj ich warianty oraz przeznaczenie. Na przykład na karcie dla `fscanf()` powinieneś mieć wypisane `scanf()`, `sscanf()`, `vscanf()` itd., a na odwrocie sposób działania tych funkcji.

Na koniec przejdź do podręcznika systemowego `man` i przeczytaj strony dotyczące poszczególnych wariantów omawianych tutaj funkcji, aby zebrać informacje niezbędne do umieszczenia na kartach. Na przykład informacje dotyczące dla `fscanf()` otrzymasz po wydaniu polecenia `man fscanf`.

Zadania dodatkowe

- Zmodyfikuj program w taki sposób, aby w ogóle nie zawierał wywołań funkcji `fscanf()`. Aby dane wejściowe w postaci ciągu tekstowego skonwertować na liczbę, należy użyć funkcji `atoi()`.
- Zmodyfikuj program w taki sposób, aby zawierał wywołania zwykłych funkcji `scanf()` zamiast `fscanf()`, a następnie zobacz, jaką to spowodowało różnicę.
- Popraw program, tak aby dane wejściowe nie zawierały na końcu znaku nowego wiersza oraz żadnych znaków odstępu.
- Wykorzystaj `scanf()` do przygotowania funkcji odczytującej jednorazowo tylko jeden znak i umieszczającej go w ciągu tekstowym imienia, ale bez przekraczania jego wielkości. Upewnij się o przygotowaniu ogólnej funkcji pozwalającej na pobranie wielkości ciągu tekstowego. Niezależnie od wszystkiego ciąg tekstowy powinien być zakończony znakiem `'\0'`.

ĆWICZENIE 25

Funkcje o zmiennej liczbie argumentów

W języku C można tworzyć własne wersje funkcji takich jak `printf()` i `scanf()` za pomocą funkcji o *zmiennej liczbie argumentów*. Tego rodzaju funkcje używają pliku nagłówkowego *stdarg.h*, a dzięki nim można opracować bardziej eleganckie interfejsy dla biblioteki. Zmienna liczba argumentów przydaje się w określonych rodzajach funkcji budujących pewne elementy, w funkcjach formatujących oraz wszelkich innych, które pobierają różną liczbę argumentów.

Poznanie funkcji o zmiennej liczbie argumentów *nie* jest niezbędne, aby nauczyć się tworzyć programy w C. W całej mojej karierze programisty użyłem tego rodzaju funkcji może ze 20 razy. Jednak wiedza o sposobie działania funkcji o zmiennej liczbie argumentów pomaga w debugowaniu programów oraz pozwala na jeszcze lepsze zrozumienie komputera.

Plik *ex25.c*:

```
/** OSTRZEŻENIE: To jest nowy kod i może zawierać błędy. */

#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#include "dbg.h"

#define MAX_DATA 100

int read_string(char **out_string, int max_buffer)
{
    *out_string = calloc(1, max_buffer + 1);
    check_mem(*out_string);

    char *result = fgets(*out_string, max_buffer, stdin);
    check(result != NULL, "Błąd danych wejściowych.");

    return 0;

error:
    if (*out_string) free(*out_string);
    *out_string = NULL;
    return -1;
}

int read_int(long *out_int)
{
    char *input = NULL;
    char *end = NULL;
```

```
30      int rc = read_string(&input, MAX_DATA);
31      check(rc == 0, "Nie udało się odczytać liczby.");
32
33      *out_int = strtol(input, &end, 10);
34      check((*end == '\0' || *end == '\n') &&
35          *input != '\0', "Nieprawidłowa liczba: %s", input);
36
37      free(input);
38      return 0;
39
40  error:
41      if (input) free(input);
42      return -1;
43  }
44
45  int read_scan(const char *fmt, ...)
46  {
47      int i = 0;
48      int rc = 0;
49      long *out_int = NULL;
50      char *out_char = NULL;
51      char **out_string = NULL;
52      int max_buffer = 0;
53
54      va_list argp;
55      va_start(argp, fmt);
56
57      for (i = 0; fmt[i] != '\0'; i++) {
58          if (fmt[i] == '%') {
59              i++;
60              switch (fmt[i]) {
61                  case '\0':
62                      sentinel("Nieprawidłowy format, kończy się znakiem %%.");
63                      break;
64
65                  case 'd':
66                      out_int = va_arg(argp, long *);
67                      rc = read_int(out_int);
68                      check(rc == 0, "Nie udało się odczytać liczby.");
69                      break;
70
71                  case 'c':
72                      out_char = va_arg(argp, char *);
73                      *out_char = fgetc(stdin);
74                      break;
75
76                  case 's':
77                      max_buffer = va_arg(argp, int);
78                      out_string = va_arg(argp, char **);
79                      rc = read_string(out_string, max_buffer);
80                      check(rc == 0, "Nie udało się odczytać ciągu tekstowego.");
81                      break;
82
```

```
83                 default:
84                     sentinel("Nieprawidłowy format.");
85             }
86         } else {
87             fgetc(stdin);
88     }
89
90     check(!feof(stdin) && !ferror(stdin), "Błąd danych wejściowych.");
91 }
92
93     va_end(argp);
94     return 0;
95
96 error:
97     va_end(argp);
98     return -1;
99 }
100
101 int main(int argc, char *argv[])
102 {
103     char *first_name = NULL;
104     char initial = ' ';
105     char *last_name = NULL;
106     long age = 0;
107
108     printf("Jak masz na imię? ");
109     int rc = read_scan("%s", MAX_DATA, &first_name);
110     check(rc == 0, "Nie udało się pobrać imienia.");
111
112     printf("Jaki masz inicjał? ");
113     rc = read_scan("%c\n", &initial);
114     check(rc == 0, "Nie udało się pobrać inicjału.");
115
116     printf("Jak masz na nazwisko? ");
117     rc = read_scan("%s", MAX_DATA, &last_name);
118     check(rc == 0, "Nie udało się pobrać nazwiska.");
119
120     printf("Ile masz lat? ");
121     rc = read_scan("%d", &age);
122     check(rc == 0, "Nie udało się pobrać wieku.");
123
124     printf("---- WYNIK ----\n");
125     printf("Imię: %s", first_name);
126     printf("Inicjał: '%c'\n", initial);
127     printf("Nazwisko: %s", last_name);
128     printf("Wiek: %d\n", age);
129
130     free(first_name);
131     free(last_name);
132     return 0;
133 error:
134     return -1;
135 }
```

Powyższy program jest podobny do przedstawionego w poprzednim ćwiczeniu, z wyjątkiem tego, że opracowałem tutaj własną wersję funkcji `scanf()` przeznaczoną do obsługi ciągów tekstowych w oczekiwany sposób. Działanie funkcji `main()` powinno być oczywiste, podobnie jak dwóch innych funkcji `read_string()` i `read_int()`, ponieważ nie pojawiło się w nich nic nowego.

Funkcja o zmiennej liczbie argumentów nosi nazwę `read_scan()` i została przeznaczona do wykonywania tych samych zadań co `scanf()`, ale za pomocą struktury danych `va_list` oraz pewnych makr i innych funkcji. Oto omówienie sposobu działania tej funkcji.

- Jako ostatni parametr funkcji podałem słowo kluczowe `...`, wskazując tym samym językowi C, że dana funkcja będzie pobierała dowolną liczbę argumentów po argumencie `fmt`. Przed tym argumentem można podać dowolną liczbę innych argumentów, natomiast po nim już nie.
- Po przygotowaniu niezbędnych zmiennych tworzymy jeszcze zmienną `va_list` i inicjalizujemy ją wraz z `va_start`. W ten sposób konfigurujemy zdefiniowany w pliku nagłówkowym *stdarg.h* mechanizm odpowiedzialny za obsługę zmiennej liczby argumentów funkcji.
- Używamy pętli `for` do przeprowadzenia iteracji przez ciąg tekstowy formatowania `fmt` i przetwarzamy ten sam rodzaj formatów, jaki jest stosowany przez `scanf()`, ale tutaj odbywa się to znacznie prościej. Mamy jedynie liczby całkowite, znaki i ciągi tekstowe.
- Po dotarciu do formatu używamy konstrukcji `switch` do ustalenia, jakie należy podjąć działania.
- Teraz *pobieramy* zmienną z `va_list argp` i używamy makra `va_arg(argp, TYPE)`, gdzie `TYPE` wskazuje dokładny typ, który zostanie przypisany danemu parametrowi funkcji. Wadą przedstawionego rozwiązania jest to, że jeśli nie będzie wystarczającej liczby parametrów, to niemal na pewno dojdzie do awarii programu.
- Interesującą różnicą w stosunku do funkcji `scanf()` jest przyjęcie założenia, że programista chce użyć funkcji `read_scan()` do utworzenia ciągu tekstowego wczytywanego po napotkaniu sekwencji formatującej `s`. Po podaniu tej sekwencji funkcja pobiera dwa parametry ze stosu `va_list argp`: maksymalną ilość danych odczytywanych przez funkcję oraz wskaźnik do znaku w ciągu tekstowym danych wyjściowych. Z wykorzystaniem tych informacji następuje wywołanie funkcji `read_string()` i rzeczywiste wykonanie zadania.
- Przedstawiona charakterystyka powoduje, że działanie funkcji `read_scan()` jest znacznie bardziej konsekwentne niż `scanf()`, ponieważ *zawsze* daje adres zmiennej (`&`), co pozwala na jej odpowiednie ustawienie.
- Na koniec, jeśli funkcja napotka znak niebędący poprawnym znakiem formatu, po prostu odczytuje jeden znak, aby go przeskoczyć i przejść dalej. Tak naprawdę ten znak nie ma znaczenia; będzie on pominięty.

Co powinieneś zobaczyć?

Po uruchomieniu omawianej funkcji otrzymasz dane wyjściowe podobne do przedstawionych w poprzednim ćwiczeniu.

Sesja dla ćwiczenia 25.:

```
$ make ex25
cc -Wall -g -DNDEBUG ex25.c -o ex25
$ ./ex25
Jak masz na imię? Zed
Jaki masz inicjał? A
Jak masz na nazwisko? Shaw
Ile masz lat? 37
---- WYNIK ----
Imię: Zed
Inicjał: 'A'
Nazwisko: Shaw
Wiek: 37
```

Jak to zepsuć?

Przedstawiony program powinien być znacznie bardziej odporny na przepełnienie bufora, ale nie obsługuje sformatowanych danych wejściowych tak dobrze jak `scanf()`. Jeżeli chcesz zepsuć program, spróbuj zmienić kod w taki sposób, aby zapomnieć o przekazaniu początkowej wielkości dla formatu `%s`. Spróbuj również przekazać większą ilość danych niż zdefiniowana w `MAX_DATA`, a następnie zobacz, jak pominięcie wywołania `calloc()` w funkcji `read_string()` zmienia sposób jej działania. Na koniec warto pamiętać o problemie, gdy funkcja `fgets()` otrzymuje znak nowego wiersza. Postaraj się więc naprawić to za pomocą `fgetc()`, ale pomiń znak `\0` kończący ciąg tekstowy.

Zadania dodatkowe

- Dokładnie się upewnij, że znasz przeznaczenie i sposób użycia poszczególnych zmiennych `out_`. Co ważniejsze, powinieneś wiedzieć, czym jest `out_string` oraz jak może być wskaźnikiem do wskaźnika. Dzięki temu zrozumiesz, jak ważna jest różnica między tym, kiedy należy ustawić wskaźnik, a kiedy treść.
- Utwórz funkcję o działaniu podobnym do `printf()`, ale bazującą na funkcji o zmiennej liczbie argumentów. Następnie zmodyfikuj `main()`, aby móc użyć tej nowej funkcji.
- Jak zwykle zapoznaj się ze stronami podręcznika systemowego na temat wymienionych powyżej funkcji, aby znać sposób ich działania na Twojej platformie. Niektóre platformy używają makr, natomiast inne funkcji, a jeszcze inne są pozbawione zarówno makr, jak i funkcji. Wszystko zależy od używanej platformy i kompilatora.

ĆWICZENIE 26

Projekt logfind

To jest niewielki projekt, o którego realizację powinieneś się pokusić. Jeżeli chcesz być efektywnym programistą C, musisz nauczyć się wykorzystywać posiadaną wiedzę do rozwiązywania problemów. W tym ćwiczeniu opiszę narzędzie, którego implementacja jest Twoim zadaniem. Opis celowo jest ogólnikowy, więc możesz zaimplementować właściwie co zechcesz. Kiedy zakończysz pracę, obejrzyj wideo przeznaczone dla tego ćwiczenia i zobacz, jak ja zaimplementowałem projekt, oraz porównaj swój kod źródłowy z moim.

Potraktuj ten projekt jako rzeczywistą łamigłówkę, którą naprawdę warto rozwiązać.

Specyfikacja logfind

Celem projektu jest opracowanie narzędzia `logfind`, pozwalającego na wyszukiwanie podanego tekstu w plikach dzienników zdarzeń. Będzie to więc specjalizowana wersja innego narzędzia, o nazwie `grep`, ale nasze jest przeznaczone tylko dla plików dzienników zdarzeń. Idea polega na tym, aby po wydaniu polecenia na przykład:

```
logfind zedshaw
```

narzędzie przeszukało wszystkie katalogi najczęściej używane do przechowywania plików dzienników zdarzeń i wyświetliło nazwę każdego pliku zawierającego słowo `zedshaw`.

Narzędzie `logfind` powinno się charakteryzować wymienionymi poniżej podstawowymi funkcjami:

1. Narzędzie `logfind` pobiera sekwencję słów i przyjmuje założenie, że użytkownik chce je potraktować jako „i". Dlatego też wywołanie `logfind zedshaw sprytny gość` powinno znaleźć wszystkie pliki zawierające słowa *zedshaw*, *sprytny* i *gość*.
2. Narzędzie powinno pobierać opcjonalny argument `-o`, jeśli podane parametry mają być potraktowane jako „lub".
3. Lista dozwolonych do przeszukiwania plików dzienników zdarzeń powinna być wczytywana z pliku *~/.logfind*.
4. Lista nazw plików może mieć dowolną zawartość dozwoloną do użycia przez funkcję `glob()`. O sposobie działania wymienionej funkcji dowiesz się więcej po wydaniu polecenia `man 3 glob`. Sugeruję rozpocząć od jednorodnej listy dokładnie wskazanych plików, a dopiero później dodać funkcjonalność `glob()`.
5. Dane wyjściowe w postaci dopasowanych plików powinny być wyświetlane na bieżąco, możliwie najszybciej.

I to już koniec opisu narzędzia `logfind`. Pamiętaj, że realizacja projektu może być *bardzo* trudna, więc trzeba będzie poświęcić nieco czasu. Utwórz pewien kod, przetestuj go, utwórz dodatkowy kod, przetestuj go itd. Kod twórz w niewielkich fragmentach i pracuj nad nim, aż dany fragment zacznie działać zgodnie z oczekiwaniami. Rozpocznij od najprostszego zadania, które jest wykonywane prawidłowo, a dopiero później powoli implementuj kolejne funkcje i usprawniaj je, aż wreszcie przygotujesz całość.

ĆWICZENIE 27

Programowanie kreatywne i defensywne

Skoro poznałeś już większość podstaw programowania w C, jesteś gotowy na to, aby zacząć stawać się prawdziwym programistą. W taki sposób przechodzisz drogę od początkującego do eksperta, w zakresie zarówno języka C, jak i podstawowych koncepcji w informatyce. Przedstawię kilka podstawowych struktur danych i algorytmów, które powinny być znane każdemu programiście. Następnie zaprezentuję kilka niezwykle interesujących rozwiązań, od lat wykorzystywanych przeze mnie podczas tworzenia rzeczywistego oprogramowania.

Zanim będę mógł to zrobić, najpierw muszę omówić pewne podstawowe umiejętności i idee pomagające w opracowywaniu lepszego oprogramowania. W ćwiczeniach od 27. do 31. poznasz zaawansowane koncepcje obejmujące nieco więcej niż tylko tworzenie kodu. Później zdobytą wiedzę wykorzystasz praktycznie do przygotowania biblioteki zawierającej użyteczne struktury danych.

Pierwszym krokiem na drodze ku zyskaniu większej wprawy w tworzeniu lepszego kodu w C (i właściwie w każdym języku programowania) jest nastawienie się na nowy sposób myślenia określany mianem *programowania defensywnego*. W programowaniu defensywnym przyjmuje się założenie, że zostanie popełnionych wiele błędów, a następnie na każdym możliwym kroku próbuje się ich unikać. W tym ćwiczeniu nauczę Cię, jak zacząć myśleć w kategoriach programowania defensywnego.

Nastawienie programowania kreatywnego

W tak krótkim ćwiczeniu trudno pokazać, jak być kreatywnym. Mogę Cię jednak zapewnić, że kreatywność oznacza podejmowanie ryzyka i otwarcie umysłu na nowe koncepcje, idee, wyzwania itd. Strach bardzo szybko zabija kreatywność, więc podejście przyjęte przeze mnie i przejęte przez wielu innych programistów polega na uznaniu wypadków za zdarzenia, które mają Cię wzmocnić i uchronić przed lękiem, że wyjdziesz na głupka. Oto podstawowe punkty przyjętego przeze mnie podejścia:

- Nie mogę popełnić błędu.
- Nie ma znaczenia, co myślą inni.
- Skoro jestem do tego przekonany, to musi to być doskonały pomysł.

Powyższe podejście mogę przyjmować jedynie tymczasowo, co czasami wymaga nawet zastosowania pewnych sztuczek. Jednak dzięki temu w mojej głowie pojawiają się nowe idee, znajduję kreatywne rozwiązania, otwieram się na nowe możliwości i ogólnie bez żadnego strachu wchodzę na nowe terytoria. Mając powyższe nastawienie, zwykle tworzę okropną, pierwszą wersję czegoś, aby po prostu zacząć pracę nad daną ideą.

Jednak po ukończeniu kreatywnego prototypu odrzucam to nastawienie i zabieram się do poważnej pracy, mającej na celu opracowanie niezawodnego rozwiązania. Błędem często popełnianym przez innych jest zachowanie kreatywnego nastawienia w trakcie fazy implementacji. To może prowadzić do zupełnie odmiennego, destrukcyjnego nastawienia, czyli ciemnej strony nastawienia kreatywnego:

- Możliwe jest utworzenie doskonałego oprogramowania.
- Mój umysł nie może się mylić — skoro nie znajduję żadnych błędów, to utworzyłem doskonałe oprogramowanie.
- Mój kod to ja, więc osoby, które krytykują perfekcyjność mojego kodu, tak naprawdę krytykują mnie.

To wszystko są kłamstwa. Bardzo często przeistaczasz się w programistę aż nazbyt dumnego z utworzonego przez siebie kodu, co jest naturalne, ale jednocześnie duma negatywnie wpływa na możliwość obiektywnego spojrzenia na kod i jego usprawnienie. Duma i silne przywiązanie do utworzonego kodu sprawiają, że ciągle umacniasz się w przekonaniu, że opracowany kod jest perfekcyjny. Dopóki ignorujesz wyrażaną przez innych krytykę Twojego kodu, możesz unikać jego usprawnienia i chronić swoje delikatne ego.

Kluczem do sukcesu pozostania kreatywnym *i* jednocześnie tworzenia niezawodnego oprogramowania jest przyjęcie nastawienia programowania defensywnego.

Nastawienie programowania defensywnego

Gdy już masz dobrze opanowane nastawienie programowania kreatywnego, najwyższa pora przyjąć nastawienie programowania defensywnego. Programista defensywny w zasadzie nienawidzi utworzonego przez siebie kodu i opiera się na poniższych stwierdzeniach:

- Oprogramowanie zawiera błędy.
- Ty to nie oprogramowanie, ale pozostajesz odpowiedzialny za istniejące w nim błędy.
- Nigdy nie wyeliminujesz błędów, możesz co najwyżej zmniejszyć prawdopodobieństwo ich występowania.

Takie nastawienie pozwala na zachowanie uczciwości względem swojej pracy, a także umożliwia krytyczną analizę kodu w celu jego dalszego usprawniania. Zwróć uwagę, że zgodnie z powyższym nastawieniem błędy nie tkwią *w Tobie*, to *kod* zawiera mnóstwo błędów. Mamy tym samym ważną różnicę do zrozumienia, ponieważ w ten sposób zdobywasz siłę w postaci obiektywizmu potrzebnego do przygotowania kolejnej implementacji.

Podobnie jak w przypadku nastawienia kreatywnego, nastawienie defensywne ma swoją ciemną stronę. Programiści defensywni są paranoikami, a to nie pozwala nawet na rozważenie możliwości, że można się mylić lub popełniać błędy. To może być doskonałe, gdy starasz się bezlitośnie zachować spójność i poprawność, ale jednocześnie zabija kreatywną energię i koncentrację.

8 strategii programisty defensywnego

Po przyjęciu nastawienia programowania defensywnego możesz przystąpić do modyfikacji prototypu i zastosowania ośmiu strategii, aby kod stał się maksymalnie niezawodny. Kiedy pracuję nad rzeczywistym oprogramowaniem, bezwzględnie stosuję omówione tutaj strategie i staram się wyeliminować jak najwięcej błędów, myśląc przy tym jak osoba, która chciałaby doprowadzić moje oprogramowanie do awarii.

Nigdy nie ufaj danym wejściowym. Nigdy nie ufaj otrzymanym danym i zawsze przeprowadzaj ich weryfikację.

Unikaj błędów. Jeżeli istnieje prawdopodobieństwo wystąpienia błędu, nieważne jak niewielkie, to staraj się zapobiec wystąpieniu tego błędu.

Awarie powinny być wczesne i otwarte. Pozwalaj na wczesne awarie, jasno i otwarcie informujące o tym, co i gdzie się stało oraz jak można to naprawić.

Dokumentuj założenia. Wyraźnie podawaj warunki początkowe (ang. *preconditions*), inwarianty (ang. *invariants*) i warunki końcowe (ang. *postconditions*).

Preferuj prewencję zamiast dokumentacji. Nie dokumentuj tego, co można zrobić za pomocą kodu lub czego można w ogóle uniknąć.

Automatyzuj wszystko. Zautomatyzuj wszystko, zwłaszcza testowanie.

Upraszczaj i wyjaśniaj. Zawsze staraj się uprościć kod do najmniejszej i najjaśniejszej postaci, która będzie działała bez utraty bezpieczeństwa.

Myśl logicznie. Nie stosuj się ślepo do reguł, ale także ich nie odrzucaj.

To nie są jedynie strategie, ale również podstawowe rzeczy, na których programiści powinni się koncentrować podczas podejmowania prób utworzenia dobrego, niezawodnego kodu. Zwróć uwagę, że nie podaję szczegółowo tego, co należy robić w poszczególnych strategiach. W dalszej części ćwiczenia znajdziesz dokładniejsze ich omówienie. Do wielu z nich będę też nawiązywać w niektórych kolejnych ćwiczeniach.

Zastosowanie ośmiu strategii

Powstaje pytanie, jak w rzeczywistości można zastosować te idee w działającym kodzie. Zamierzam tutaj przedstawić kilka kroków, które zawsze wykonywałem względem kodu prezentowanego w książce. Dzięki temu poszczególne koncepcje poznasz na konkretnych przykładach. Oczywiście idee te nie są ograniczone do jedynie przykładów przedstawionych w książce i powinieneś je potraktować jako rodzaj przewodnika pokazującego, jak jeszcze bardziej zwiększyć niezawodność tworzonego kodu.

Nigdy nie ufaj danym wejściowym

Spójrzmy na przykład projektu złego i dobrego. Nie mogę powiedzieć, że to jest dobry projekt, ponieważ można go jeszcze usprawnić. Przeanalizuj dwie przedstawione poniżej funkcje przeznaczone do kopiowania ciągu tekstowego oraz prostej funkcji `main()`, pozwalającej na przetestowanie tych funkcji.

Plik *ex27.c*:

```
#undef NDEBUG
#include "dbg.h"
#include <stdio.h>
#include <assert.h>

/*
* Naiwne kopiowanie po przyjęciu założenia, że wszystkie dane wejściowe zawsze
* będą prawidłowe.
* Przykład na podstawie wersji K&R C, jedynie nieco uporządkowany.
*/
void copy(char to[], char from[])
{
    int i = 0;

    // Działanie pętli while nie zakończy się, jeśli na końcu zabraknie znaku '\0'.
    while ((to[i] = from[i]) != '\0') {
        ++i;
    }
}

/*
* Bezpieczniejsza wersja sprawdzająca pod kątem wielu najczęściej występujących
* błędów.
* Długość każdego ciągu tekstowego wykorzystano do kontrolowania pętli
* i zakończenia jej działania.
*/
int safercopy(int from_len, char *from, int to_len, char *to)
{
    assert(from != NULL && to != NULL && "Zmienne from i to nie mogą mieć
    ↪wartości NULL.");
    int i = 0;
    int max = from_len > to_len - 1 ? to_len - 1 : from_len;

    // Wielkość to_len musi wynosić przynajmniej 1 bajt.
    if (from_len < 0 || to_len <= 0)
        return -1;

    for (i = 0; i < max; i++) {
        to[i] = from[i];
    }

    to[to_len - 1] = '\0';

    return i;
}

int main(int argc, char *argv[])
{
    // Postaraj się dokładnie zrozumieć, dlaczego możemy ustalić te wielkości
    char from[] = "0123456789";
```

```
47      int from_len = sizeof(from);
48
49      // Zwróć uwagę, że to jest 7 znaków plus \0.
50      char to[] = "0123456";
51      int to_len = sizeof(to);
52
53      debug("Kopiowanie '%s':%d do '%s':%d", from, from_len, to, to_len);
54
55      int rc = safercopy(from_len, from, to_len, to);
56      check(rc > 0, "Nie udało się skopiować danych.");
57      check(to[to_len - 1] == '\0', "Ciąg tekstowy jest nieprawidłowo zakończony.");
58
59      debug("Wynik: '%s':%d", to, to_len);
60
61      // Teraz spróbujemy zepsuć kod.
62      rc = safercopy(from_len * -1, from, to_len, to);
63      check(rc == -1, "Kopiowanie powinno zakończyć się niepowodzeniem #1");
64      check(to[to_len - 1] == '\0', "Ciąg tekstowy jest nieprawidłowo zakończony.");
65
66      rc = safercopy(from_len, from, 0, to);
67      check(rc == -1, "Kopiowanie powinno zakończyć się niepowodzeniem #2");
68      check(to[to_len - 1] == '\0', "Ciąg tekstowy jest nieprawidłowo zakończony.");
69
70      return 0;
71
72  error:
73      return 1;
74  }
```

Przedstawiona w powyższym programie funkcja `copy()` to typowy kod w C i stanowi źródło bardzo dużej liczby błędów przepełnienia bufora. Sama funkcja jest błędna, ponieważ zawsze wymaga prawidłowo zakończonego ciągu tekstowego C (czyli ze znakiem `'\0'` na końcu), do którego przetworzenia używana jest pętla `while`. Problem polega na tym, że zagwarantowanie spełnienia wymogu zakończenia ciągu tekstowego przez znak `'\0'` jest niezwykle trudne i jeśli taka sytuacja nie będzie prawidłowo obsłużona, spowoduje działanie pętli `while` w nieskończoność. *Kamieniem węgielnym przygotowania niezawodnego kodu jest to, aby nigdy nie tworzyć pętli, która mogłaby działać w nieskończoność*.

Funkcja `safercopy()` próbuje rozwiązać ten problem, wymagając od komponentu wywołującego podania wielkości dwóch ciągów tekstowych, które będą przetwarzane przez tę funkcję. Dzięki temu możliwe staje się przeprowadzenie pewnych operacji sprawdzających te ciągi tekstowe, niedostępne podczas użycia funkcji `copy()`. Funkcja `safercopy()` sprawdza, czy wielkości ciągów tekstowych są odpowiednie, czy mają wystarczającą ilość miejsca oraz czy *zawsze* będą prawidłowo zakończone. Niemożliwe jest, aby użycie tej funkcji doprowadziło do powstania pętli działającej w nieskończoność, jak to się może zdarzyć podczas kopiowania za pomocą `copy()`.

Omawiana tutaj idea polega na tym, aby nigdy nie ufać otrzymywanym danym wejściowym. Jeżeli przyjmiesz założenie, że funkcja otrzyma nieprawidłowo zakończony ciąg tekstowy (co zdarza się dość często), to możesz ją zaprojektować w taki sposób, aby poprawne działanie funkcji opierało się na prawidłowym zakończeniu ciągu tekstowego. Jeśli argumenty

nigdy nie powinny przyjmować wartości `NULL`, trzeba po prostu je sprawdzić pod tym kątem. Jeżeli wielkość powinna mieścić się w pewnym przedziale, należy to sprawdzić. Najlepiej przyjąć założenie o nieprawidłowym wywołaniu, a następnie spróbować jeszcze bardziej utrudnić przejście do kolejnego nieprawidłowego stanu.

Dochodzimy tutaj do następującej kwestii: tworzone przez Ciebie oprogramowanie akceptuje dane wejściowe z całego wszechświata. Sławne ostatnie zdanie programisty brzmi „Nikt tego nie zrobi". Spotykałem już programistów wypowiadających tego rodzaju stwierdzenia, a *następnego dnia* ktoś zrobił dokładnie tak lub przeprowadził udany atak na aplikację. Jeżeli sądzisz, że nikt czegoś nie zrobi, po prostu dodaj nieco kodu, aby mieć pewność, że nie będzie można skutecznie zaatakować i złamać aplikacji. Będziesz zadowolony, gdy tak postąpisz.

Przedstawiam poniżej niepełną listę rzeczy, które próbuję zrobić we wszystkich funkcjach tworzonych w języku C:

- Dla każdego parametru identyfikuję warunki wstępne, a także definiuję czy powinny spowodować awarię, czy po prostu zwrot informacji o błędzie. W przypadku tworzenia biblioteki powinieneś preferować zgłaszanie błędów zamiast występowania awarii.
- Na początku dodaję wywołania `assert()` w postaci `assert(test && "treść komunikatu");`, sprawdzające pod kątem awarii każdy warunek wstępny. Ta niewielka sztuczka przeprowadza test. Jeśli zakończy się on niepowodzeniem, to system operacyjny zwykle wywoła tak przygotowaną funkcję `assert()` zawierającą komunikat. To jest niezwykle użyteczne, kiedy próbujesz ustalić, dlaczego w danym miejscu istnieje wywołanie `assert()`.
- Dla innych warunków wstępnych definiuję zwrot kodu błędu lub używam omówionego wcześniej makra `check()` do wyświetlenia komunikatu o błędzie. Nie zdecydowałem się na użycie wymienionego makra w tym przykładzie, ponieważ mogłoby zagmatwać porównanie.
- Dokumentuję, *dlaczego* te warunki wstępne istnieją, aby po wystąpieniu błędu programista mógł określić, czy naprawdę są konieczne.
- Jeżeli modyfikujesz dane wejściowe, upewnij się o ich prawidłowym sformowaniu, gdy funkcja istnieje, lub o przerwaniu działania w przeciwnym razie.
- Zawsze sprawdzaj kody błędów używanych funkcji. Na przykład często zdarza się, że programista zapomina o sprawdzeniu kodu zwróconego przez wywołanie `fopen()` lub `fread()`, co powoduje wykorzystanie zasobów choć zwrócony kod sugeruje wystąpienie błędu. Skutkiem będzie awaria programu bądź otwarcie drogi umożliwiającej przeprowadzenie ataku na program.
- Powinieneś zachować spójność w zakresie zwracanych kodów błędu, aby móc je stosować we wszystkich funkcjach. Kiedy wpadniesz już w ten nawyk, zrozumiesz, dlaczego moje makra działają w taki, a nie inny sposób.

Wykonywanie wymienionych powyżej prostych rzeczy pozwala na wprowadzenie usprawnień podczas obsługi zasobów i chroni przed sporą ilością błędów.

Unikanie błędów

W odpowiedzi na przedstawiony wcześniej fragment kodu wielu mogłoby stwierdzić: „Istnieje niewielkie prawdopodobieństwo, że ktoś w nieprawidłowy sposób użyje kopiowania". Pomimo ogromnej liczby ataków przeprowadzanych na tego rodzaju funkcje wielu nadal jest przekonanych o niewielkim prawdopodobieństwie wystąpienia pokazanego błędu. To zabawne, ponieważ ludzie niezwykle często mylą się, określając prawdopodobieństwo wystąpienia zdarzenia. Jednak zdecydowanie lepiej radzą sobie w określeniu, czy coś jest *możliwe*. Dlatego też mogą powiedzieć, że błąd w funkcji `copy()` jest mało prawdopodobny, natomiast nie mogą zaprzeczyć *możliwości* jego wystąpienia.

Kluczowym powodem jest to, że aby coś było prawdopodobne, najpierw musi być możliwe. Określenie możliwości wystąpienia jest łatwe, ponieważ każdy może sobie wyobrazić pewne zdarzenia. Natomiast już nie tak łatwe jest określenie prawdopodobieństwa wystąpienia danego zdarzenia. Czy niebezpieczeństwo nieprawidłowego użycia funkcji `copy()` wynosi 20%, 10%, czy 1%? Kto to może wiedzieć? Konieczne jest zebranie dowodów, sprawdzenie poziomu awarii w wielu pakietach oprogramowania i prawdopodobnie zapytanie innych programistów o sposób użycia przez nich tej funkcji.

Jeżeli chcesz unikać błędów, nadal musisz starać się chronić przed tym, co jest możliwe, ale przede wszystkim skierować energię na to, co jest najbardziej prawdopodobne. Nie jest realne obsłużenie wszystkich możliwych sposobów, na jakie można zepsuć tworzone przez Ciebie oprogramowanie, ale powinieneś przynajmniej spróbować. Jednocześnie jeśli nie ograniczysz wysiłków do najbardziej prawdopodobnych zdarzeń, będziesz marnować czas na sprawdzanie kodu pod kątem nieistotnych ataków.

Poniżej przedstawiłem proces ustalenia, jak unikać błędów w oprogramowaniu.

- Wypisz wszystkie możliwe błędy, jakie mogą wystąpić, niezależnie od stopnia ich prawdopodobieństwa (oczywiście nie zapomnij także o podaniu powodu). Nie ma sensu umieszczać błędów typu „obcy będą grzebać Ci w pamięci, aby wykraść hasła dostępu".
- Każdemu możliwemu błędowi przypisz prawdopodobieństwo określające procentową liczbę operacji, które mogą spowodować pojawienie się luki w zabezpieczeniach. W przypadku obsługi żądań pochodzących z internetu podaj procentowo liczbę żądań, które mogą spowodować błąd. Natomiast w przypadku wywołań funkcji podaj procentowo liczbę takich wywołań.
- Określ liczbę godzin pracy nad kodem w celu zapewnienia ochrony przed tymi błędami. Możesz również zastosować inną metrykę. Ważne jest, aby uniknąć poświęcania czasu na pracę nad czymś niemożliwym, gdy na liście nadal znajdują się pozycje łatwe do naprawienia.
- Uporządkuj pozycje listy według wymaganego wysiłku (od najłatwiejszych do najtrudniejszych) i prawdopodobieństwa (od najwyższego do najmniejszego). W ten sposób przygotujesz listę zadań.
- Postaraj się uniknąć wszystkich błędów wymienionych na liście. Zaczynaj od usunięcia możliwości, następnie zmniejszaj prawdopodobieństwo, jeśli nie możesz czegoś wyeliminować.

- Jeżeli pozostaną błędy, których nie potrafisz usunąć, to udokumentuj je, aby ktoś inny mógł spróbować je usunąć.

Postępowanie według tej procedury daje Ci elegancką listę rzeczy do zrobienia, a co ważniejsze, pozwala na uniknięcie pracy nad niepotrzebnymi kwestiami, gdy nadal do zrobienia zostały znacznie ważniejsze rzeczy. Możesz tutaj zachować mniej lub bardziej formalne podejście. Jeżeli przeprowadzasz pełny audyt bezpieczeństwa, lepiej to zrobić z całym zespołem i przygotować elegancki arkusz kalkulacyjny. Natomiast jeśli tworzysz funkcję, to po prostu przeanalizuj kod i odpowiednie informacje umieść w komentarzach. Najważniejsze jest zaprzestanie przyjmowania założenia, że błędy się nie zdarzają, i usuwanie błędów przy jednoczesnym unikaniu marnowania wysiłku.

Awarie powinny być wczesne i otwarte

Po napotkaniu błędu w kodzie w języku C masz dwie możliwości:

- zwrot kodu błędu;
- przerwanie działania procesu.

Tak to po prostu wygląda. Dlatego też musisz zagwarantować, że awarie wystąpią szybko, będą jasno i wyraźnie udokumentowane, zostanie wyświetlony komunikat o błędzie, aby inni programiści mogli łatwo tego błędu unikać. Do tego celu służy opracowane przeze mnie makro `check()`. Po znalezieniu każdego błędu wymienione makro wyświetla odpowiedni komunikat, nazwę pliku i numer wiersza, w którym wystąpił dany błąd, a ponadto wymusza zwrot kodu błędu. Jeżeli w trakcie pracy używasz mojego makra, to ostatecznie i tak robisz, co trzeba i jak trzeba.

Osobiście preferuję zwrot kodu błędu zamiast przerwania działania programu. Jeżeli błąd jest katastrofalny w skutkach, to przerwanie działania programu okazuje się konieczne. Takich błędów na szczęście jest niewiele. Dobrym przykładem sytuacji, w której decyduję o przerwaniu działania programu, jest nieprawidłowy wskaźnik, jak to mogłeś zobaczyć w przypadku funkcji `safercopy()`. Zamiast narażać użytkownika na doświadczenie błędu naruszenia ochrony pamięci, wolę przechwycić błąd i przerwać działanie programu. Jednak jeśli często zdarza się przekazywanie wartości `NULL`, to prawdopodobnie zmienię podejście i użyję makra `check()`, aby komponent wywołujący mógł zaadaptować się do zaistniałej sytuacji, a program kontynuować działanie.

W bibliotekach staram się ze wszystkich sił, aby *nigdy* nie przerywać działania programu. Oprogramowanie wykorzystujące moją bibliotekę może podjąć decyzję o ewentualnym przerwaniu działania programu, a sama biblioteka tylko w przypadku, gdy naprawdę jest używana w skrajnie nieprawidłowy sposób.

Na koniec warto wspomnieć, że otwartość w zakresie błędów nie oznacza użycia tego samego komunikatu lub kodu dla więcej niż jednego możliwego błędu. Z takim podejściem możesz najczęściej spotkać się w przypadku zasobów zewnętrznych. Biblioteka otrzymuje informacje o błędzie w gnieździe, a następnie po prostu zgłasza „wystąpienie błędu w gnieździe". Jednak zamiast tego powinien być zgłoszony błąd w gnieździe, który można prawidłowo debugować i usunąć. Podczas projektowania systemu zgłaszania błędów upewnij się, czy są wyświetlane odpowiednie komunikaty o różnych możliwych błędach.

Dokumentuj założenia

Jeżeli stosujesz tę strategię, to przygotowujesz rodzaj kontraktu przedstawiającego oczekiwany sposób zachowania funkcji. Utworzyłeś warunki początkowe dla każdego argumentu, obsłużyłeś możliwe błędy i zapewniłeś eleganckie obsłużenie awarii. Kolejnym krokiem jest więc ukończenie kontraktu przez dodanie inwariantów i warunków końcowych.

Inwariant to warunek, który musi być spełniony w pewnych stanach podczas wykonywania funkcji. Inwarianty nie są często spotykane w prostych funkcjach, ale podczas pracy ze skomplikowanymi strukturami stają się znacznie bardziej potrzebne. Dobrym przykładem inwariantu jest warunek, aby struktura zawsze była poprawnie zainicjalizowana podczas jej użycia. Innym przykładem będzie to, że sortowana struktura danych zawsze powinna być sortowana w trakcie przetwarzania.

Warunek końcowy ma zagwarantować wartość końcową lub wynik wykonania funkcji. Może być połączony z inwariantami, ale to zwykle coś prostego, na przykład „funkcja zawsze zwraca `0` lub `-1` w przypadku błędu". Warunki końcowe są najczęściej dokumentowane, ale jeśli funkcja zwraca zaalokowany zasób, można dodać warunek końcowy, aby mieć pewność, czy nastąpił zwrot pewnej wartości, a nie `NULL`. Ewentualnie można wykorzystać `NULL` do wskazania błędu, więc wówczas warunek końcowy sprawdza, czy zasób został usunięty z pamięci po wystąpieniu błędu.

Podczas programowania w języku C inwarianty i warunki końcowe są zwykle używane częściej w dokumentacji niż w rzeczywistym kodzie bądź asercjach. Najlepszym sposobem na zapewnienie ich obsługi jest dodanie wywołań `assert()`, gdy istnieje możliwość ich wystąpienia, i udokumentowanie pozostałych. Jeżeli tak postąpisz, po napotkaniu błędu programista będzie mógł poznać założenia, jakie przyjąłeś podczas tworzenia danej funkcji.

Preferuj prewencję zamiast dokumentacji

Dość często spotykany problem wiąże się z faktem, że podczas tworzenia kodu programiści koncentrują się na jego udokumentowaniu zamiast po prostu usunięciu. Mój ulubiony znajduje się w systemie Ruby on Rails i polega na założeniu, że wszystkie miesiące mają po 30 dni. Obsługa kalendarza jest trudna, więc zamiast usunąć błąd, programiści po prostu wyświetlają niewielki komentarz informujący o celowości danego błędu i od lat odmawiają jego usunięcia. Za każdym razem, gdy ktoś zgłasza ten błąd, programiści szaleją i wykrzykują, że „ten błąd jest doskonale udokumentowany".

Dokumentacja tak naprawdę nie ma żadnego znaczenia, jeśli nie można usunąć problemu. Gdy funkcja zawiera fatalną w skutkach usterkę, nie należy jej umieszczać w programie aż do chwili usunięcia problemu. W przypadku wspomnianego wcześniej błędu w Ruby on Rails brak funkcji daty byłby lepszy niż celowe wstawianie błędnej daty, na której i tak nikt nie może polegać.

Kiedy będziesz przeprowadzać operacje porządkujące w trakcie fazy programowania defensywnego, postaraj się usunąć wszelkie usterki. Jeżeli okaże się, że więcej czasu spędzasz na dokumentowaniu kolejnych niemożliwych do rozwiązania problemów, rozważ ponowne opracowanie danej funkcjonalności lub po prostu jej usunięcie. Jeżeli *naprawdę* musisz zachować

tak koszmarnie zepsutą funkcjonalność, sugeruję jej zapisanie, udokumentowanie i znalezienie sobie nowej pracy, zanim zaczniesz być wskazywany jako winny dostarczenia tak fatalnie działającej funkcjonalności.

Automatyzuj wszystko

Jesteś programistą, co oznacza, że powinieneś pozbawić innych zajęcia dzięki zastosowaniu automatyzacji. Szczytem będzie zdjęcie z własnych barków jak największej ilości pracy przez użycie automatyzacji. Oczywiście nie chcesz wyeliminować wszystkiego, co robisz, ale jeśli poświęcasz cały dzień na ręczne przeprowadzanie testów w terminalu, to Twoim zadaniem nie jest programowanie. Zajmujesz się zapewnieniem jakości oprogramowania i prawdopodobnie powinieneś zautomatyzować operacje, aby pozbyć się tej pracy, której przecież i tak nie chcesz wykonywać.

Najłatwiejszym sposobem jest opracowanie zautomatyzowanych testów, czyli testów jednostkowych. W tej książce pokażę Ci, jak łatwo to zrobić, choć jednocześnie uniknę większości dogmatów dotyczących sposobów tworzenia testów. Skoncentruję się na ich tworzeniu, wyborze celów dla testów oraz na efektywnym ich przeprowadzaniu.

Poniżej przedstawiam najczęstsze obszary, na których programiści nie stosują automatyzacji, choć powinni:

- testowanie i sprawdzanie poprawności,
- proces kompilacji,
- wdrażanie oprogramowania,
- administrowanie systemem,
- zgłaszanie błędów.

Spróbuj poświęcić nieco czasu na automatyzację w wymienionych powyżej obszarach, a zyskasz dodatkowy czas, który możesz przeznaczyć na pracę nad oprogramowaniem. Jeżeli to zadanie sprawi Ci przyjemność, być może powinieneś pracować nad oprogramowaniem ułatwiającym automatyzację.

Upraszczaj i wyjaśniaj

Koncepcja prostoty jest dość śliskim tematem dla wielu osób, zwłaszcza tych inteligentniejszych. Ogólnie rzecz biorąc, mogą pomylić zdolność pojmowania z prostotą. Jeżeli coś będzie zrozumiane, zdecydowanie jest proste. Rzeczywisty test prostoty polega na porównaniu czegoś z czymś innym, co może być prostsze. Spotkasz osoby tworzące bardzo skomplikowane i ograniczone struktury, ponieważ wychodzą z założenia, że prostsza wersja tej samej rzeczy będzie niewłaściwa. Romans ze znacznym poziomem skomplikowania to choroba drążąca programowanie.

Możesz walczyć z tą zarazą, zaczynając od powiedzenia sobie: „Prostota jest czysta, a nie brudna, niezależnie od tego, co robią inni". Jeżeli ktoś tworzy szalony wzorzec obejmujący 19 klas w 12 interfejsach, a Ty to samo osiągasz za pomocą dwóch operacji ciągu tekstowego, to wygrywasz. W takim przypadku inni się mylą, niezależnie od tego, że to swoje skomplikowane monstrum uważają za eleganckie rozwiązanie.

Poniżej przedstawiłem najprostszy test pozwalający ocenić, która funkcja jest lepsza.

- Upewnij się, że obie funkcje nie zawierają błędów. Nie ma znaczenia jak prosta lub szybka jest funkcja, jeśli znajdują się w niej błędy.
- Jeżeli nie możesz usunąć błędów w jednej funkcji, wybierz tę drugą.
- Czy funkcje nie generują tego samego wyniku? Wybierz tę funkcję, która generuje oczekiwany wynik.
- Jeżeli funkcje generują ten sam wynik, wybierz tę z mniejszą liczbą funkcjonalności, odgałęzień lub tę, którą uważasz za prostszą.
- Upewnij się, że nie wybrałeś funkcji sprawiającej większe wrażenie. Prosta, choć nieco „brudna" zawsze jest lepsza od skomplikowanej, choć „czystej".

Być może zauważyłeś, że na końcu niemal się poddałem i pozostawiłem ostateczny wybór Twojej ocenie. To zakrawa na ironię, ale prostota to również bardzo skomplikowana kwestia, więc użycie własnego zmysłu jest najlepszym podejściem. Upewnij się jedynie, że wraz z dorośnięciem i nabyciem większego doświadczenia zweryfikujesz także to, co uważasz za „dobre".

Myśl logicznie

Ostatnia strategia jest najważniejsza, ponieważ pozwala wyrwać się z nastawienia programowania defensywnego i przejść do nastawienia kreatywnego. Programowanie defensywne jest autorytarne i może być okrutne. Zadaniem w tego rodzaju nastawieniu jest stosowanie się do reguł, bez których można coś pominąć lub się rozproszyć.

Nastawienie autorytarne ma jednak wadę w postaci wyłączenia kreatywnego myślenia. Reguły są potrzebne do wykonania zadania, ale jeśli staniesz się ich niewolnikiem, zabiją Twoją kreatywność.

Ostatnia strategia oznacza, że co pewien czas powinieneś kwestionować stosowane reguły. Przyjmij wówczas założenie, że mogą być błędne, podobnie jak w przypadku analizowanego oprogramowania. Zwykle robię sobie przerwę od programowania i pozwalam odpocząć od reguł po sesji programowania defensywnego. Następnie jestem gotowy do wykonania pewnej pracy kreatywnej lub dalszego programowania defensywnego, jeśli zachodzi potrzeba.

Kolejność nie ma znaczenia

Na koniec chciałbym jeszcze dodać pewną rzecz dotyczącą przedstawionej powyżej filozofii — nie musisz ściśle stosować się do kolejności „TWORZENIE! OBRONA! TWORZENIE! OBRONA!". Być może na początku zechcesz użyć takiego rozwiązania, ale odpowiednie nastawienie wybieram w zależności od tego, co muszę zrobić. Często nastawienia nakładają się na siebie, bez wyraźnie zdefiniowanych granic.

Nie uważam, że którekolwiek z omówionych nastawień jest lepsze od drugiego. Nie istnieje również wyraźna granica między nimi. Podczas programowania musisz zachować kreatywność i dyscyplinę, więc pracuj nad obydwoma nastawieniami i nieustannie usprawniaj je.

Zadania dodatkowe

- Kod przedstawiony dotąd w książce (oraz w jej pozostałej części) potencjalnie łamie omówione reguły. Powróć do wcześniejszych ćwiczeń i zdobytą tutaj wiedzę wykorzystaj do sprawdzenia, czy potrafisz poprawić dotychczasowy kod lub znaleźć w nim błędy.
- Wybierz projekt typu open source i przeprowadź na nim podobną analizę kodu. Wyślij poprawkę po usunięciu błędu w kodzie.

ĆWICZENIE 28

Pośrednie pliki Makefile

W trzech kolejnych ćwiczeniach utworzymy szkielet katalogu projektu przeznaczony do późniejszej budowy własnych programów w języku C. Szkielet ten będziemy wykorzystywać w pozostałej części książki. To ćwiczenie koncentruje się tylko na pliku *Makefile*, abyś mógł dokładnie zrozumieć jego przeznaczenie.

Celem budowanej struktury jest ułatwienie opracowywania średniej wielkości programów bez konieczności odwoływania się do narzędzi konfiguracyjnych. Jeżeli szkielet zostanie przygotowany prawidłowo, możesz naprawdę daleko zajść, używając jedynie GNU `make` i pewnych niewielkich skryptów powłoki.

Podstawowa struktura projektu

Zaczynamy od utworzenia katalogu o nazwie *c-skeleton* i umieszczenia w nim podstawowych plików i katalogów niezbędnych w wielu projektach. Oto stosowany przeze mnie zestaw startowy.

Sesja dla ćwiczenia 28.:

```
$ mkdir c-skeleton
$ cd c-skeleton/
$ touch LICENSE README.md Makefile
$ mkdir bin src tests
$ cp dbg.h src/ # Ten plik pochodzi z ćwiczenia 19.
$ ls -l
total 8
-rw-r--r-- 1 zedshaw staff 0 Mar 31 16:38 LICENSE
-rw-r--r-- 1 zedshaw staff 1168 Apr 1 17:00 Makefile
-rw-r--r-- 1 zedshaw staff 0 Mar 31 16:38 README.md
drwxr-xr-x 2 zedshaw staff 68 Mar 31 16:38 bin
drwxr-xr-x 2 zedshaw staff 68 Apr 1 10:07 build
drwxr-xr-x 3 zedshaw staff 102 Apr 3 16:28 src
drwxr-xr-x 2 zedshaw staff 68 Mar 31 16:38 tests
$ ls -l src
total 8
-rw-r--r-- 1 zedshaw staff 982 Apr 3 16:28 dbg.h
$
```

Na końcu wydałem polecenie `ls -l`, aby pokazać ostateczną zawartość katalogu szkieletu.

Oto omówienie poszczególnych elementów mojego zestawu startowego.

LICENSE. Jeżeli udostępniasz kod źródłowy własnych projektów, powinieneś dołączyć także licencję. Jeśli tego nie zrobisz, domyślnie kod jest uznawany za chroniony prawami autorskimi i nikt inny nie ma do niego żadnych praw.

README.md. Ten plik zawiera informacje podstawowe o sposobie użycia projektu. Rozszerzenie *.md* wskazuje na plik w formacie Markdown.

Makefile. Podstawowy plik wraz z konfiguracją kompilacji projektu.

bin. Katalog przeznaczony dla programów, które mogą być uruchamiane przez użytkowników. Katalog ten jest zwykle pusty, a jeśli go nie będzie, to zostanie utworzony przez plik *Makefile*.

build. Katalog przeznaczony na biblioteki i inne komponenty. Podobnie jak w przypadku *bin* także ten katalog jest zwykle pusty, a jeśli go nie będzie, to zostanie utworzony przez plik *Makefile*.

src. Katalog przeznaczony na kod źródłowy, najczęściej w plikach z rozszerzeniami *.c* i *.h*.

tests. Katalog przeznaczony na zautomatyzowane testy.

src/dbg.h. Do katalogu *src* skopiowałem plik *dbg.h* utworzony w ćwiczeniu 19.

Teraz omówię poszczególne elementy znajdujące się w tym katalogu szkieletowym, abyś mógł dokładnie poznać ich przeznaczenie i sposób działania.

Makefile

Pierwszym omówionym elementem będzie plik *Makefile*, ponieważ jeśli dokładnie go poznasz, wszystko pozostałe stanie się łatwiejsze do opanowania. Plik *Makefile* użyty w tym ćwiczeniu jest znacznie bardziej szczegółowy niż stosowany dotąd. Dokładne omówienie poszczególnych jego fragmentów przedstawię po tym, jak wpiszesz jego zawartość.

Plik *Makefile*:

```
 1 CFLAGS=-g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG $(OPTFLAGS)
 2 LIBS=-ldl $(OPTLIBS)
 3 PREFIX?=/usr/local
 4
 5 SOURCES=$(wildcard src/**/*.c src/*.c)
 6 OBJECTS=$(patsubst %.c,%.o,$(SOURCES))
 7
 8 TEST_SRC=$(wildcard tests/*_tests.c)
 9 TESTS=$(patsubst %.c,%,$(TEST_SRC))
10
11 TARGET=build/libYOUR_LIBRARY.a
12 SO_TARGET=$(patsubst %.a,%.so,$(TARGET))
13
14 # Docelowe wersje programu (tak zwane cele).
15 all: $(TARGET) $(SO_TARGET) tests
16
17 dev: CFLAGS=-g -Wall -Isrc -Wall -Wextra $(OPTFLAGS)
18 dev: all
19
20 $(TARGET): CFLAGS += -fPIC
21 $(TARGET): build $(OBJECTS)
```

```
22      ar rcs $@ $(OBJECTS)
23      ranlib $@
24 $(SO_TARGET): $(TARGET) $(OBJECTS)
25      $(CC) -shared -o $@ $(OBJECTS)
26
27 build:
28      @mkdir -p build
29      @mkdir -p bin
30
31 # Testy jednostkowe.
32 .PHONY: tests
33 tests: CFLAGS += $(TARGET)
34 tests: $(TESTS)
35      sh ./tests/runtests.sh
36
37 # Operacje porządkujące.
38 clean:
39      rm -rf build $(OBJECTS) $(TESTS)
40      rm -f tests/tests.log
41      find . -name "*.gc*" -exec rm {} \;
42      rm -rf `find . -name "*.dSYM" -print`
43
44 # Instalacja.
45 install: all
46      install -d $(DESTDIR)/$(PREFIX)/lib/
47      install $(TARGET) $(DESTDIR)/$(PREFIX)/lib/
48
49 # Sprawdzenie.
50 check:
51      @echo Pliki zawierające potencjalnie niebezpieczne funkcje.
52      @egrep '[^_.>a-zA-Z0-9](str(n?cpy|n?cat|xfrm|n?dup|str|pbrk|tok|_)\
53          |stpn?cpy|a?sn?printf|byte_)' $(SOURCES) || true
```

Pamiętaj o konieczności stosowania w pliku *Makefile* wcięć tworzonych za pomocą tabulatorów, a nie spacji. Używany przez Ciebie edytor tekstu powinien o tym wiedzieć i robić to prawidłowo. Jeśli tak nie jest, wybierz inny edytor tekstu. Żaden programista nie powinien używać edytora, który nie sprawdza się nawet przy tak prostym zadaniu.

Nagłówek

Przedstawiony powyżej plik *Makefile* jest przeznaczony do niezawodnej kompilacji biblioteki na niemal każdej platformie i wykorzystuje funkcje specjalne GNU `make`. Ponieważ później będziemy pracować nad wspomnianą biblioteką, więc poniżej znajdziesz omówienie poszczególnych fragmentów pliku. Rozpoczynamy od nagłówka.

- *Makefile:1*. Tutaj znajdują się standardowe opcje `CFLAGS` definiowane we wszystkich projektach oraz kilka dodatkowych, które będą niezbędne do kompilacji bibliotek. *Być może wystąpi konieczność modyfikacji tych ustawień w celu dopasowania do używanej platformy*. Zwróć uwagę na zmienną `OPTFLAGS` na końcu sekcji; zmienna ta pozwala programistom na zmianę opcji kompilacji, gdy zachodzi potrzeba.

- *Makefile:2*. Te opcje są używane podczas linkowania biblioteki. Oczywiście istnieje możliwość zmiany sposobu linkowania dzięki użyciu zmiennej `OPTLIBS`.
- *Makefile:3*. Ten kod powoduje ustawienie zmiennej *opcjonalnej* o nazwie `PREFIX`, która przyjmie podaną wartość tylko wtedy, gdy programista nie dostarczy własnego ustawienia `PREFIX`. Do tego celu służy notacja `?=`.
- *Makefile:5*. Ten skomplikowany wiersz fantastycznego kodu *dynamicznie* tworzy zmienną `SOURCES` dzięki użyciu znaku wieloznacznego * podczas wyszukiwania wszystkich plików *.c* w katalogu *src*. Konieczne jest podanie zarówno *src*/**/*.c*, jak i *src*/*.c*, aby narzędzie GNU `make` uwzględniło pliki w katalogu *src* oraz jego podkatalogach.
- *Makefile:6*. Po przygotowaniu listy plików źródłowych można użyć `patsubst` w celu pobrania listy `SOURCES` plików *.c* i utworzenia *nowej* listy wszystkich plików obiektowych. Odbywa się to przez nakazanie `patsubst` zmianę wszystkich rozszerzeń %.*c* na %.*o*, a następnie przypisanie ich do `OBJECTS`.
- *Makefile:8*. Ponownie używamy znaku wieloznacznego w celu wyszukania wszystkich plików źródłowych testów zawierających testy jednostkowe. To są zupełnie inne pliki niż biblioteki zawierające kod źródłowy.
- *Makefile:9*. Znów wykorzystujemy sztuczkę opartą na `patsubst` do dynamicznego pobrania celów `TESTS`. Pozbywamy się rozszerzenia *.c*, aby pełny program zachował tę samą nazwę. Wcześniej podczas tworzenia plików obiektowych rozszerzenie *.c* zastępowaliśmy rozszerzeniem *.o*.
- *Makefile:11*. Na końcu wskazujemy, że docelowy program to *build*/*libNAZWA-BIBLIOTEKI.a*, zmienisz to podczas rzeczywistej kompilacji biblioteki.

W ten sposób omówiliśmy początek pliku *Makefile*. Chciałbym w tym miejscu wyjaśnić, co rozumiem przez zwrot „pozwala programistom na zmianę opcji kompilacji, gdy zachodzi potrzeba". Po wydaniu polecenia `make` możesz zrobić coś takiego:

```
# OSTRZEŻENIE! To jedynie demonstracja, w rzeczywistości tak się nie robi.
# Poniższe polecenie instaluje bibliotekę w katalogu /tmp.
$ make PREFIX=/tmp install
# Poniższe polecenie nakazuje dodanie pthread.
$ make OPTFLAGS=-pthread
```

Po przekazaniu opcji dopasowanych do tego samego rodzaju zmiennych, jakie znajdują się w pliku *Makefile*, zostaną uwzględnione podczas kompilacji. Tym samym zyskujesz możliwość zmiany sposobu działania *Makefile*. Pierwsza zmienna modyfikuje `PREFIX`, aby biblioteka została zainstalowana w katalogu */tmp*. Z kolei druga ustawia `OPTFLAGS` w taki sposób, aby została użyta opcja `-pthread`.

Docelowe wersje programu

Kontynuujemy omawianie pliku *Makefile* i przechodzimy do fragmentu odpowiedzialnego za utworzenie plików obiektowych i docelowych wersji programu.

Makefile:15. Pierwszy cel mamy wtedy, gdy narzędzie `make` jest wykonywane wraz z opcjami domyślnymi, bez podania konkretnego celu. Tutaj celowi nadaliśmy nazwę `all:` oraz `$(TARGET) tests` jako cele do kompilacji. Odszukaj zmienną `TARGET`, a przekonasz się, że masz do czynienia z biblioteką. Dlatego też `all:` spowoduje kompilację biblioteki. Z kolei cel `tests` został skonfigurowany w dalszej części pliku *Makefile* i powoduje kompilację testów jednostkowych.

Makefile:17. W tym miejscu mamy cel przeznaczony do utworzenia „wersji programistycznej" programu i jednocześnie prezentację techniki zmiany opcji dla wyłącznie jednego celu. Jeżeli decyduję się na przygotowanie „wersji programistycznej" programu, oczekuję, że opcja `CFLAGS` będzie zawierała opcje takie jak `-Wextra`, które są użyteczne podczas wyszukiwania błędów. Po umieszczeniu ich jako opcji w wierszu celu, biorąc pod uwagę inny wiersz kodu dotyczący celu pierwotnego (tutaj `all`), mamy możliwość zmiany ustawionych opcji. Z tej możliwości korzystamy przy ustawianiu różnych opcji dla poszczególnych platform, które tego wymagają.

Makefile:20. Kompilacja biblioteki `TARGET`, niezależnie od tego, gdzie się znajduje. Używana jest również ta sama sztuczka z wiersza 15., pozwalająca na zmianę opcji i sposoby ich modyfikacji podczas działania narzędzia. W omawianym przypadku dodajemy `-fPIC` dla biblioteki, używając składni `+=`.

Makefile:21. Przechodzimy do rzeczywistych celów. Tutaj najpierw następuje utworzenie katalogu *build*, a następnie kompilacja wszystkich obiektów (`OBJECTS`).

Makefile:22. Tutaj mamy wykonanie polecenia `ar` faktycznie tworzącego `TARGET`. Składnia `$@ $(OBJECTS)` ma następujące znaczenie: „Umieść tam cel dla tego źródła *Makefile*, a następnie wszystkie obiekty (`OBJECTS`)". W omawianym przykładzie `$@` jest mapowane na `$(TARGET)` w wierszu 20., co z kolei jest mapowane na *build/libNAZWA-BIBLIOTEKI.a*. Wydaje się, że trzeba tutaj monitorować wiele rzeczy, i tak faktycznie jest, ale kiedy to opanujesz, to po zmianie po prostu `TARGET` na początku pliku będziesz mógł skompilować zupełnie nową bibliotekę.

Makefile:23. W celu utworzenia biblioteki konieczne jest wykonanie `ranlib` względem `TARGET` — biblioteka zostanie skompilowana.

Makefile:24 – 25. Utworzenie katalogów *build* i *bin*, o ile jeszcze nie istnieją. Do katalogów odwołujemy się w wierszu 20., gdzie używamy celu `build` i upewniamy się o utworzeniu katalogu *build*.

W ten sposób przygotowaliśmy wszystko, co jest niezbędne do kompilacji oprogramowania. Przystępujemy teraz do przygotowania infrastruktury pozwalającej na kompilację i przeprowadzanie testów jednostkowych w zautomatyzowany sposób.

Testy jednostkowe

Język C różni się od innych języków programowania, ponieważ znacznie łatwiej można utworzyć niewielki program dla każdej testowanej rzeczy. Niektóre frameworki testów próbują emulować stosowaną w innych językach koncepcję modułu i dynamiczne wczytywanie, ale to nie sprawdza się zbyt dobrze w C. Ponadto takie podejście jest niepotrzebne, ponieważ można utworzyć pojedynczy program wykonywany dla każdego testu.

Najpierw omówię fragment pliku *Makefile* poświęcony testom, a następnie zajrzysz do zawartości katalogu *tests*/ faktycznie odpowiadającej za testy.

Makefile:32. Jeżeli masz cel, który nie jest rzeczywisty, ale jedynie będzie katalogiem lub plikiem o podanej nazwie, to konieczne jest dodanie `.PHONY:`, aby narzędzie `make` ignorowało plik i zawsze było wykonywane.

Makefile:33. Wykorzystuję znaną Ci już sztuczkę z modyfikacją zmiennej `CFLAGS`, aby dodać `TARGET` do kompilacji. W ten sposób każdy program testu będzie połączony z biblioteką `TARGET`. W omawianym przypadku do połączenia dodajemy `build/libNAZWA-BIBLIOTEKI.a`.

Makefile:34. W tym miejscu jest zdefiniowany rzeczywisty cel `tests:`, który zależy od wszystkich programów wymienionych w zmiennej `TESTS` utworzonej na początku pliku *Makefile*. Znaczenie tego wiersza jest następujące: „Wykorzystaj wiedzę o kompilacji programów i bieżące ustawienia `CFLAGS` do kompilacji wszystkich programów wymienionych w `TESTS`".

Makefile:35. Po kompilacji wszystkich programów wymienionych w `TESTS` wykonujemy prosty skrypt powłoki (utworzymy go za chwilę), który wie, w jaki sposób uruchomić te programy i wyświetlić wygenerowane przez nie dane wyjściowe. Ten wiersz jest faktycznie wykonywany i dzięki niemu otrzymujesz wynik przeprowadzenia testów.

Aby testy jednostkowe działały, konieczne jest utworzenie niewielkiego skryptu powłoki potrafiącego uruchamiać programy. W katalogu *tests* utwórz więc skrypt *runtests.sh* o przedstawionej poniżej zawartości.

Plik *runtests.sh*:

```
 1 echo "Wykonywanie testów jednostkowych:"
 2
 3 for i in tests/*_tests
 4 do
 5     if test -f $i
 6     then
 7         if $VALGRIND ./$i 2>> tests/tests.log
 8         then
 9             echo $i PASS
10         else
11             echo "BŁĄD w teście $i: zajrzyj do pliku tests/tests.log"
12             echo "------"
13             tail tests/tests.log
14             exit 1
15         fi
16     fi
17 done
18
19 echo ""
```

Z tego pliku będziemy korzystać później, gdy będzie omawiany sposób działania testów jednostkowych.

Operacje porządkujące

Mamy już w pełni działające testy jednostkowe. Kolejnym krokiem jest przeprowadzenie operacji porządkujących, gdy zachodzi potrzeba wyzerowania wszystkiego.

Makefile:38. Cel `clean:` rozpoczyna wykonywanie operacji porządkujących, gdy trzeba uprzątnąć projekt.

Makefile:39 – 42. Polecenia te usuwają większość śmieci pozostawionych przez różne uruchamiane kompilatory i narzędzia. Usuwany jest również katalog *build*. Wykorzystujemy też sztuczkę, aby na końcu pozbyć się także dziwnych katalogów **.dSYM*, pozostawionych przez Xcode na potrzeby debugowania.

Jeżeli okaże się, że pozostały jeszcze inne śmieci do usunięcia, wystarczy po prostu zmodyfikować używaną przez ten cel listę elementów do usunięcia.

Instalacja

Po przeprowadzeniu wcześniejszych etapów potrzebna jest nam możliwość zainstalowania projektu. Skoro ten plik *Makefile* jest przeznaczony do kompilacji biblioteki, wystarczy umieścić odpowiednią nazwę katalogu w `PREFIX`, najczęściej będzie to */usr/local/lib*.

Makefile:45. Ten wiersz powoduje zależność celu `install:` od `all:`, więc wydanie polecenia `make install` powoduje kompilację wszystkiego w projekcie.

Makefile:46. Program `install` jest używany do utworzenia katalogu *lib*, jeśli jeszcze nie istnieje. W omawianym przykładzie próbujemy przeprowadzić jak najbardziej elastyczną instalację, używając dwóch zmiennych będących konwencjami dla instalacji. Pierwsza to `DESTDIR`, przeznaczona dla instalatora umieszczającego skompilowane pliki w bezpiecznych lub dziwnych miejscach, aby przygotować pakiety. Druga to `PREFIX`, wykorzystywana, gdy użytkownik chce zainstalować projekt w innej lokalizacji niż */usr/local*.

Makefile:47. Następnie za pomocą `install` przeprowadzana jest faktyczna instalacja biblioteki w jej miejscu docelowym.

Przeznaczeniem programu `install` jest zapewnienie, że użytkownik ma odpowiednie uprawnienia. Do wykonania polecenia `make install` zwykle konieczne są uprawnienia użytkownika *root*, a więc typowy proces kompilacji odbywa się po wydaniu polecenia `make && sudo make install`.

Sprawdzenie

Ostatni fragment omawianego pliku *Makefile* można uznać za bonus. Zamieszczam go w moich projektach C, ponieważ pomaga w wykryciu użycia złych funkcji w języku C. Do wspomnianych funkcji zaliczam przede wszystkim funkcje dotyczące ciągów tekstowych oraz inne funkcje niechronionego bufora.

Makefile:50. Cel `check:`, pozwala na wykonanie makra `check()`, gdy wystąpi potrzeba.

Makefile:51. To jest po prostu sposób na wyświetlenie komunikatu. Polecenie `@echo` nakazuje narzędziu `make` wyświetlenie nie polecenia, ale danych wyjściowych.

Makefile:52. Ten wiersz zawiera przypisanie zmiennej w postaci ogromnego wyrażenia regularnego wyszukującego złe funkcje, takie jak `strcpy()`.

Makefile:53. Wykonanie polecenia `egrep` na plikach kodu źródłowego w celu wyszukania wszelkich złych wzorców. Umieszczony na końcu polecenia fragment `|| true` stanowi rodzaj zabezpieczenia przed uznaniem przeze mnie, że działanie `egrep` zakończyło się niepowodzeniem, jeśli nie zostaną znalezione żadne błędy.

Kiedy powyższe polecenie zostanie wykonane, spowoduje dziwny efekt wyświetlenia błędu, gdy tak naprawdę nie dzieje się nic złego.

Co powinieneś zobaczyć?

Przed nami są jeszcze dwa ćwiczenia, zanim zakończymy przygotowywanie szkieletu katalogu projektu, ale już teraz możemy przetestować pewne funkcje pliku *Makefile*.

Sesja dla ćwiczenia 28.:

```
$ make clean
rm -rf build
rm -f tests/tests.log
find . -name "*.gc*" -exec rm {} \;
rm -rf `find . -name "*.dSYM" -print`
$ make check
$ make
```

Uruchomienie celu `clean:` jest możliwe, ponieważ nie mamy żadnych plików kodu źródłowego w katalogu *src*, i dlatego polecenia wymienionego celu tak naprawdę nie będą wykonane. Pracę dokończymy w kolejnym ćwiczeniu.

Zadania dodatkowe

- Spróbuj faktycznie wykorzystać przygotowany tutaj plik *Makefile* przez umieszczenie w katalogu *src* plików nagłówkowego i kodu źródłowego, a następnie rozpocznij kompilację biblioteki. Plik kodu źródłowego nie powinien wymagać funkcji `main()`.
- Spróbuj ustalić, jakie funkcje są wyszukiwane przez cel `check:` na podstawie podanego wyrażenia regularnego.
- Jeżeli nie przeprowadzasz zautomatyzowanych testów jednostkowych, poczytaj nieco na ten temat, aby być przygotowanym na później.

ĆWICZENIE 29

Biblioteki i linkowanie

Punktem centralnym każdego programu w C jest możliwość połączenia z bibliotekami dostarczanymi przez system operacyjny. Linkowanie to proces pozwalający na wzbogacenie programu o dodatkową funkcjonalność, przygotowaną przez innych i oferowaną przez system operacyjny. Wykorzystywałeś już biblioteki standardowe dołączane automatycznie. Teraz przechodzę do wyjaśnienia różnych typów bibliotek oraz ich przeznaczenia.

Trzeba zacząć od stwierdzenia, że w każdym języku programowania biblioteki są kiepsko zaprojektowane. Zupełnie nie wiem dlaczego, ale można odnieść wrażenie, że twórcy języka uznają linkowanie za coś, co można poprawić później. Biblioteki są zwykle zagmatwane, trudne w użyciu, nie pozwalają na prawidłowe wersjonowanie i ostatecznie są dołączane odmiennie w różnych miejscach.

Język C nie jest pod tym względem wyjątkiem, ale sposób linkowania i przygotowania samych bibliotek w C jest pochodną tego, jak system operacyjny UNIX i jego formaty plików wykonywalnych zostały zaprojektowane wiele lat temu. Poznanie przeprowadzanego przez kompilator C procesu linkowania na pewno pomoże Ci w dokładniejszym poznaniu sposobu działania systemu operacyjnego i uruchamiania przez niego programów.

Wyróżniamy dwa podstawowe typy bibliotek:

Statyczna. Tego rodzaju bibliotekę utworzyłeś za pomocą wywołań `ar` i `ranlib` podczas pracy nad *libNAZWA-BIBLIOTEKI.a* w poprzednim ćwiczeniu. Ten rodzaj biblioteki to zaledwie kontener dla zbioru plików obiektowych (*.o*) i ich funkcji. Podczas tworzenia programów możesz taką bibliotekę potraktować jako jeden wielki plik *.o*.

Dynamiczna. Tego rodzaju biblioteki mają rozszerzenie *.so*, *.dll* lub milion innych w systemie OS X, w zależności od wersji lub osoby, która danego dnia nad określonym plikiem pracowała. Mówiąc całkiem poważnie, w systemie OS X mamy rozszerzenia takie jak *.dylib*, *.bundle*, *.framework*, choć między nimi nie ma zbyt wielu różnic. Pliki te są skompilowane i umieszczone w pewnych dobrze znanych katalogach. Po uruchomieniu programu system operacyjny dynamicznie wczytuje tego rodzaju pliki, a następnie „w locie" łączy je z programem.

Biblioteki statyczne lubię stosować w małych i średnich projektach, ponieważ łatwiej z nimi pracować, a poza tym działają w wielu systemach operacyjnych. Ponadto lubię umieszczać maksymalną ilość kodu w bibliotece statycznej, co pozwala na późniejsze jej dołączenie do testów jednostkowych i plików programów, gdy zachodzi potrzeba.

Biblioteki dynamiczne sprawdzają się dobrze w dużych systemach, gdy ilość miejsca jest ograniczona, lub w przypadku istnienia ogromnej liczby programów opierających działanie na tej samej funkcjonalności. W takiej sytuacji nie chcemy statycznie łączyć z każdym programem całego kodu najczęściej używanych funkcji. Umieszczenie tego kodu w bibliotece dynamicznej pozwala na jego wczytanie tylko jednokrotnie i udostępnienie wszystkim oczekującym go programom.

W poprzednim ćwiczeniu dowiedziałeś się, jak przygotować podstawy dla biblioteki statycznej (plik z rozszerzeniem *.a*), z której będziemy korzystać w pozostałej części książki. W tym ćwiczeniu pokażę, jak utworzyć prostą bibliotekę *.so* i dynamicznie wczytywać ją za pomocą wywołania `dlopen()` w systemie operacyjnym UNIX. Przedstawię ręczne wykonanie poszczególnych operacji, abyś mógł dokładnie zobaczyć, jak cały proces przebiega w rzeczywistości. Następnie w ramach zadania dodatkowego do utworzenia biblioteki użyjesz szkieletu *c-skeleton*.

Dynamiczne wczytywanie biblioteki współdzielonej

Pracę rozpoczynamy od przygotowania dwóch plików kodu źródłowego. Pierwszy zostanie użyty do faktycznego utworzenia biblioteki *libex29.so*, natomiast drugi do utworzenia programu *ex29*, który będzie wczytywał tę bibliotekę i korzystał z oferowanych przez nią funkcji.

Plik *libex29.c*:

```
 1 #include <stdio.h>
 2 #include <ctype.h>
 3 #include "dbg.h"
 4
 5
 6 int print_a_message(const char *msg)
 7 {
 8     printf("CIĄG TEKSTOWY: %s\n", msg);
 9
10     return 0;
11 }
12
13
14 int uppercase(const char *msg)
15 {
16     int i = 0;
17
18     // BŁĄD: problem z zakończeniem ciągu tekstowego znakiem \0.
19     for(i = 0; msg[i] != '\0'; i++) {
20         printf("%c", toupper(msg[i]));
21     }
22
23     printf("\n");
24
25     return 0;
26 }
27
28 int lowercase(const char *msg)
29 {
30     int i = 0;
31
32     // BŁĄD: problem z zakończeniem ciągu tekstowego znakiem \0.
```

```
    for(i = 0; msg[i] != '\0'; i++) {
        printf("%c", tolower(msg[i]));
    }

    printf("\n");

    return 0;
}

int fail_on_purpose(const char *msg)
{
    return 1;
}
```

W powyższym kodzie nie znajduje się nic nadzwyczajnego. Mamy tam kilka błędów, które pozostawiłem celowo, aby sprawdzić, czy zwrócisz na to uwagę. Poprawieniem tych błędów zajmiemy się nieco później.

Naszym celem jest wykorzystanie funkcji `dlopen()`, `dlsym()` i `dlclose()` do pracy z funkcjami oferowanymi przez tę bibliotekę.

Plik *ex29.c*:

```
#include <stdio.h>
#include "dbg.h"
#include <dlfcn.h>

typedef int (*lib_function) (const char *data);

int main(int argc, char *argv[])
{
    int rc = 0;
    check(argc == 4, "UŻYCIE: e29 libex29.so funkcja dane");

    char *lib_file = argv[1];
    char *func_to_run = argv[2];
    char *data = argv[3];

    void *lib = dlopen(lib_file, RTLD_NOW);
    check(lib != NULL, "Nie udało się otworzyć biblioteki %s: %s", lib_file,
        dlerror());

    lib_function func = dlsym(lib, func_to_run);
    check(func != NULL,
        "Nie znaleziono funkcji %s w bibliotece %s: %s", func_to_run,
        lib_file, dlerror());

    rc = func(data);
    check(rc == 0, "Funkcja %s zwróciła %d dla danych: %s", func_to_run,
        rc, data);

    rc = dlclose(lib);
    check(rc == 0, "Nie udało się zamknąć %s", lib_file);
```

```
31
32      return 0;
33
34  error:
35      return 1;
36  }
```

Teraz przedstawię dokładne omówienie powyższego kodu źródłowego, abyś dokładnie wiedział, co się dzieje w tym niewielkim, choć niezwykle użytecznym kodzie.

ex29.c:5. Ta definicja funkcji będzie później używana do wywoływania funkcji w bibliotece. Nie ma tutaj nic nowego, ale upewnij się, że działanie tego polecenia jest zrozumiałe.

ex29.c:16. Po przeprowadzeniu zwykłej konfiguracji dla małego programu wykorzystuję funkcję `dlopen()` w celu wczytania biblioteki wskazanej przez `lib_file`. Wartością zwrotną funkcji jest uchwyt, z którego będziemy korzystać później, a samo działanie przypomina otworzenie pliku.

ex29.c:17 – 18. W przypadku wystąpienia błędu przeprowadzamy standardowe sprawdzenie i kończymy działanie programu. Zwróć uwagę na użycie funkcji `dlerror()` na końcu w celu ustalenia błędu związanego z biblioteką.

ex29.c:20. Wywołanie `dlsym()` służy w celu pobrania funkcji z biblioteki za pomocą *ciągu tekstowego* jej nazwy i umieszczenie jej w `func_to_run`. Ten fragment kodu ma potężne możliwości, ponieważ dynamicznie pobieramy wskaźnik do funkcji, opierając się na ciągu tekstowym pobranym z tablicy `argv` zawierającej parametry podane w powłoce.

ex29.c:25. Następnie wywołujemy wskazaną funkcję i sprawdzamy jej wartość zwrotną.

ex29.c:29. Na końcu zamykamy bibliotekę, podobnie jak ma to miejsce podczas pracy z plikiem. Zwykle biblioteka pozostaje otwarta przez cały czas działania programu, więc jej zamknięcie na końcu nie jest tutaj aż tak użyteczne. W tym miejscu po prostu pokazuję tę operację.

Co powinieneś zobaczyć?

Skoro już znasz sposób działania programu, poniżej przedstawiam sesję, w trakcie której skompilowałem bibliotekę *libex29.so* i program *ex29*, a następnie je wykorzystałem. Dokładnie przeanalizuj tę sesję, aby zobaczyć, jak wygląda ręczna kompilacja biblioteki.

Sesja dla ćwiczenia 29.:

```
# Kompilacja pliku biblioteki i utworzenie pliku z rozszerzeniem .so.
# Na niektórych platformach może wystąpić konieczność użycia -fPIC.
# Dodaj tę opcję, jeśli otrzymasz błędy.
$ cc -c libex29.c -o libex29.o
$ cc -shared -o libex29.so libex29.o

# Kompilacja programu używającego biblioteki.
```

```
$ cc -Wall -g -DNDEBUG ex29.c -ldl -o ex29

# Kilka prób użycia biblioteki.
$ ex29 ./libex29.so print_a_message "witaj, świecie!"
-bash: ex29: command not found
$ ./ex29 ./libex29.so print_a_message "witaj, świecie!"
CIĄG TEKSTOWY: witaj, świecie!
$ ./ex29 ./libex29.so uppercase "witaj, świecie!"
WITAJ, ŚWIECIE!
$ ./ex29 ./libex29.so lowercase "WITAJ, śWiEcIe!"
witaj, świecie!
$ ./ex29 ./libex29.so fail_on_purpose "i fail"
[ERROR] (ex29.c:23: errno: None) Funkcja fail_on_purpose zwróciła 1 dla\
    danych: i fail

# Próba podania nieprawidłowych argumentów.
$ ./ex29 ./libex29.so fail_on_purpose
[ERROR] (ex29.c:11: errno: Brak) UŻYCIE: e29 libex29.so funkcja dane
# Próba wywołania nieistniejącej funkcji.
$ ./ex29 ./libex29.so adfasfasdf asdfadff
[ERROR] (ex29.c:20: errno: Brak) Nie znaleziono funkcji adfasfasdf
    w bibliotece libex29.so: dlsym(0x1076009b0, adfasfasdf):\
        symbol not found

# Próba wczytania nieistniejącego pliku .so.
$ ./ex29 ./libex.so adfasfasdf asdfadfas
[ERROR] (ex29.c:17: errno: No such file or directory) Nie udało się
    otworzyć biblioteki libex.so: dlopen(libex.so, 2): image not found
$
```

Jeden problem, jaki możesz napotkać, wynika z tego, że każdy system operacyjny, każda wersja każdego systemu operacyjnego, a także każdy kompilator w każdej wersji każdego systemu operacyjnego wydaje się zmieniać sposób kompilacji biblioteki współdzielonej za każdym razem, gdy początkujący programista próbuje to zrobić. Jeżeli polecenie służące do kompilacji biblioteki *libex29.so* nie działa, poinformuj mnie o tym — dodam odpowiednie komentarze dla innych platform.

OSTRZEŻENIE Czasami wydasz polecenie, o którym sądzisz, że jest standardowe, na przykład `cc -Wall -g -DNDEBUG -ldl ex29.c -o ex29`, w nadziei na otrzymanie pliku wynikowego, a kompilacja jednak zakończy się niepowodzeniem. Na niektórych platformach z niewiadomych mi powodów kolejność podawania bibliotek ma znaczenie. Dlatego też w systemach Debian i Ubuntu musisz wydać polecenie `cc -Wall -g -DNDEBUG ex29.c -ldl -o ex29`, aby wszystko zadziałało. Tak po prostu jest. Przedstawione polecenie działa w OS X i dlatego je pozostawiłem. Jeżeli jednak w przyszłości będziesz przeprowadzał linkowanie biblioteki dynamicznej i pojawi się komunikat o niemożliwości jej znalezienia, spróbuj zmienić kolejność elementów w poleceniu i zobacz, czy to pomoże.

Irytujący jest fakt, że rzeczywiste różnice między platformami to nic więcej poza kolejnością argumentów w powłoce. Nie ma żadnego powodu, dla którego

umieszczenie elementu `-ldl` w innym miejscu na liście argumentów miałoby mieć jakiekolwiek znaczenie. Przecież to jedynie opcja i konieczność pamiętania o stosowaniu jej w określonej kolejności jest bardzo irytująca.

Jak to zepsuć?

Otwórz plik *libex29.so* w edytorze pozwalającym na obsługę plików binarnych. Zmień kilka bajtów i zamknij bibliotekę. Następnie sprawdź, czy funkcja `dlopen()` będzie w stanie wczytać bibliotekę pomimo jej uszkodzenia.

Zadania dodatkowe

- Czy zwróciłeś uwagę na błędny kod w pliku *libex29.c*? Czy zauważyłeś, że pomimo użycia pętli `for` nadal mamy sprawdzenie pod kątem znaku `'\0'`? Popraw ten błąd, aby funkcja zawsze sprawdzała wielkość ciągu tekstowego, który będzie przetwarzała.
- Wykorzystaj szkielet przygotowany w katalogu *c-skeleton* i utwórz nowy projekt dla tego ćwiczenia. Plik *libex29.c* umieść w katalogu *src*. Wprowadź odpowiednie zmiany w pliku *Makefile*, aby biblioteka została skompilowana jako *build/libex29.so*.
- Weź plik *ex29.c* i umieść go w katalogu *tests*, aby został uruchomiony jako test jednostkowy. Wprowadź wszystkie niezbędne zmiany, aby wczytywana była biblioteka *build/libex29.so*, i uruchom testy ręcznie, podobnie jak zrobiłem to nieco wcześniej w ćwiczeniu.
- Zapoznaj się z umieszczoną w podręczniku systemowym `man` dokumentacją dla funkcji `dlopen()` i ustal wszystkie powiązane z nią funkcje. Wypróbuj kilka innych opcji funkcji `dlopen()` poza `RTLD_NOW`.

ĆWICZENIE 30

Zautomatyzowane testowanie

Zautomatyzowane testowanie jest często używane w innych językach programowania, takich jak Python i Ruby, natomiast rzadko w C. Częściowo wynika to z trudności, jakie sprawia automatyczne wczytywanie i testowanie fragmentów kodu w języku C. W tym ćwiczeniu przygotujemy bardzo mały framework testowania, a następnie wykorzystamy utworzony wcześniej szkielet katalogu projektu do wykonania przykładowego testu.

Framework, który tutaj zastosujemy i umieścimy w szkielecie katalogu projektu *c-skeleton*, nosi nazwę *minunit*, a jego początek sięga niewielkiego fragmentu kod utworzonego przez Jera Design. Kod ten rozbudowałem do przedstawionej poniżej postaci.

Plik *minunit.h*:

```
#undef NDEBUG
#ifndef _minunit_h
#define _minunit_h

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

#define mu_suite_start() char *message = NULL

#define mu_assert(test, message) if (!(test)) {\
    log_err(message); return message; }
#define mu_run_test(test) debug("\n-----%s", " " #test); \
    message = test(); tests_run++; if (message) return message;

#define RUN_TESTS(name) int main(int argc, char *argv[]) {\
    argc = 1; \
    debug("----- WYKONYWANIE: %s", argv[0]);\
    printf("----\nWYKONYWANIE: %s\n", argv[0]);\
    char *result = name();\
    if (result != 0) {\
        printf("NIEPOWODZENIE: %s\n", result);\
    }\
    else {\
        printf("WSZYSTKIE TESTY ZOSTAŁY ZALICZONE\n");\
    }\
    printf("Liczba wykonanych testów: %d\n", tests_run);\
    exit(result != 0);\
}

int tests_run;

#endif
```

Z pierwotnego kodu nie zostało praktycznie nic, ponieważ używam makr pochodzących z pliku *dbg.h* oraz dużego makra `RUN_TESTS()`, które utworzyłem na końcu powyższego pliku. Pomimo niewielkiej ilości kodu otrzymujemy w pełni funkcjonalny system do przeprowadzania testów jednostkowych. System ten można wykorzystać we własnym kodzie C, ponieważ do wykonywania tekstów używa skryptu powłoki.

Przygotowanie frameworka testów jednostkowych

Aby kontynuować pracę, powinieneś mieć działający plik *libex29.c* w katalogu *src*. Ponadto musisz wykonać zadanie dodatkowe z poprzedniego ćwiczenia, polegające na utworzeniu prawidłowo działającego programu wczytującego *ex29.c*. W poprzednim ćwiczeniu poprosiłem Cię o wprowadzenie zmian pozwalających na działanie wymienionego programu jako testu jednostkowego. Jednak teraz przedstawię moje rozwiązanie i pokażę, jak osiągnąć taki sam efekt za pomocą *minunit.h*.

Pracę zaczynamy od utworzenia prostego, pustego zestawu testów jednostkowych w pliku *libex29_tests.c*, który należy umieścić w katalogu *tests*. Poniżej przedstawiłem zawartość tego pliku.

Plik *libex29_tests.c*:

```
#include "minunit.h"

char *test_dlopen()
{

    return NULL;
}

char *test_functions()
{

    return NULL;
}

char *test_failures()
{

    return NULL;
}

char *test_dlclose()
{

    return NULL;
}

char *all_tests()
```

```
28 {
29     mu_suite_start();
30
31     mu_run_test(test_dlopen);
32     mu_run_test(test_functions);
33     mu_run_test(test_failures);
34     mu_run_test(test_dlclose);
35
36     return NULL;
37 }
38
39 RUN_TESTS(all_tests);
```

W powyższym kodzie możesz zobaczyć sposób użycia makra `RUN_TESTS()` z pliku *test/minunit.h*, a także zastosowanie innych makr. Rzeczywisty kod funkcji został usunięty, aby dokładnie pokazać strukturę testu jednostkowego. Poniżej przedstawiam dokładne omówienie zawartości pliku.

libex29_tests.c:1. Polecenie dołączające framework *minunit.h*.

libex29_tests.c:3 – 7. Pierwszy test. Struktura testów nie zawiera argumentów. Wartością zwrotną jest `char *`. W przypadku *sukcesu* to będzie `NULL`. To jest niezwykle ważne, ponieważ inne makra będą przekazywały komunikaty o błędach do komponentu wykonującego testy.

libex29_tests.c:9 – 25. Tutaj mamy kolejne testy o takiej samej strukturze jak w przypadku pierwszego.

libex29_tests.c:27. Funkcja wykonująca testy i kontrolująca wszystkie pozostałe testy. Jej struktura pozostaje taka sama jak w innych zestawach testów, ale ta wersja zawiera konfigurację dodatkową.

libex29_tests.c:29. Konfiguracja dodatkowa dla testu `mu_suite_start()`.

libex29_tests.c:31. W taki sposób można określić testy do wykonania, używając makra `mu_run_test()`.

libex29_tests.c:36. Po wskazaniu testów do wykonania wartością zwrotną jest `NULL`, podobnie jak zwykłej funkcji testu.

libex29_tests.c:39. Na końcu używamy dużego makra `RUN_TESTS()` do powiązania z metodą `main()` wraz z wszystkimi testami i nakazujemy wykonanie zestawu `all_tests()`.

I to wszystko, co jest potrzebne do wykonania testów. Teraz spróbuj je wykonać w katalogu szkieletu projektu. Oto wynik podjęcia takiej próby:

Sesja dla ćwiczenia 30.:

```
Znaki niewidoczne
```

Zacząłem od wydania polecenia `make clean`, a następnie rozpocząłem kompilację, której skutkiem było utworzenie plików *libNAZWA-BIBLIOTEKI.a* i *libNAZWA-BIBLIOTEKI.so*. Pamiętaj, że te operacje przeprowadziłeś, wykonując zadania dodatkowe w poprzednim ćwiczeniu.

Jeżeli nie udało Ci się, poniżej przedstawiam plik *.diff* z różnicami względem aktualnie używanego pliku *Makefile*.

Plik *ex30.Makefile.diff*:

```
diff --git a/code/c-skeleton/Makefile b/code/c-skeleton/Makefile
index 135d538..21b92bf 100644
--- a/code/c-skeleton/Makefile
+++ b/code/c-skeleton/Makefile
@@ -9,9 +9,10 @@ TEST_SRC=$(wildcard tests/*_tests.c)
 TESTS=$(patsubst %.c,%,$(TEST_SRC))

 TARGET=build/libYOUR_LIBRARY.a
+SO_TARGET=$(patsubst %.a,%.so,$(TARGET))

 # Docelowe wersje programu.
-all: $(TARGET) tests
+all: $(TARGET) $(SO_TARGET) tests

 dev: CFLAGS=-g -Wall -Isrc -Wall -Wextra $(OPTFLAGS)
 dev: all
@@ -21,6 +22,9 @@ $(TARGET): build $(OBJECTS)
     ar rcs $@ $(OBJECTS)
     ranlib $@

+$(SO_TARGET): $(TARGET) $(OBJECTS)
+ $(CC) -shared -o $@ $(OBJECTS)
+
 build:
     @mkdir -p build
     @mkdir -p bin
```

Po wprowadzeniu powyższych zmian powinieneś być w stanie wszystko skompilować i możesz wreszcie uzupełnić pozostałe testy jednostkowe.

Plik *libex29_tests.c*:

```
 1 #include "minunit.h"
 2 #include <dlfcn.h>
 3
 4 typedef int (*lib_function) (const char *data);
 5 char *lib_file = "build/libYOUR_LIBRARY.so";
 6 void *lib = NULL;
 7
 8 int check_function(const char *func_to_run, const char *data,
 9     int expected)
10 {
11     lib_function func = dlsym(lib, func_to_run);
12     check(func != NULL,
13         "Nie znaleziono funkcji %s w bibliotece %s: %s", func_to_run,
14         lib_file, dlerror());
15
16     int rc = func(data);
```

```
17      check(rc == expected, "Funkcja %s zwróciła %d dla danych: %s",
18          func_to_run, rc, data);
19
20      return 1;
21  error:
22      return 0;
23  }
24
25  char *test_dlopen()
26  {
27      lib = dlopen(lib_file, RTLD_NOW);
28      mu_assert(lib != NULL, " Nie udało się otworzyć biblioteki do testów.");
29
30      return NULL;
31  }
32
33  char *test_functions()
34  {
35      mu_assert(check_function("print_a_message", "Witaj", 0),
36          "Wykonanie funkcji print_a_message() zakończyło się niepowodzeniem.");
37      mu_assert(check_function("uppercase", "Witaj", 0),
38          "Wykonanie funkcji uppercase() zakończyło się niepowodzeniem.");
39      mu_assert(check_function("lowercase", "Witaj", 0),
40          "Wykonanie funkcji lowercase() zakończyło się niepowodzeniem.");
41
42      return NULL;
43  }
44
45  char *test_failures()
46  {
47      mu_assert(check_function("fail_on_purpose", "Witaj", 1),
48          "Wykonanie funkcji fail_on_purpose() powinno zakończyć się
            ↪niepowodzeniem.");
49
50      return NULL;
51  }
52
53  char *test_dlclose()
54  {
55      int rc = dlclose(lib);
56      mu_assert(rc == 0, " Nie udało się zamknąć biblioteki.");
57
58      return NULL;
59  }
60
61  char *all_tests()
62  {
63      mu_suite_start();
64
65      mu_run_test(test_dlopen);
66      mu_run_test(test_functions);
67      mu_run_test(test_failures);
68      mu_run_test(test_dlclose);
```

```
69
70      return NULL;
71  }
72
73  RUN_TESTS(all_tests);
```

Mam nadzieję, że teraz już wiesz, o co w tym wszystkim chodzi, ponieważ w powyższym fragmencie kodu nie ma nic nowego poza funkcją `check_function()`. Można dostrzec często powtarzany wzorzec, polegający na kopiowaniu pewnego fragmentu kodu, a następnie prostą automatyzację przez utworzenie funkcji lub makra przeznaczonego do realizacji tego zadania. W omawianym przykładzie wykonywane są funkcje z biblioteki *.so*, którą należy wczytać, i dlatego do tego celu utworzyłem niewielką funkcję.

Zadania dodatkowe

- Przedstawione rozwiązanie działa, ale prawdopodobnie jest nieco chaotyczne. Oczyść katalog *c-skeleton* w taki sposób, aby zawierał wszystkie użyte pliki, ale usuń z nich kod powiązany z ćwiczeniem 29. Tak przygotowany katalog powinieneś móc wykorzystać w charakterze szkieletu dla każdego nowego projektu bez konieczności wprowadzania w nim zbyt wielu zmian.
- Przeanalizuj plik *runtests.sh*, a następnie poszukaj informacji o składni powłoki Bash, aby dokładnie zrozumieć sposób działania skryptu. Czy potrafisz utworzyć wersję tego skryptu w języku C?

ĆWICZENIE 31

Najczęściej spotykane niezdefiniowane zachowanie

W tym miejscu książki jest już najwyższa pora na przedstawienie najczęściej spotykanych rodzajów niezdefiniowanego zachowania, na jakie się możesz natknąć. W języku C istnieje 191 przypadków zachowania uznanych przez komitet standaryzacyjny za takie, które nie będą zdefiniowane przez standard, a tym samym ich interpretacja pozostaje otwarta. Wprawdzie część z tych przypadków zdecydowanie nie dotyczy zadań wykonywanych przez kompilator, ale większość to po prostu przykład leniwej kapitulacji komitetu, która prowadzi do irytacji programistów lub co gorsza, defektów w tworzonym oprogramowaniu. Oto jeden z przykładów lenistwa:

W trakcie tokenizacji napotkano niedopasowany znak „or" w wierszu kodu źródłowego.

W powyższym przypadku standard C99 w rzeczywistości pozwala twórcy kompilatora nie sprostać zadaniu przetwarzania danych, któremu bez problemów mógłby sprostać początkujący programista. Dlaczego tak się dzieje? Kto to może wiedzieć… Ale prawdopodobnie ktoś w komitecie standardów pracował nad kompilatorem C z przedstawioną usterką i zezwolił na uwzględnienie jej w standardzie, zamiast poprawić kompilator. Ewentualnie, jak mam w zwyczaju mówić, mamy do czynienia ze zwykłym lenistwem.

Sedno kwestii niezdefiniowanego zachowania to różnice między abstrakcyjną maszyną C zdefiniowaną w standardzie i rzeczywistymi komputerami. Standard C opisuje język C zgodnie ze ściśle zdefiniowaną abstrakcyjną maszyną. To jest absolutnie poprawny sposób projektowania języka, z wyjątkiem obszaru, na którym standard C okazał się zawodny: nie wymaga od kompilatorów implementacji tej abstrakcyjnej maszyny i wymuszenia jej specyfikacji. Twórca kompilatora może zupełnie zignorować abstrakcyjną maszynę w 191 miejscach standardu. Tak naprawdę powinna być używana nazwa „maszyna abstrakcyjna, ale", na przykład „to jest ściśle zdefiniowana maszyna abstrakcyjna, ale…".

W ten sposób komitet standardów i twórcy kompilatorów mogą jednocześnie mieć ciastko i zjeść ciastko. Możemy więc mieć standard pełen pominięć, luźną specyfikację i błędy, ale kiedy *programista* natknie się na jeden z wymienionych problemów, to komitet standardów i twórcy kompilatorów mogą po prostu wskazać maszynę abstrakcyjną i powiedzieć beznamiętnym głosem: „MASZYNA ABSTRAKCYJNA TO JEDYNE, CO MA ZNACZENIE. NIE ZAPEWNIASZ ZGODNOŚCI Z NIĄ!". Tak więc w 191 miejscach standardu twórcy kompilatorów nie muszą się dostosowywać, natomiast Ty oczywiście tak. W ten sposób stałeś się niejako obywatelem drugiej kategorii, mimo że język programowania C został tak naprawdę utworzony dla Ciebie.

Mamy więc sytuację, w której *Ty* — a nie twórca kompilatora — jesteś odpowiedzialny za wymuszenie zastosowania reguł maszyny abstrakcyjnej i kiedy nieuchronnie poniesiesz porażkę, to będzie to Twoja wina. Kompilator nie ma obowiązku oznaczania niezdefiniowanego zachowania lub rozsądnego działania, a Twoją winą jest to, że nie zapamiętałeś wszystkich 191

skomplikowanych wybojów na drodze języka C. To tworzy doskonałą sytuację dla klasycznego typu wszystkowiedzącego, który będzie w stanie zapamiętać wszystkie wspomniane 191 przypadków, a następnie wykorzysta tę wiedzę do nękania początkujących programistów C.

Hipokryzja związana z istnieniem niezdefiniowanego zachowania jest szczególnie irytująca. Jeżeli pokażesz fantastyczny kod C prawidłowo korzystający z ciągów tekstowych C, ale jednocześnie pozwalający na nadpisanie znaku kończącego ciąg tekstowy, wszystkowiedzący może powiedzieć: „To jest niezdefiniowane zachowanie. To nie jest wina języka C!". Natomiast jeśli pokażesz mu przykład niezdefiniowanego zachowania taki jak `while(x) x <<= 1`, to wszystkowiedzący może powiedzieć: „To jest niezdefiniowane zachowanie, idioto! Popraw ten cholerny kod!". W ten sposób fanatycy C mogą niezdefiniowane zachowanie wykorzystać zarówno do obrony czystości projektu C, jak i przeciwko Tobie, nazywając Cię „idiotą tworzącym nieprawidłowy kod". Znaczenie niezdefiniowanego zachowania można w niektórych przypadkach odczytać jako „możesz tutaj zignorować kwestie bezpieczeństwa, ponieważ to nie jest wina C", natomiast w innych jako „jesteś idiotą, gdy tworzysz tego rodzaju kod". Rozróżnienie między tymi dwoma przypadkami nie zostało podane w standardzie języka C.

Jak widzisz, *nie* jestem fanem długiej listy niezdefiniowanego zachowania. Przed wprowadzeniem standardu C99 musiałem zapamiętać wszystkie przypadki, ale nie zawracałem sobie już głowy zapamiętywaniem kolejnych zmian. Po prostu zdecydowałem się na zmianę podejścia i znalazłem takie, które pozwala na maksymalne unikanie niezdefiniowanego zachowania, przy jednoczesnej próbie pozostania w zgodzie ze specyfikacją maszyny abstrakcyjnej, także podczas pracy z rzeczywistymi komputerami. Jednak okazało się to prawie niemożliwe i dlatego nie tworzę już więcej nowego kodu w języku C — z powodu istnienia w nim oczywistych problemów.

OSTRZEŻENIE Poniżej podaję pochodzące od Alana Turinga techniczne wyjaśnienie, dlaczego niezdefiniowane zachowanie w C jest problemem.

1. Niezdefiniowane zachowanie w języku C ma podstawy leksykalne, semantyczne i wykonawcze.
2. Zachowanie leksykalne i semantyczne może być wykryte przez kompilator.
3. Zachowanie wykonawcze według definicji przedstawionej przez Turinga zalicza się do *problemu niezdecydowania*, a tym samym jest tak zwanym problemem NP-kompletnym.
4. To oznacza, że w celu uniknięcia niezdefiniowanego zachowania w C konieczne jest rozwiązanie jednego z najstarszych nierozwiązywalnych problemów w informatyce, co w praktyce nie pozwala na uniknięcie niezdefiniowanego zachowania w C.

Ujmując rzecz zwięźlej: „Jeżeli jedynym sposobem sprawdzenia, czy przez niezdefiniowane zachowanie zostały złamane reguły maszyny abstrakcyjnej, jest uruchomienie programu C, to nigdy nie będzie można całkowicie uniknąć niezdefiniowanego zachowania".

---

20 najczęściej spotykanych przypadków niezdefiniowanego zachowania

W tej sekcji wymienię 20 najczęściej spotykanych przypadków niezdefiniowanego zachowania i wyjaśnię, jak możesz postarać się ich unikać. Ogólnie rzecz biorąc, najlepszym sposobem na uniknięcie niezdefiniowanego zachowania jest tworzenie czystego kodu, ale mimo tego niezdefiniowane zachowanie jest czasami po prostu niemożliwe do uniknięcia. Na przykład próba zapisu już po znaku kończącym ciąg tekstowy C jest przykładem niezdefiniowanego zachowania — łatwo do tego doprowadzić zupełnie przypadkowo lub może do takiej próby dojść na skutek ataku prowadzonego z zewnątrz. Na liście przedstawiłem także przypadki związane z niezdefiniowanym zachowaniem zaliczające się do tej samej kategorii, ale występujące w innym kontekście.

Najczęściej spotykane niezdefiniowane zachowanie

1. Odniesienie się do obiektu po zakończeniu jego cyklu życiowego (6.2.4).
   - Użycie wartości wskaźnika do obiektu, którego cykl życiowy się zakończył (6.2.4).
   - Została użyta wartość obiektu wraz z automatycznym czasem trwania jej przechowywania, choć nie można jej określić (6.2.4, 6.7.8, 6.8).
2. Konwersja na postać lub z postaci typu liczby całkowitej powoduje wygenerowanie wartości spoza zakresu, który może być przedstawiony (6.3.1.4).
   - Deprecjacja jednego z typów rzeczywistych liczb zmiennoprzecinkowych na inny typ powoduje wygenerowanie wartości spoza zakresu, który może być przedstawiony (6.3.1.5).
3. Dwie deklaracje tego samego obiektu lub funkcji określają niezgodne typy (6.2.7).
4. i-wartość typu tablica zostaje skonwertowana na wskaźnik do początkowego elementu tablicy, a obiekt tablicy ma zarejestrowaną klasę magazynu danych (6.3.2.1).
   - Podjęto próbę użycia wartości wyrażenia `void` bądź też dla wyrażenia `void` została zastosowana jawna lub niejawna konwersja (z wyjątkiem `void`) (6.3.2.2).
   - Konwersja wskaźnika do liczby całkowitej powoduje wygenerowanie wartości spoza zakresu, który może być przedstawiony (6.3.2.3).
   - Konwersja między dwoma wskaźnikami powoduje wygenerowanie nieprawidłowego wyniku (6.3.2.3).
   - Użycie wskaźnika do wywołania funkcji, której typ pozostaje niezgodny ze wskazywanym typem (6.3.2.3).
   - Operand jednoargumentowego operatora * ma nieprawidłową wartość (6.5.3.2).
   - Wskaźnik został skonwertowany na postać inną niż typ liczby całkowitej bądź wskaźnika (6.5.4).

- Dodawanie lub odejmowanie wskaźnika do obiektu tablicy i typu liczby całkowitej powoduje otrzymanie wyniku nieprowadzącego do tego samego obiektu tablicy (6.5.6).
- Dodanie lub odejmowanie wskaźnika do obiektu tablicy i typu liczby całkowitej powoduje otrzymanie wyniku prowadzącego poza obiekt tablicy i używanego jako operand jednoargumentowego obliczanego operatora * (6.5.6).
- Odejmowanie wskaźników, które nie prowadzą do tego samego obiektu tablicy (6.5.6).
- Indeks tablicy jest spoza zakresu, nawet jeśli obiekt pozornie jest dostępny za pomocą danego indeksu (na przykład jak w wyrażeniu l-wartości `a[1][7]`, gdy podana jest deklaracja `int a[4][5]`) (6.5.6).
- Wynik odejmowania dwóch wskaźników nie jest możliwy do przedstawienia za pomocą obiektu typu `ptrdiff_t` (6.5.6).
- Za pomocą operatorów relacji następuje porównanie wskaźników, które nie prowadzą do tej samej agregacji lub unii (ani w ogóle do tego samego obiektu tablicy) (6.5.8).
- Podjęta została próba uzyskania dostępu lub wygenerowania wskaźnika do nieistniejącego elementu struktury elastycznej tablicy, gdy wskazywany obiekt nie zawiera elementów dla tej tablicy (6.7.2.1).
- Dwa typy wskaźników, które muszą być zgodne, nie są identyczne lub nie są wskaźnikami prowadzącymi do zgodnych typów (6.7.5.1).
- Wielkość wyrażenia w deklaracji tablicy nie jest stałą i jego obliczenie w trakcie działania programu daje w efekcie wartość niedodatnią (6.7.5.2).
- Wskaźnik przekazany do parametru tablicy funkcji biblioteki nie zawiera wartości zapewniającej, że wszystkie adresy obliczeń i dostępu do obiektu pozostaną prawidłowe (7.1.4).

5. Program próbuje zmodyfikować literał w postaci ciągu tekstowego (6.4.5).
6. Dostęp do wartości przechowywanej przez obiekt odbywa się inaczej niż przez l-wartość dozwolonego typu (6.5).
7. Podjęta została próba modyfikacji wywołania funkcji, operatora warunkowego, operatora przypisania, operatora przecinka lub uzyskania do niego dostępu po kolejnym punkcie sekwencji (6.5.2.2, 6.5.15, 6.5.16, 6.5.17).
8. Wartość drugiego operandu operatora dzielenia lub modulo wynosi zero (6.5.5).
9. Obiekt jest przypisywany do niedokładnie nakładającego się obiektu lub do dokładnie nakładającego się obiektu o niezgodnym typie (6.5.16.1).
10. Stała w procedurze inicjalizacyjnej nie jest lub nie przyjmuje jednej z następujących postaci: stała arytmetyczna, stała wskaźnika `NULL`, stała adresu lub stała adresu dla typu obiektu plus bądź minus stała wyrażenia w postaci liczby całkowitej (6.6).
    - Stała wyrażenia arytmetycznego nie ma typu arytmetycznego; ma operandy niebędące liczbami całkowitymi, liczbami zmiennoprzecinkowymi, typami

wyliczeniowymi, znakami albo wyrażeniami `sizeof()` lub zawiera rzutowanie (na zewnątrz operandów do `sizeof()`) inne niż konwersja między typami arytmetycznymi (6.6).

11. Za pomocą l-wartości typu innego niż kwalifikowana stała podjęta została próba modyfikacji obiektu zdefiniowanego za pomocą typu kwalifikowanej stałej (6.7.3).
12. Funkcja wraz z zewnętrznym połączeniem została zadeklarowana za pomocą specyfikatora funkcji osadzonej, ale nie jest zdefiniowana w tej samej jednostce translacji (6.7.4).
13. Użyta została wartość nienazwanego elementu składowego struktury lub unii (6.7.8).
14. Wykonywanie programu dotarło do nawiasu zamykającego funkcję, `}`, a wartość wywołania funkcji jest używana przez wywołujący ją komponent (6.9.1).
15. Plik o nazwie takiej samej jak jeden ze standardowych plików nagłówkowych niestanowiący części danej implementacji został umieszczony w jednym ze standardowych miejsc przeszukiwanych pod kątem dołączanych plików źródłowych (7.1.2).
16. Wartość argumentu funkcji obsługującej znak nie jest równa wartości EOF lub możliwej do przedstawienia jako typ znakowy bez znaku (ang. *unsigned char*) (7.4).
17. Wartość funkcji konwersji lub arytmetycznej liczby całkowitej nie może być przedstawiona (7.8.2.1, 7.8.2.2, 7.8.2.3, 7.8.2.4, 7.20.6.1, 7.20.6.2, 7.20.1).
18. Wartość wskaźnika do obiektu `FILE` została użyta po zamknięciu powiązanego z nim pliku (7.19.3).
    - Strumień dla funkcji `fflush()` prowadzi do strumienia danych wejściowych lub strumienia uaktualnienia, w której ostatnia operacja miała dane wejściowe (7.19.5.2).
    - Ciąg tekstowy wskazywany przez argument `mode` w wywołaniu funkcji `fopen()` niedokładnie odpowiada jednemu z określonych znaków sekwencji (7.19.5.3).
    - Operacja danych wyjściowych w strumieniu uaktualnienia jest przeprowadzana po operacji danych wejściowych bez zmiany wywołania do funkcji `fflush()` lub funkcji pozycjonowania pliku. Ewentualnie operacja danych wejściowych w strumieniu uaktualnienia jest przeprowadzana po operacji danych wyjściowych bez zmiany wywołania do funkcji `fflush()` lub funkcji pozycjonowania pliku (7.19.5.3).
19. Specyfikacja konwersji dla funkcji sformatowanych danych wyjściowych zawiera znak `#` lub `0` wraz ze specyfikatorem konwersji innym niż opisany (7.19.6.1, 7.24.2.1).
    - Specyfikator konwersji `s` zostaje napotkany przez jedną z funkcji sformatowanych danych wyjściowych, a argument nie zawiera znaku oznaczającego koniec ciągu tekstowego (o ile nie będzie podana wielkość ciągu tekstowego, ponieważ wtedy nie jest wymagany znak oznaczający koniec ciągu tekstowego) (7.19.6.1, 7.24.2.1).
    - Zawartość tablicy podanej w wywołaniu funkcji `fgets()`, `gets()` lub `fgtsw()` została użyta po wystąpieniu błędu odczytu (7.19.7.2, 7.19.7.7, 7.24.3.2).

20. Wskaźnik inny niż `NULL` zwrócony przez wywołanie funkcji `calloc()`, `malloc()` lub `realloc()` wraz z żądaną zerową wielkością został użyty w celu uzyskania dostępu do obiektu (7.20.3).

- Została użyta wartość wskaźnika odwołującego się do przestrzeni dealokowanej przez wywołanie funkcji `free()` lub `realloc()` (7.20.3).
- Argument wskaźnika do wywołania funkcji `free()` lub `realloc()` nie odpowiada wcześniej zwróconemu wskaźnikowi przez wywołanie `calloc()`, `malloc()` lub `realloc()`. Ewentualnie przestrzeń została dealokowana przez wywołanie funkcji `free()` lub `realloc()` (7.20.3.2, 7.20.3.4).

Istnieje znacznie więcej przypadków niezdefiniowanego zachowania, ale wymienione powyżej to te, z którymi spotykałem się najczęściej lub które najczęściej występują w kodzie C. Są to jednocześnie przypadki najtrudniejsze do uniknięcia. Dlatego też jeśli zapamiętasz przynajmniej wymienione powyżej, to będziesz w stanie unikać najważniejszych przypadków niezdefiniowanego zachowania w C.

ĆWICZENIE 32

Lista dwukierunkowa

Zadaniem tej książki jest pokazanie, jak naprawdę działa komputer, co obejmuje również poznanie sposobu funkcjonowania różnych struktur danych i algorytmów. Komputery same z siebie nie przeprowadzają użytecznych operacji. Aby komputer mógł robić przydatne rzeczy, potrzebuje struktury danych, a następnie organizuje przetwarzanie tych struktur. W różnych językach programowania znajdują się dołączone biblioteki implementujące wszystkie tego rodzaju struktury lub dostępna jest bezpośrednia składnia dla nich. W języku C musisz samodzielnie zaimplementować wszystkie niezbędne struktury danych, co czyni z niego doskonały język, pozwalający poznać faktyczny sposób działania struktur danych.

Moim celem jest pomóc Ci w trzech zadaniach:

- Zrozumienie, co tak naprawdę dzieje się w kodzie języka Python, Ruby lub JavaScript w następującej postaci: `data = {"name": "Zed"}`.
- Utworzenie jeszcze lepszego kodu w języku C przez użycie struktur danych w odniesieniu do doskonale znanych problemów, dla których istnieją opracowane gotowe rozwiązania.
- Poznanie podstawowego zbioru struktur danych i algorytmów, aby jeszcze dokładniej wiedzieć, co najlepiej sprawdzi się w danych sytuacjach.

Czym są struktury danych?

Nazwa *struktura danych* jest samoobjaśniająca się. To po prostu metoda organizacji danych dopasowana do określonego modelu. Być może ten model jest przeznaczony do rozpoczęcia przetwarzania danych w zupełnie nowy sposób. A może jedynie jest zorganizowany tak, aby zapewnić możliwość efektywnego przechowywania danych na dysku. W tej książce będziemy stosować prosty wzorzec, zapewniający niezawodne działanie struktur danych:

- zdefiniowanie struktury dla głównej struktury zewnętrznej;
- zdefiniowanie struktury dla treści, zwykle węzłów wraz z łączami między nimi;
- utworzenie funkcji operujących na tych dwóch strukturach.

Istnieją jeszcze inne style struktur danych w języku C, ale przedstawiony powyżej wzorzec sprawdza się doskonale i zapewnia spójność podczas tworzenia większości struktur danych.

Budowa biblioteki

W pozostałej części książki będziemy tworzyć bibliotekę, z której później będziesz mógł korzystać. Biblioteka ta będzie miała następujące elementy:

- pliki nagłówkowe (*.h*) dla każdej struktury danych,
- pliki implementacji (*.c*) dla algorytmów,

- testy jednostkowe przeznaczone do przetestowania całej funkcjonalności i zagwarantowania jej prawidłowego działania,
- dokumentacja wygenerowana automatycznie na podstawie plików nagłówkowych.

Ponieważ mamy szkielet katalogu projektu (*c-skeleton*), więc użyjemy go do utworzenia projektu *liblcthw*.

Sesja dla ćwiczenia 32.:

```
$ cp -r c-skeleton liblcthw
$ cd liblcthw/
$ ls
LICENSE Makefile README.md bin build src tests
$ vim Makefile
$ ls src/
dbg.h libex29.c libex29.o
$ mkdir src/lcthw
$ mv src/dbg.h src/lcthw
$ vim tests/minunit.h
$ rm src/libex29.* tests/libex29*
$ make clean
rm -rf build tests/libex29_tests
rm -f tests/tests.log
find . -name "*.gc*" -exec rm {} \;
rm -rf `find . -name "*.dSYM" -print`
$ ls tests/
minunit.h runtests.sh
$
```

W powyższej sesji wykonałem następujące operacje:

- Utworzyłem kopię katalogu *c-skeleton*.
- Przeprowadziłem edycję pliku *Makefile*, aby zmienić nazwę *libNAZWA-BIBLIOTEKI.a* na *liblcthw* jako nowy cel (`TARGET`).
- Utworzyłem katalog *src/lcthw* przeznaczony na kod źródłowy.
- Przeniosłem plik *src/dbg.h* do nowego katalogu.
- Przeprowadziłem edycję pliku *tests/minunit.h* w celu dodania wiersza `#include <lcthw/dbh.h>`.
- Pozbyłem się niepotrzebnych plików kodu źródłowego i testów z projektu *libex29.**.
- Uprzątnąłem wszystko pozostałe.

Teraz jesteś gotowy do rozpoczęcia pracy nad budową biblioteki. Pierwsza tworzona tutaj struktura danych to lista dwukierunkowa.

Lista dwukierunkowa

Pierwszą strukturą danych, jaką dodamy do `liblcthw`, będzie lista dwukierunkowa (ang. *doubly linked list*). To jest najprostsza struktura danych, jaką można utworzyć i jaka oferuje użyteczne właściwości dla pewnych operacji. Działanie listy dwukierunkowej opiera się na węzłach zawierających wskaźniki do następnego i poprzedniego elementu. Lista dwukierunkowa zawiera wskaźniki do obu wymienionych elementów, natomiast lista jednokierunkowa (ang. *singly linked list*) tylko do następnego elementu.

Ponieważ każdy węzeł ma wskaźniki prowadzące do poprzedniego i następnego elementu, a także dlatego, że monitorowany jest pierwszy i ostatni element listy, za pomocą listy dwukierunkowej można bardzo szybko przeprowadzić pewne operacje. Każde zadanie obejmujące wstawienie lub usunięcie elementu będzie wykonywane bardzo szybko. Ponadto tego rodzaju listy są niezwykle łatwe do implementacji dla większości programistów.

Największą wadą listy jest to, że poruszanie się po niej wymaga przetworzenia każdego wskaźnika po drodze. Oznacza to, że operacje wyszukiwania, operacje sortowania i iteracji przez elementy będą w większości wykonywane wolno. Ponadto tak naprawdę nie można przechodzić do losowo wybranych elementów listy. Jeżeli masz tablicę elementów, za pomocą odpowiedniego indeksu możesz przejść do elementu w środku tablicy. Z kolei lista używa strumienia wskaźników. Dlatego też jeśli chcesz uzyskać dostęp do dziesiątego elementu, musisz najpierw przejść przez pierwszych dziewięć.

Definicja

Jak wspomniałem na początku ćwiczenia, zaczniemy od utworzenia pliku nagłówkowego zawierającego polecenia odpowiedniej struktury C.

Plik *list.h*:

```
#ifndef lcthw_List_h
#define lcthw_List_h

#include <stdlib.h>

struct ListNode;

typedef struct ListNode {
    struct ListNode *next;
    struct ListNode *prev;
    void *value;
} ListNode;

typedef struct List {
    int count;
    ListNode *first;
    ListNode *last;
} List;
```

```
List *List_create();
void List_destroy(List * list);
void List_clear(List * list);
void List_clear_destroy(List * list);

#define List_count(A) ((A)->count)
#define List_first(A) ((A)->first != NULL ? (A)->first->value : NULL)
#define List_last(A) ((A)->last != NULL ? (A)->last->value : NULL)

void List_push(List * list, void *value);
void *List_pop(List * list);
void List_unshift(List * list, void *value);
void *List_shift(List * list);

void *List_remove(List * list, ListNode * node);

#define LIST_FOREACH(L, S, M, V) ListNode *_node = NULL;\
    ListNode *V = NULL;\

for(V = _node = L->S; _node != NULL; V = _node = _node->M)

#endif
```

Zaczynamy od utworzenia dwóch struktur dla `ListNode` i `List`, które będą zawierały wspomniane węzły. W ten sposób powstaje struktura danych przeznaczona do użycia w funkcjach i makrach, jakie zostaną później zdefiniowane. Jeżeli spojrzysz na te funkcje, to zobaczysz, że są całkiem proste. Wprawdzie do tych funkcji powrócimy przy okazji omawiania implementacji, ale mam nadzieję, że łatwo odgadniesz ich przeznaczenie.

Każdy element `ListNode` zawiera trzy komponenty wewnątrz struktury danych:

- Wartość będąca wskaźnikiem do czegoś i przechowująca to, co chcemy umieścić na liście.
- Wskaźnik `ListNode *next` prowadzący do następnej struktury `ListNode` przechowującej następny element na liście.
- Wskaźnik `ListNode *prev` prowadzący do poprzedniego elementu. To jest skomplikowane, prawda? Nazwanie poprzedniego elementu po prostu „poprzednim". Mógłbym użyć słowa „wcześniejszy" lub „tylny", ale robią tak tylko pajace.

Struktura `List` jest zaledwie kontenerem przeznaczonym do przechowywania struktur `ListNode` połączonych ze sobą w łańcuchu. Monitoruje wartości określające liczbę elementów na liście (`count`), a także pierwszy (`first`) i ostatni (`last`).

Na koniec spójrz na wiersz 37. pliku *src/lcthw/list.h*, w którym zdefiniowałem makro `LIST_FOREACH()`. Jest to często spotykane rozwiązanie w programowaniu, polegające na utworzeniu makra przeznaczonego do wygenerowania kodu iteracji, aby programiści nie mogli w tym kodzie namieszać. W przypadku struktur danych zapewnienie prawidłowego przetwarzania tego rodzaju może być trudne, więc wspomniane makro okazuje się dużą pomocą. Sposób jego użycia zobaczysz, gdy będziemy omawiać plik implementacji.

Implementacja

Na tym etapie powinieneś już znać ogólny sposób działania listy dwukierunkowej. Przypominam, że to po prostu węzły wraz ze wskaźnikami prowadzącymi do poprzedniego i następnego elementu na liście. Teraz możesz więc przystąpić do wprowadzenia kodu w pliku *src/lcthw/list.c* i poznać implementację poszczególnych operacji.

Plik *list.c*

```
#include <lcthw/list.h>
#include <lcthw/dbg.h>

List *List_create()
{
    return calloc(1, sizeof(List));
}

void List_destroy(List * list)
{
    LIST_FOREACH(list, first, next, cur) {
        if (cur->prev) {
            free(cur->prev);
        }
    }

    free(list->last);
    free(list);
}

void List_clear(List * list)
{
    LIST_FOREACH(list, first, next, cur) {
        free(cur->value);
    }
}

void List_clear_destroy(List * list)
{
    List_clear(list);
    List_destroy(list);
}

void List_push(List * list, void *value)
{
    ListNode *node = calloc(1, sizeof(ListNode));
    check_mem(node);

    node->value = value;

    if (list->last == NULL) {
        list->first = node;
        list->last = node;
```

```
44      } else {
45          list->last->next = node;
46          node->prev = list->last;
47          list->last = node;
48      }
49
50      list->count++;
51
52  error:
53      return;
54  }
55
56  void *List_pop(List * list)
57  {
58      ListNode *node = list->last;
59      return node != NULL ? List_remove(list, node) : NULL;
60  }
61
62  void List_unshift(List * list, void *value)
63  {
64      ListNode *node = calloc(1, sizeof(ListNode));
65      check_mem(node);
66
67      node->value = value;
68
69      if (list->first == NULL) {
70          list->first = node;
71          list->last = node;
72      } else {
73          node->next = list->first;
74          list->first->prev = node;
75          list->first = node;
76      }
77
78      list->count++;
79
80  error:
81      return;
82  }
83
84  void *List_shift(List * list)
85  {
86      ListNode *node = list->first;
87      return node != NULL ? List_remove(list, node) : NULL;
88  }
89
90  void *List_remove(List * list, ListNode * node)
91  {
92      void *result = NULL;
93
94      check(list->first && list->last, "Lista jest pusta.");
95      check(node, "Wartością node nie może być NULL.");
96
```

```
 97      if (node == list->first && node == list->last) {
 98          list->first = NULL;
 99          list->last = NULL;
100      } else if (node == list->first) {
101          list->first = node->next;
102          check(list->first != NULL,
103              "Nieprawidłowa lista, jakoś tak się stało, że pierwszy element
                 ↪ma wartość NULL.");
104          list->first->prev = NULL;
105      } else if (node == list->last) {
106          list->last = node->prev;
107          check(list->last != NULL,
108              "Nieprawidłowa lista, jakoś tak się stało, że następny element
                 ↪ma wartość NULL.");
109      list->last->next = NULL;
110      } else {
111          ListNode *after = node->next;
112          ListNode *before = node->prev;
113          after->prev = before;
114          before->next = after;
115      }
116
117          list->count--;
118          result = node->value;
119          free(node);
120
121  error:
122      return result;
123  }
```

Następnie implementujemy wszystkie operacje na liście dwukierunkowej, czego nie można zrobić za pomocą prostych makr. Zamiast omawiać każdy, najmniejszy wiersz kodu przedstawionego pliku, poniżej prezentuję jedynie ogólne omówienie każdej operacji zdefiniowanej w plikach *list.h* i *list.c*, a później zostawię Cię, abyś mógł przeanalizować kod.

`list.h:List_count()`. Zwraca liczbę elementów na liście. Ta wartość jest modyfikowana wraz z dodawaniem i usuwaniem elementów.

`list.h:List_first()`. Zwraca pierwszy element listy, ale nie usuwa go.

`list.h:List_last()`. Zwraca ostatni element listy, ale nie usuwa go.

`list.h:LIST_FOREACH()`. Przeprowadza iterację przez elementy listy.

`list.h:List_create()`. Po prostu tworzy główną strukturę `List`.

`list.h:List_destroy()`. Usuwa strukturę `List` wraz z wszystkimi elementami, jakie mogły się w niej znajdować.

`list.h:List_clear()`. Wygodna funkcja pozwalająca na zwolnienie *wartości* w poszczególnych węzłach, a nie same węzły.

`list.h:List_clear_destroy()`. Usuwa zawartość listy oraz samą listy. Działanie nie należy do zbyt efektywnych, ponieważ iteracja przez listę odbywa się dwukrotnie.

list.h:List_push(). To jest pierwsza operacja pokazująca zalety listy dwukierunkowej. Umieszcza nowy element na końcu listy. Ponieważ to po prostu kilka wskaźników, operacja jest niezwykle szybka.

list.h:List_pop(). Działanie odwrotne do `List_pop()` powoduje usunięcie ostatniego elementu listy i jego zwrot.

list.h:List_unshift(). Kolejna operacja łatwa do przeprowadzenia na liście dwukierunkowej, czyli bardzo szybkie dodanie elementów na *początku* listy. W omawianym przypadku nadałem funkcji nazwę `List_unshift()`, ponieważ nic lepszego nie przyszło mi na myśl.

list.h:List_shift(). Podobnie jak w przypadku `List_pop()`, ale ta funkcja powoduje usunięcie i zwrot pierwszego elementu listy.

list.h:List_remove(). To jest operacja faktycznie odpowiedzialna za usunięcie elementu w przypadku użycia `List_pop()` lub `List_shift()`. Usuwanie elementów wydaje się trudne w przypadku struktur danych i ta funkcja to potwierdza. Obsługuje ona niemałą liczbę warunków w zależności od tego, gdzie znajduje się element przeznaczony do usunięcia: na początku, na końcu, na początku i na końcu, w środku.

Większość funkcji znajdujących się w omówionym pliku niczym specjalnym się nie wyróżnia i dlatego bardzo łatwo można odgadnąć przeznaczenie kodu. Zdecydowanie powinieneś skoncentrować się na makrze `LIST_FOREACH()` użytym w funkcji `List_destroy()`, aby przekonać się, jak bardzo może ono uprościć tę często wykonywaną operację.

Testy

Po skompilowaniu kodu źródłowego listy możemy przystąpić do przygotowania testów, aby mieć pewność, że utworzone wcześniej operacje listy funkcjonują prawidłowo.

Plik *list_tests.c*:

```
 1 #include "minunit.h"
 2 #include <lcthw/list.h>
 3 #include <assert.h>
 4
 5 static List *list = NULL;
 6 char *test1 = "dane testowe 1";
 7 char *test2 = "dane testowe 2";
 8 char *test3 = "dane testowe 3";
 9
10 char *test_create()
11 {
12     list = List_create();
13     mu_assert(list != NULL, "Nie udało się utworzyć listy.");
14
15     return NULL;
16 }
17
```

```
18 char *test_destroy()
19 {
20     List_clear_destroy(list);
21
22     return NULL;
23
24 }
25
26 char *test_push_pop()
27 {
28     List_push(list, test1);
29     mu_assert(List_last(list) == test1, "Nieprawidłowa ostatnia wartość.");
30
31     List_push(list, test2);
32     mu_assert(List_last(list) == test2, "Nieprawidłowa ostatnia wartość");
33
34     List_push(list, test3);
35     mu_assert(List_last(list) == test3, "Nieprawidłowa ostatnia wartość.");
36     mu_assert(List_count(list) == 3, "Nieprawidłowa liczba elementów podczas
       ↪tworzenia nowego na końcu listy.");
37
38     char *val = List_pop(list);
39     mu_assert(val == test3, "Nieprawidłowa wartość podczas usuwania elementu
       ↪na końcu listy.");
40
41     val = List_pop(list);
42     mu_assert(val == test2, "Nieprawidłowa wartość podczas usuwania elementu
       ↪na końcu listy.");
43
44     val = List_pop(list);
45     mu_assert(val == test1, "Nieprawidłowa wartość podczas usuwania elementu
       ↪na końcu listy.");
46     mu_assert(List_count(list) == 0, "Nieprawidłowa liczba elementów
       ↪po usunięciu elementu na końcu listy.");
47
48     return NULL;
49 }
50
51 char *test_unshift()
52 {
53     List_unshift(list, test1);
54     mu_assert(List_first(list) == test1, "Nieprawidłowa pierwsza wartość.");
55
56     List_unshift(list, test2);
57     mu_assert(List_first(list) == test2, "Nieprawidłowa pierwsza wartość.");
58
59     List_unshift(list, test3P
60     mu_assert(List_first(list) == test3, "Nieprawidłowa ostatnia wartość.");
61     mu_assert(List_count(list) == 3, "Nieprawidłowa liczba elementów po dodaniu
       ↪nowego na początku listy.");
62
63     return NULL;
64 }
```

```
65
66 char *test_remove()
67 {
68     // Musimy przetestować jedynie usunięcie elementu w środku listy,
69     // ponieważ usunięcie na początku i końcu listy jest testowane w innych miejscach.
70
71     char *val = List_remove(list, list->first->next);
72     mu_assert(val == test2, "Nieprawidłowy usunięty element.");
73     mu_assert(List_count(list) == 2, "Nieprawidłowa liczba elementów
       ↪po usunięciu elementu.");
74     mu_assert(List_first(list) == test3, "Nieprawidłowy pierwszy element
       ↪po usunięciu.");
75     mu_assert(List_last(list) == test1, "Nieprawidłowy ostatni element
       ↪po usunięciu.");
76
77     return NULL;
78 }
79
80 char *test_shift()
81 {
82     mu_assert(List_count(list) != 0, "Nieprawidłowa liczba elementów przed
       ↪usunięciem pierwszego na liście.");
83
84     char *val = List_shift(list);
85     mu_assert(val == test3, "Nieprawidłowa wartość podczas usuwania pierwszego
       ↪elementu na liście.");
86
87     val = List_shift(list);
88     mu_assert(val == test1, "Nieprawidłowa wartość podczas usuwania pierwszego
       ↪elementu na liście.");
89     mu_assert(List_count(list) == 0, "Nieprawidłowa liczba elementów
       ↪po usunięciu pierwszego na liście.");
90
91     return NULL;
92 }
93
94 char *all_tests()
95 {
96     mu_suite_start();
97
98     mu_run_test(test_create);
99     mu_run_test(test_push_pop);
100    mu_run_test(test_unshift);
101    mu_run_test(test_remove);
102    mu_run_test(test_shift);
103    mu_run_test(test_destroy);
104
105    return NULL;
106 }
107
108 RUN_TESTS(all_tests);
```

Przedstawiony powyżej test po prostu wykonuje każdą operację i sprawdza, czy przebiega ona zgodnie z oczekiwaniami. Zastosowałem tutaj pewne uproszczenie i utworzyłem zaledwie jedną listę (`List *list`) dla całego programu, a następnie przeprowadziłem na niej testy. Dzięki temu uniknąłem konieczności tworzenia listy dla każdego testu. To jednak może oznaczać, że niektóre testy zostały zaliczone ze względu na sposób wykonania poprzedniego. W takim przypadku staram się, aby każdy test zachował listę bez zmian lub wykorzystywał wyniki otrzymane w poprzednim teście.

Co powinieneś zobaczyć?

Jeżeli wszystko zrobiłeś prawidłowo, po kompilacji i uruchomieniu testów jednostkowych powinieneś otrzymać następujące dane wyjściowe.

Sesja dla ćwiczenia 32.:

```
$ make
cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG -fPIC -c -o\
    src/lcthw/list.o src/lcthw/list.c
ar rcs build/liblcthw.a src/lcthw/list.o
ranlib build/liblcthw.a
cc -shared -o build/liblcthw.so src/lcthw/list.o
cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG build/liblcthw.a
    tests/list_tests.c -o tests/list_tests
sh ./tests/runtests.sh
Wykonywanie testów jednostkowych:
----
WYKONYWANIE: ./tests/list_tests
WSZYSTKIE TESTY ZOSTAŁY ZALICZONE
Liczba wykonanych testów: 6
tests/list_tests PASS
$
```

Upewnij się o wykonaniu sześciu testów, o przeprowadzeniu kompilacji bez żadnych ostrzeżeń i błędów, a także o utworzeniu plików *build/liblcthw.a* i *build/liblcthw.so*.

Jak można usprawnić kod?

Zamiast pokazywać, jak zepsuć kod, tym razem zaproponuję różne możliwości jego usprawnienia.

- Działanie funkcji `List_clear_destroy()` może być znacznie efektywniejsze dzięki wykorzystaniu makro `LIST_FOREACH()` i wykonaniu obu wywołań `free()` wewnątrz jednej pętli.
- Możesz dodać asercje dla warunków początkowych, aby program nie otrzymywał wartości `NULL` dla parametrów `List *list`.

- Możesz dodać inwarianty sprawdzające, czy zawartość listy zawsze jest prawidłowa. Na przykład wartość `count` nigdy nie jest mniejsza niż `0`, zatem możesz sprawdzić, czy wartość `count` jest większa niż `0`, a następnie czy `first` ma wartość inną niż `NULL`.
- Możesz dodać dokumentację do pliku nagłówkowego w postaci komentarzy — przed każdą strukturą, funkcją i każdym makrem — opisujących przeznaczenie danego fragmentu kodu.

Powyższe usprawnienia są przykładem omówionych wcześniej w książce praktyk programowania defensywnego, które zwiększają niezawodność kodu i poprawiają jego użyteczność. Powinieneś wprowadzić wymienione usprawnienia, a następnie poszukać jak najwięcej kolejnych, aby w maksymalnym stopniu usprawnić ten kod.

Zadania dodatkowe

- Poszukaj informacji dodatkowych o listach jedno- i dwukierunkowej, a także o tym, w jakich sytuacjach są preferowane poszczególne listy.
- Poszukaj informacji o ograniczeniach listy dwukierunkowej. Na przykład wprawdzie tego rodzaju lista jest efektywna podczas wstawiania i usuwania elementów, ale jednocześnie jest bardzo wolna w trakcie iteracji przez wszystkie elementy.
- Jakie operacje zostały pominięte w ćwiczeniu, które według Ciebie mogą być użyteczne? Wybrane przykłady to kopiowanie, łączenie i dzielenie. Zaimplementuj te operacje oraz utwórz dla nich testy jednostkowe.

ĆWICZENIE 33

Algorytmy listy dwukierunkowej

Zamierzam omówić dwa algorytmy stosowane do sortowania listy. Jednak wcześniej muszę Cię ostrzec, że jeśli potrzebujesz posortowanych danych, to nie powinieneś używać list. Niestety, listy są okropne podczas sortowania. Mamy dostępne znacznie lepsze struktury danych, jeśli jednym z wymagań jest sortowanie. Zaprezentuję tutaj dwa algorytmy sortowania, które wybrałem, ponieważ są nieco trudniejsze do użycia wraz z listami. Niejako przy okazji będziesz mógł się zastanowić, jak efektywnie operować elementami list.

Algorytmy te umieszczę w dwóch plikach — pliku nagłówkowym *list_algos.h* i pliku implementacji *list_algos.c*, a następnie utworzymy plik *list_algos_test.c* przeznaczony na testy jednostkowe dla tych algorytmów. Po prostu podążaj za przedstawioną przeze mnie strukturą, ponieważ w ten sposób zachowana będzie przejrzystość rozwiązania. Jeżeli kiedykolwiek będziesz pracować nad innymi bibliotekami, to pamiętaj, że przedstawiona tutaj struktura nie jest powszechnie stosowana.

W ćwiczeniu znajdzie się również dodatkowe wyzwanie dla Ciebie i byłoby dobrze, gdybyś spróbował nie oszukiwać podczas jego realizacji. Najpierw przedstawię *test jednostkowy*, który później wprowadzisz. Następnie poproszę Cię o zaimplementowanie obu algorytmów na podstawie ich opisów w Wikipedii, jeszcze przed tym, gdy spojrzysz na kod opracowany przeze mnie.

Sortowanie bąbelkowe i sortowanie przez scalanie

Czy wiesz, co tak wspaniałego jest w internecie? Po prostu mogę odesłać Cię do znajdujących się w Wikipedii artykułów dotyczących sortowania bąbelkowego (ang. *bubble sort*) i sortowania przez scalanie (ang. *merge sort*). W ten sposób mogę uniknąć konieczności napisania wielu zdań. Poniżej pokażę, jak można w rzeczywistości zaimplementować wymienione algorytmy sortowania, ale posłużę się przy tym pseudokodem. Oto podpowiedzi, jak mógłbyś się zabrać do implementowania tych algorytmów.

- Przeczytaj opis i zapoznaj się z ewentualnymi wizualizacjami.
- Narysuj na papierze algorytm w postaci kwadratów i linii lub weź fizyczne karty do gry (bądź karty z numerami) i spróbuj ręcznie przygotować algorytm. Dzięki temu otrzymasz konkretną demonstrację sposobu działania algorytmu.
- W pliku implementacji *list_algos.c* utwórz szkielet funkcji, wprowadź odpowiednie zmiany w pliku nagłówkowym *list_algos.h*, a następnie przygotuj szkielety testów.
- Utwórz pierwszy test kończący się niepowodzeniem, a następnie popraw kod, aby wszystko kompilowało się prawidłowo.
- Powróć na stronę artykułu w Wikipedii, skopiuj i wklej stamtąd pseudokod (nie kod w języku C!) do pierwszej tworzonej funkcji.

- Przekształć pseudokod na dobry kod w C, tak jak Cię nauczyłem dotąd w książce. Wykorzystaj przygotowany wcześniej test jednostkowy do upewnienia się, że nowy kod działa prawidłowo.
- Uzupełnij kilka dodatkowych testów sprawdzających przypadki skrajne, na przykład pusta lista, lista już posortowana itd.
- Powtórz operację dla drugiego algorytmu i przetestuj go.

Powyżej zdradziłem sekret, jak można ustalić działanie i przygotować implementację większości algorytmów, przynajmniej aż do chwili, gdy napotkasz naprawdę zakręcony algorytm. W omawianym przypadku na podstawie artykułów w Wikipedii implementujesz sortowanie bąbelkowe i sortowanie przez scalanie, co wydaje się dobre na początek.

Test jednostkowy

Poniżej przedstawiłem test jednostkowy przeznaczony dla pseudokodu.

Plik *list_algos_tests.c*:

```
#include "minunit.h"
#include <lcthw/list_algos.h>
#include <assert.h>
#include <string.h>

char *values[] = { "XXXX", "1234", "abcd", "xjvef", "NDSS" };

#define NUM_VALUES 5

List *create_words()
{
    int i = 0;
    List *words = List_create();

    for (i = 0; i < NUM_VALUES; i++) {
        List_push(words, values[i]);
    }

    return words;
}

int is_sorted(List * words)
{
    LIST_FOREACH(words, first, next, cur) {
        if (cur->next && strcmp(cur->value, cur->next->value) > 0) {
            debug("%s %s", (char *)cur->value,
                (char *)cur->next->value);
            return 0;
        }
    }

```

```
32      return 1;
33  }
34
35  char *test_bubble_sort()
36  {
37      List *words = create_words();
38
39      // Poniższy kod powinien działać na liście wymagającej posortowania.
40      int rc = List_bubble_sort(words, (List_compare) strcmp);
41      mu_assert(rc == 0, "Sortowanie bąbelkowe zakończyło się niepowodzeniem.");
42      mu_assert(is_sorted(words),
43          "Słowa nie są posortowane po przeprowadzeniu sortowania bąbelkowego.");
44
45      // Poniższy kod powinien działać na już posortowanej liście.
46      rc = List_bubble_sort(words, (List_compare) strcmp);
47      mu_assert(rc == 0, "Sortowanie bąbelkowe już posortowanej listy zakończyło
        ↪się niepowodzeniem.");
48      mu_assert(is_sorted(words),
49          "Słowa powinny być sortowane, jeśli przeprowadzono sortowanie
            ↪bąbelkowe.");
50
51      List_destroy(words);
52
53      // Poniższy kod powinien działać na pustej liście.
54      words = List_create(words);
55      rc = List_bubble_sort(words, (List_compare) strcmp);
56      mu_assert(rc == 0, "Sortowanie bąbelkowe pustej listy zakończyło się
        ↪niepowodzeniem.");
57      mu_assert(is_sorted(words), "Słowa powinny być sortowane, jeśli lista jest
        ↪pusta.");
58
59      List_destroy(words);
60
61      return NULL;
62  }
63
64  char *test_merge_sort()
65  {
66      List *words = create_words();
67
68      // Poniższy kod powinien działać na liście wymagającej posortowania.
69      List *res = List_merge_sort(words, (List_compare) strcmp);
70      mu_assert(is_sorted(res), "Słowa nie są posortowane po przeprowadzeniu
        ↪sortowania przez scalanie.");
71
72      List *res2 = List_merge_sort(res, (List_compare) strcmp);
73      mu_assert(is_sorted(res),
74          "Lista nadal powinna być posortowana po przeprowadzeniu sortowania przez
            ↪scalanie.");
75      List_destroy(res2);
76      List_destroy(res);
77
78      List_destroy(words);
```

```
79     return NULL;
80 }
81
82 char *all_tests()
83 {
84     mu_suite_start();
85
86     mu_run_test(test_bubble_sort);
87     mu_run_test(test_merge_sort);
88
89     return NULL;
90 }
91
92 RUN_TESTS(all_tests);
```

Sugeruję rozpoczęcie od sortowania bąbelkowego, a po przygotowaniu działającego algorytmu przejście do sortowania przez scalanie. Pracę zacząłbym od przygotowania prototypów i szkieletów funkcji pozwalających na kompilację wszystkich trzech plików, choć oczywiście żaden test nie będzie zaliczony. Następnie dodawałbym kolejny kod implementacji aż do otrzymania prawidłowo działającego algorytmu.

Implementacja

Czy oszukiwałeś podczas przygotowywania implementacji algorytmów sortowania? W późniejszych ćwiczeniach przedstawię test jednostkowy i powiem, co należy zaimplementować. Dlatego też dobrym ćwiczeniem dla Ciebie będzie powstrzymanie się od analizy mojego kodu aż do chwili, gdy opracujesz prawidłowo funkcjonującą własną implementację wymienionych algorytmów sortowania. Poniżej przedstawiłem zawartość przygotowanych przeze mnie plików *list_algos.h* i *list_algos.c*.

Plik *list_algos.h*:

```
#ifndef lcthw_List_algos_h
#define lcthw_List_algos_h

#include <lcthw/list.h>

typedef int (*List_compare) (const void *a, const void *b);

int List_bubble_sort(List * list, List_compare cmp);

List *List_merge_sort(List * list, List_compare cmp);

#endif
```

Plik *list_algos.c*:

```
1 #include <lcthw/list_algos.h>
2 #include <lcthw/dbg.h>
3
```

```
 4 inline void ListNode_swap(ListNode * a, ListNode * b)
 5 {
 6     void *temp = a->value;
 7     a->value = b->value;
 8     b->value = temp;
 9 }
10
11 int List_bubble_sort(List * list, List_compare cmp)
12 {
13     int sorted = 1;
14
15     if (List_count(list) <= 1) {
16         return 0; // Lista jest już posortowana.
17     }
18
19     do {
20         sorted = 1;
21         LIST_FOREACH(list, first, next, cur) {
22             if (cur->next) {
23                 if (cmp(cur->value, cur->next->value) > 0) {
24                     ListNode_swap(cur, cur->next);
25                     sorted = 0;
26                 }
27             }
28         }
29     } while (!sorted);
30
31     return 0;
32 }
33
34 inline List *List_merge(List * left, List * right, List_compare cmp)
35 {
36     List *result = List_create();
37     void *val = NULL;
38
39     while (List_count(left) > 0 || List_count(right) > 0) {
40         if (List_count(left) > 0 && List_count(right) > 0) {
41             if (cmp(List_first(left), List_first(right)) <= 0) {
42                 val = List_shift(left);
43             } else {
44                 val = List_shift(right);
45             }
46
47             List_push(result, val);
48         } else if (List_count(left) > 0) {
49             val = List_shift(left);
50             List_push(result, val);
51         } else if (List_count(right) > 0) {
52             val = List_shift(right);
53             List_push(result, val);
54         }
55     }
56
```

```
57      return result;
58  }
59
60  List *List_merge_sort(List * list, List_compare cmp)
61  {
62      if (List_count(list) <= 1) {
63          return list;
64      }
65
66      List *left = List_create();
67      List *right = List_create();
68      int middle = List_count(list) / 2;
69
70      LIST_FOREACH(list, first, next, cur) {
71          if (middle > 0) {
72              List_push(left, cur->value);
73          } else {
74              List_push(right, cur->value);
75          }
76
77          middle--;
78      }
79
80      List *sort_left = List_merge_sort(left, cmp);
81      List *sort_right = List_merge_sort(right, cmp);
82
83      if (sort_left != left)
84          List_destroy(left);
85      if (sort_right != right)
86          List_destroy(right);
87
88      return List_merge(sort_left, sort_right, cmp);
89  }
```

Sortowanie bąbelkowe nie jest trudne do zaimplementowania, choć ten algorytm działa naprawdę wolno. Natomiast sortowanie przez scalanie jest znacznie bardziej skomplikowane i szczerze mówiąc, mógłbym poświęcić nieco więcej czasu na optymalizację tego kodu, ale kosztem zmniejszenia jego przejrzystości.

Istnieje jeszcze inny sposób implementacji sortowania przez scalanie, oparty na metodzie wstępującej (ang. *bottom-up*), ale jest nieco trudniejszy do zrozumienia i dlatego nie przedstawię go w książce. Jak wcześniej wspomniałem, algorytmy sortowania listy są zupełnie bezcelowe. Możesz poświęcić cały dzień, próbując poprawić wydajność, a nadal będzie ona niższa niż w przypadku innych struktur danych obsługujących sortowanie. Po prostu nie używaj list, gdy potrzebujesz posortowanych danych.

Co powinieneś zobaczyć?

Jeżeli wszystko działa prawidłowo, powinieneś otrzymać dane wyjściowe podobne do przedstawionych poniżej.

Sesja dla ćwiczenia 33.:

```
$ make clean all
rm -rf build src/lcthw/list.o src/lcthw/list_algos.o\
    tests/list_algos_tests tests/list_tests
rm -f tests/tests.log
find . -name "*.gc*" -exec rm {} \;
rm -rf `find . -name "*.dSYM" -print`
cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG -fPIC -c -o\
    src/lcthw/list.o src/lcthw/list.c
cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG -fPIC -c -o\
    src/lcthw/list_algos.o src/lcthw/list_algos.c
ar rcs build/liblcthw.a src/lcthw/list.o src/lcthw/list_algos.o
ranlib build/liblcthw.a
cc -shared -o build/liblcthw.so src/lcthw/list.o src/lcthw/list_algos.o
cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG build/liblcthw.a\
    tests/list_algos_tests.c -o tests/list_algos_tests
cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG build/liblcthw.a\
    tests/list_tests.c -o tests/list_tests
sh ./tests/runtests.sh
Wykonywanie testów jednostkowych:
----
WYKONYWANIE: ./tests/list_algos_tests
WSZYSTKIE TESTY ZOSTAŁY ZALICZONE
Liczba wykonanych testów: 2
tests/list_algos_tests PASS
----
WYKONYWANIE: ./tests/list_tests
WSZYSTKIE TESTY ZOSTAŁY ZALICZONE
Liczba wykonanych testów: 6
tests/list_tests PASS
$
```

W kolejnych ćwiczeniach nie będę już pokazywał danych wyjściowych, o ile nie będzie to absolutnie konieczne. Od tej chwili powinieneś wiedzieć, że wykonuję testy i że wszystkie zostają zaliczone, więc cały projekt kompiluje się bez problemów.

Jak można usprawnić kod?

Powróćmy jeszcze na chwilę do opisu algorytmów — istnieje kilka sposobów na usprawnienie ich implementacji. Poniżej wymieniłem te najbardziej oczywiste.

- Sortowanie przez scalanie przeprowadza obłędnie dużo operacji kopiowania i tworzenia list, więc znajdź sposób na zmniejszenie liczby tych operacji.
- W opublikowanym w Wikipedii artykule dotyczącym sortowania bąbelkowego wspomniano o kilku optymalizacjach. Spróbuj je zaimplementować.
- Czy potrafisz użyć funkcji `List_split()` i `List_join()`, o ile oczywiście je zaimplementowałeś, do poprawienia algorytmu sortowania przez scalanie?

- Przeanalizuj wszystkie strategie programowania defensywnego i spróbuj poprawić niezawodność implementacji, zapewnić ochronę przed wskaźnikami `NULL`, a następnie utworzyć opcjonalny inwariant debugowania działający podobnie jak `is_sorted()` po sortowaniu.

Zadania dodatkowe

- Przygotuj test jednostkowy porównujący wydajność obu algorytmów. Informacje dotyczące podstawowej funkcji przeznaczonej do mierzenia czasu znajdziesz w podręczniku systemowym (`man 3 time`). Nie zapomnij o konieczności wykonania wystarczającej liczby iteracji, aby otrzymać próbki z przynajmniej kilku sekund.
- Eksperymentuj z ilością danych na listach przeznaczonych do sortowania i sprawdź, jaki ma to wpływ na czas sortowania.
- Znajdź sposób na symulację wypełnienia list o różnych, losowo wybranych wielkościach i zmierz czas potrzebny na wykonanie operacji. Otrzymane wyniki przedstaw graficznie, aby zobaczyć, jak wyglądają w porównaniu z opisem algorytmu.
- Spróbuj wyjaśnić, dlaczego sortowanie list jest kiepskim pomysłem.
- Zaimplementuj funkcję `List_insert_sorted()`, która pobiera wartość i używa `List_compare()` do wstawienia elementu w odpowiednim położeniu, aby lista zawsze pozostała posortowana. Jak użycie takiej metody wygląda w porównaniu z sortowaniem listy po jej utworzeniu?
- Spróbuj zaimplementować sortowanie przez scalanie, używając metody wstępującej, której omówienie znajdziesz w artykule w Wikipedii. Przedstawiony tam kod jest już w języku C, więc powinien być łatwy do odtworzenia. Jednak spróbuj zrozumieć sposób jego działania w porównaniu z wolniejszą, przygotowaną przeze mnie wersją.

ĆWICZENIE 34

Tablica dynamiczna

Tablica dynamiczna potrafi sama się powiększać i obsługuje większość funkcji oferowanych przez listy. Zwykle wymaga mniejszej ilości miejsca, działa szybciej oraz charakteryzuje się innymi zaletami. W tym ćwiczeniu przedstawię także kilka wad tablicy dynamicznej, takich jak np. wolno wykonywana operacja usuwania elementów na początku tablicy. Rozwiązaniem tego problemu jest po prostu usuwanie elementów na końcu tablicy.

Tablica dynamiczna to tak naprawdę tablica zawierająca wskaźniki `void **` prealokowane jednocześnie i prowadzące do danych. W przypadku listy otrzymaliśmy pełną strukturę przechowującą wskaźnik `*wartość`, natomiast w przypadku tablicy dynamicznej mamy pojedynczą tablicę wraz ze wszystkimi wskaźnikami. Oznacza to, że nie potrzebujemy żadnych innych wskaźników prowadzących do poprzedniego i następnego rekordu, ponieważ dostaniemy się do nich bezpośrednio za pomocą indeksu tablicy dynamicznej.

Na początek przedstawię plik nagłówkowy przeznaczony dla implementacji tworzonej w ćwiczeniu.

Plik *darray.h*:

```
#ifndef _DArray_h
#define _DArray_h
#include <stdlib.h>
#include <assert.h>
#include <lcthw/dbg.h>

typedef struct DArray {
    int end;
    int max;
    size_t element_size;
    size_t expand_rate;
    void **contents;
} DArray;

DArray *DArray_create(size_t element_size, size_t initial_max);

void DArray_destroy(DArray * array);

void DArray_clear(DArray * array);

int DArray_expand(DArray * array);

int DArray_contract(DArray * array);

int DArray_push(DArray * array, void *el);

void *DArray_pop(DArray * array);
```

```
void DArray_clear_destroy(DArray * array);


#define DArray_last(A) ((A)->contents[(A)->end - 1])
#define DArray_first(A) ((A)->contents[0])
#define DArray_end(A) ((A)->end)
#define DArray_count(A) DArray_end(A)
#define DArray_max(A) ((A)->max)


#define DEFAULT_EXPAND_RATE 300

static inline void DArray_set(DArray * array, int i, void *el)
{
    check(i < array->max, "Próba ustawienia wartości wykraczającej poza
    ↪tablicę.");
    if (i > array->end)
        array->end = i;
    array->contents[i] = el;
error:
    return;
}

static inline void *DArray_get(DArray * array, int i)
{
    check(i < array->max, "Próba pobrania wartości spoza tablicy.");
    return array->contents[i];
error:
    return NULL;
}

static inline void *DArray_remove(DArray * array, int i)
{
    void *el = array->contents[i];
    array->contents[i] = NULL;
    return el;
}

static inline void *DArray_new(DArray * array)
{
    check(array->element_size > 0,
        "Nie można użyć DArray_new w przypadku tablicy dynamicznej o wielkości
        ↪0.");
    return calloc(1, array->element_size);
error:
    return NULL;
}
#define DArray_free(E) free((E))

#endif
```

Powyższy plik nagłówkowy pokazuje nową technikę umieszczenia funkcji typu `static inline` bezpośrednio w pliku nagłówkowym. Tego rodzaju definicje funkcji działają podobnie jak

stosowane przez nas dotąd makra `#define`, ale są przykładem przejrzystszego i łatwiejszego do utworzenia kodu. Jeżeli musisz utworzyć blok kodu dla makra i nie potrzebujesz funkcjonalności generowania kodu, to wykorzystaj funkcję typu `static inline`.

Nową technikę porównaj z makrem `LIST_FOREACH()`, które *generowało* poprawną pętlę `for` dla listy. Takie rozwiązanie byłoby niemożliwe do osiągnięcia za pomocą funkcji typu `static inline`, ponieważ w rzeczywistości opiera się ono na wygenerowaniu wewnętrznego bloku kodu dla pętli. Jedyną możliwością byłoby użycie funkcji wywołania zwrotnego, ale wtedy otrzymujemy rozwiązanie o mniejszej wydajności i na dodatek trudniejsze w użyciu.

Kolejnym krokiem jest utworzenie testu jednostkowego na naszej tablicy dynamicznej (`DArray`).

Plik *darray_tests.c*:

```
#include "minunit.h"
#include <lcthw/darray.h>

static DArray *array = NULL;
static int *val1 = NULL;
static int *val2 = NULL;

char *test_create()
{
    array = DArray_create(sizeof(int), 100);
    mu_assert(array != NULL, "Wykonanie DArray_create() zakończyło się
    ↪niepowodzeniem.");
    mu_assert(array->contents != NULL, "Nieprawidłowa zawartość tablicy
    ↪dynamicznej.");
    mu_assert(array->end == 0, "Koniec nie znajduje się w odpowiednim
    ↪miejscu.");
    mu_assert(array->element_size == sizeof(int),
        "Nieprawidłowa wielkość elementu.");
    mu_assert(array->max == 100, "Nieprawidłowa wielkość maksymalna
    ↪na początku.");

    return NULL;
}

char *test_destroy()
{
    DArray_destroy(array);

    return NULL;
}

char *test_new()
{
    val1 = DArray_new(array);
    mu_assert(val1 != NULL, "Nie udało się utworzyć nowego elementu.");

    val2 = DArray_new(array);
    mu_assert(val2 != NULL, "Nie udało się utworzyć nowego elementu.");
```

```
35
36      return NULL;
37  }
38
39  char *test_set()
40  {
41      DArray_set(array, 0, val1);
42      DArray_set(array, 1, val2);
43
44      return NULL;
45  }
46
47  char *test_get()
48  {
49      mu_assert(DArray_get(array, 0) == val1, "Nieprawidłowa pierwsza wartość.");
50      mu_assert(DArray_get(array, 1) == val2, "Nieprawidłowa druga wartość.");
51
52      return NULL;
53  }
54
55  char *test_remove()
56  {
57      int *val_check = DArray_remove(array, 0);
58      mu_assert(val_check != NULL, "Wartość nie powinna być NULL.");
59      mu_assert(*val_check == *val1, "Powinna być pobrana pierwsza wartość.");
60      mu_assert(DArray_get(array, 0) == NULL, "Tablica powinna być usunięta.");
61      DArray_free(val_check);
62
63      val_check = DArray_remove(array, 1);
64      mu_assert(val_check != NULL, "Wartość nie powinna być NULL.");
65      mu_assert(*val_check == *val2, "Powinna być pobrana pierwsza wartość.");
66      mu_assert(DArray_get(array, 1) == NULL, "Tablica powinna być usunięta.");
67      DArray_free(val_check);
68
69      return NULL;
70  }
71
72  char *test_expand_contract()
73  {
74      int old_max = array->max;
75      DArray_expand(array);
76      mu_assert((unsigned int)array->max == old_max + array->expand_rate,
77          "Nieprawidłowa wielkość po rozszerzeniu tablicy.");
78
79      DArray_contract(array);
80      mu_assert((unsigned int)array->max == array->expand_rate + 1,
81          "Powinna pozostać na poziomie co najmniej expand_rate.");
82
83      DArray_contract(array);
84      mu_assert((unsigned int)array->max == array->expand_rate + 1,
85          "Powinna pozostać na poziomie co najmniej expand_rate.");
86
87      return NULL;
```

```
88 }
89
90 char *test_push_pop()
91 {
92     int i = 0;
93     for (i = 0; i < 1000; i++) {
94         int *val = DArray_new(array);
95         *val = i * 333;
96         DArray_push(array, val);
97     }
98
99     mu_assert(array->max == 1201, "Nieprawidłowa wielkość maksymalna.");
100
101    for (i = 999; i >= 0; i--) {
102        int *val = DArray_pop(array);
103        mu_assert(val != NULL, "Wartość nie powinna być NULL.");
104        mu_assert(*val == i * 333, "Nieprawidłowa wartość.");
105        DArray_free(val);
106    }
107
108    return NULL;
109 }
110
111 char *all_tests()
112 {
113     mu_suite_start();
114
115     mu_run_test(test_create);
116     mu_run_test(test_new);
117     mu_run_test(test_set);
118     mu_run_test(test_get);
119     mu_run_test(test_remove);
120     mu_run_test(test_expand_contract);
121     mu_run_test(test_push_pop);
122     mu_run_test(test_destroy);
123
124     return NULL;
125 }
126
127 RUN_TESTS(all_tests);
```

Kod przedstawiony w poniższym pliku pokazuje sposób użycia wszystkich operacji, co znacznie powinno ułatwić Ci późniejszą implementację `DArray`.

Plik *darray.c*:

```
1 #include <lcthw/darray.h>
2 #include <assert.h>
3
4 DArray *DArray_create(size_t element_size, size_t initial_max)
5 {
6     DArray *array = malloc(sizeof(DArray));
7     check_mem(array);
```

```
 8     array->max = initial_max;
 9     check(array->max > 0, "Wartość initial_max musi być większa niż 0.");
10
11     array->contents = calloc(initial_max, sizeof(void *));
12     check_mem(array->contents);
13
14     array->end = 0;
15     array->element_size = element_size;
16     array->expand_rate = DEFAULT_EXPAND_RATE;
17
18     return array;
19
20 error:
21     if (array)
22         free(array);
23     return NULL;
24 }
25
26 void DArray_clear(DArray * array)
27 {
28     int i = 0;
29     if (array->element_size > 0) {
30         for (i = 0; i < array->max; i++) {
31             if (array->contents[i] != NULL) {
32                 free(array->contents[i]);
33             }
34         }
35     }
36 }
37
38 static inline int DArray_resize(DArray * array, size_t newsize)
39 {
40     array->max = newsize;
41     check(array->max > 0, "Wartość newsize musi być większa niż 0.");
42
43     void *contents = realloc(
44         array->contents, array->max * sizeof(void *));
45     // Sprawdzamy wartość contents i zakładamy, że ponowna alokacja nie uszkodzi
       // pierwotnej w przypadku błędu.
46
47     check_mem(contents);
48
49     array->contents = contents;
50
51     return 0;
52 error:
53     return -1;
54 }
55
56 int DArray_expand(DArray * array)
57 {
58     size_t old_max = array->max;
59     check(DArray_resize(array, array->max + array->expand_rate) == 0,
```

```
60          "Nie udało się zwiększyć tablicy do nowego rozmiaru: %d",
61          array->max + (int)array->expand_rate);
62
63      memset(array->contents + old_max, 0, array->expand_rate + 1);
64      return 0;
65
66  error:
67      return -1;
68  }
69
70  int DArray_contract(DArray * array)
71  {
72      int new_size = array->end < (int)array->expand_rate ?
73          (int)array->expand_rate : array->end;
74
75      return DArray_resize(array, new_size + 1);
76  }
77
78  void DArray_destroy(DArray * array)
79  {
80      if parray) {
81          if (array->contents)
82              free(array->contents);
83          free(array);
84      }
85  }
86
87  void DArray_clear_destroy(DArray * array)
88  {
89      DArray_clear(array);
90      DArray_destroy(array);
91  }
92
93  int DArray_push(DArray * array, void *el)
94  {
95      array->contentsParray->end] = el;
96      array->end++;
97
98      if (DArray_end(array) >= DArray_max(array)) {
99          return DArray_expand(array);
100     } else {
101         return 0;
102     }
103 }
104
105 void *DArray_pop(DArray * array)
106 {
107     check(array->end - 1 >= 0, "Próba usunięcia elementu z pustej tablicy.");
108
109     void *el = DArray_remove(array, array->end - 1);
110     array->end--;
111
112     if (DArray_end(array) > (int)array->expand_rate
```

```
113             && DArray_end(array) % array->expand_rate) {
114         DArray_contract(array);
115     }
116
117     return el;
118 error:
119     return NULL;
120 }
```

Powyższy fragment pokazał jeszcze inny sposób radzenia sobie ze skomplikowanym kodem. Zamiast przechodzić od razu do implementacji w pliku .c, najpierw spójrz na plik nagłówkowy, a następnie zapoznaj się z testem jednostkowym. W ten sposób zyskujesz wiedzę o tym, jak poszczególne fragmenty programu współpracują ze sobą, co znacznie ułatwia zrozumienie całości.

Wady i zalety

Zastosowanie tablicy dynamicznej jest lepszym rozwiązaniem, gdy trzeba zoptymalizować wymienione poniżej operacje.

- Iteracja. Możesz użyć zwykłej pętli `for` wraz z funkcjami `DArray_count()` oraz `DArray_get()` i na tym zakończyć pracę. Nie są konieczne żadne specjalne makra, a samo rozwiązanie działa szybciej, ponieważ nie trzeba przeprowadzać iteracji przez wskaźniki.
- Indeksowanie. Za pomocą funkcji `DArray_get()` i `DArray_set()` można uzyskać dostęp do dowolnie wybranego elementu. W przypadku listy musisz przejść przez `N` elementów, aby dotrzeć do elementu `N+1`.
- Usuwanie. Strukturę i jej zawartość możesz usunąć w dwóch operacjach. Dla porównania — lista wymaga serii wywołań `free()` i przejścia przez wszystkie elementy.
- Klonowanie. Strukturę i jej zawartość można sklonować także w trakcie dwóch operacji przez po prostu jej skopiowanie wraz z przechowywanymi elementami. Z kolei lista wymaga przejścia przez wszystkie elementy i oddzielnego kopiowania każdego `ListNode` plus zawartości.
- Sortowanie. Jak się dowiedziałeś w poprzednim ćwiczeniu, lista jest kiepskim wyborem, gdy chcesz mieć posortowane dane. Natomiast tablica dynamiczna otwiera przed Tobą całą gamę doskonałych algorytmów sortowania, ponieważ możesz uzyskać dostęp do dowolnych elementów.
- Ogromne dane. Jeżeli trzeba przechowywać ogromną ilość danych, tablica dynamiczna będzie lepszym rozwiązaniem niż lista, ponieważ wymaga mniejszej ilości pamięci niż zawierające tę samą ilość danych struktury `ListNode`.

Jednak lista także ma pewne zalety, między innymi dotyczące wymienionych poniżej operacji.

- Wstawianie i usuwanie elementów na początku. Tablica dynamiczna wymaga do tego celu specjalnego traktowania, aby móc efektywnie wykonać tę operację, i zwykle wiąże się ona z kopiowaniem pewnych danych.

- Dzielenie lub złączanie. Lista może po prostu skopiować wskaźniki — i na tym koniec pracy. Z kolei tablica dynamiczna musi wykonać kopiowanie całych tablic używanych w danej operacji.
- Mniejsze dane. Jeżeli trzeba przechowywać jedynie kilka elementów, to lista prawdopodobnie wymaga mniejszej ilości miejsca niż zwykła tablica dynamiczna. Wynika to z tego, że tablica dynamiczna wcześniej musi zostać rozszerzona, aby mieć miejsce na przyszłe operacje wstawiania elementów. Natomiast lista zmienia wielkość tylko wtedy, gdy zachodzi potrzeba.

Mając to wszystko na uwadze, preferuję zastosowanie tablicy dynamicznej w większości przypadków, w których inni korzystają z listy. Z kolei użycie list ograniczam jedynie do struktur danych wymagających niewielkiej liczby węzłów dodawanych lub usuwanych na początku lub końcu listy. W dalszej części książki pokażę Ci jeszcze dwie podobne struktury danych: stos (`Stack`) i kolejkę (`Queue`).

Jak można usprawnić kod?

Jak zwykle przeanalizuj kod wszystkich funkcji i zastosuj strategie programowania defensywnego, warunki początkowe, inwarianty i wszystko inne, co może pomóc jeszcze bardziej poprawić niezawodność kodu.

Zadania dodatkowe

- Usprawnij testy jednostkowe w taki sposób, aby pokrywały więcej operacji, a następnie przetestuj je za pomocą pętli `for` i upewnij się o ich prawidłowym działaniu.
- Poszukaj informacji o tym, jak za pomocą tablicy dynamicznej zaimplementować sortowanie bąbelkowe i sortowanie przez scalanie, ale jeszcze tego nie rób. Implementacją algorytmów tablicy dynamicznej zajmiemy się w następnym ćwiczeniu i wtedy będziesz miał okazję się wykazać.
- Utwórz testy wydajności dla najczęściej wykonywanych operacji i porównaj je z tymi samymi operacjami wykonywanymi za pomocą list. Tego rodzaju zadanie już wykonywałeś, ale tym razem utwórz test jednostkowy ciągle powtarzający daną operację. Następnie w komponencie odpowiedzialnym za wykonywanie testów przeprowadź pomiar czasu testów.
- Spójrz na implementację `DArray_expand()`, w której zastosowano stały wzrost wartości (wielkość+300). Zmiana wielkości tablicy dynamicznej jest zwykle implementowana z użyciem mnożenia (wielkość×2). Przekonałem się jednak, że takie podejście powoduje niepotrzebnie duże zużycie pamięci, tak naprawdę nie przynosząc w zamian żadnej wyraźnej poprawy wydajności działania. Sprawdź moje założenie i przekonaj się, czy wolisz wzrost oparty na mnożeniu, czy jednak stały wzrost wielkości.

ĆWICZENIE 35

Sortowanie i wyszukiwanie

W tym ćwiczeniu przedstawię cztery algorytmy sortowania i jeden wyszukiwania. Omówione będą następujące algorytmy sortowania: sortowanie szybkie (ang. *quick sort*), sortowanie przez kopcowanie (ang. *heap sort*), sortowanie przez scalanie (ang. *merge sort*), sortowanie pozycyjne (ang. *radix sort*). Następnie pokażę, jak można przeprowadzić wyszukiwanie binarne po zakończeniu sortowania pozycyjnego.

Jednak zaliczam się do leniwych osób; w większości standardowych bibliotek C można znaleźć implementacje algorytmów sortowania przez kopcowanie, sortowania szybkiego i sortowania przez scalanie. Oto sposób, w jaki można użyć wymienionych algorytmów.

Plik *darray_algos.c*:

```
#include <lcthw/darray_algos.h>
#include <stdlib.h>

int DArray_qsort(DArray * array, DArray_compare cmp)
{
    qsort(array->contents, DArray_count(array), sizeof(void *), cmp);
    return 0;
}

int DArray_heapsort(DArray * array, DArray_compare cmp)
{
    return heapsort(array->contents, DArray_count(array),
        sizeof(void *), cmp);
}

int DArray_mergesort(DArray * array, DArray_compare cmp)
{
    return mergesort(array->contents, DArray_count(array),
        sizeof(void *), cmp);
}
```

Powyżej przedstawiłem całą implementację pliku *darray_algos.c*; powinna działać w większości nowoczesnych systemów UNIX. Działanie każdego z wymienionych algorytmów polega na sortowaniu zawartości magazynu `contents` wskaźników `void` za pomocą funkcji `DArray_compare()`. Poniżej możesz zobaczyć zawartość pliku nagłówkowego dla *darray_algos.c*.

Plik *darray_algos.h*:

```
#ifndef darray_algos_h
#define darray_algos_h

#include <lcthw/darray.h>

typedef int (*DArray_compare) (const void *a, const void *b);
```

```
int DArray_qsort(DArray * array, DArray_compare cmp);

int DArray_heapsort(DArray * array, DArray_compare cmp);

int DArray_mergesort(DArray * array, DArray_compare cmp);

#endif
```

Wielkość plików nagłówkowego i implementacji jest mniej więcej taka sama. Przechodzimy teraz do wykorzystania przedstawionych powyżej funkcji w testach jednostkowych.

Plik *darray_algos_tests.c*:

```
#include "minunit.h"
#include <lcthw/darray_algos.h>

int testcmp(char **a, char **b)
{
    return strcmp(*a, *b);
}

DArray *create_words()
{
    DArray *result = DArray_create(0, 5);
    char *words[] = P "asdfasfd",
        "werwar", "13234", "asdfasfd", "oioj" };
    int i = 0;

    for (i = 0; i < 5; i++) {
        DArray_push(result, words[i]);
    }

    return result;
}

int is_sorted(DArray * array)
{
    int i = 0;

    for (i = 0; i < DArray_count(array) - 1; i++) {
        if (strcmp(DArray_get(array, i), DArray_get(array, i + 1)) > 0) {
            return 0;
        }
    }

    return 1;
}

char *run_sort_test(int (*func) (DArray *, DArray_compare),
    const char *name)
{
    DArray *words = create_words();
    mu_assert(!is_sorted(words), "Na początku słowa nie powinny być posortowane.");
```

```
41
42      debug("--- Testowanie algorytmu sortowania %s", name);
43      int rc = func(words, (DArray_compare) testcmp);
44      mu_assert(rc == 0, "Sortowanie zakończyło się niepowodzeniem.");
45      mu_assert(is_sorted(words), "Nie posortowano.");
46
47      DArray_destroy(words);
48
49      return NULL;
50  }
51
52  char *test_qsort()
53  {
54      return run_sort_test(DArray_qsort, "qsort");
55  }
56
57  char *test_heapsort()
58  {
59      return run_sort_test(DArray_heapsort, "heapsort");
60  }
61
62  char *test_mergesort()
63  {
64      return run_sort_test(DArray_mergesort, "mergesort");
65  }
66
67  char *all_tests()
68  {
69      mu_suite_start();
70
71      mu_run_test(test_qsort);
72      mu_run_test(test_heapsort);
73      mu_run_test(test_mergesort);
74
75      return NULL;
76  }
77
78  RUN_TESTS(all_tests);
```

Warto zwrócić uwagę na jedną rzecz, która nie dawała mi spokoju przez cały dzień: definicja `testcmp` w wierszu 4. Konieczne jest użycie `char **`, a *nie* `char *`, ponieważ wartością zwrotną `qsort()` jest wskaźnik do wskaźników znajdujących się w tablicy `contents`. Funkcja `qsort()` i pozostałe funkcje sortowania skanują tablicę i przekazują funkcji porównującej *wskaźniki* do poszczególnych elementów w tablicy. Ponieważ tablica `contents` zawiera wskaźniki, więc otrzymujemy wskaźnik do wskaźnika.

Tak oto zaimplementowaliśmy trzy trudne algorytmy sortowania w zaledwie 20 wierszach kodu. Wprawdzie można by na tym zakończyć, ale skoro to ćwiczenie zostało poświęcone poznawaniu działania algorytmów sortowania, to w sekcji „Zadania dodatkowe" będziesz je wszystkie usprawniał.

Sortowanie pozycyjne i wyszukiwanie binarne

Ponieważ algorytmy sortowania szybkiego, sortowania przez kopcowanie i przez scalanie będziesz implementował samodzielnie, w tym miejscu zaprezentuję inny algorytm — sortowania pozycyjnego. Jego użyteczność podczas sortowania tablic liczb całkowitych jest nieco ograniczona, ale wydaje się, że działa w sposób magiczny. Na potrzeby przykładu utworzę specjalną strukturę danych o nazwie `RadixMap`, która będzie wykorzystywana do mapowania jednej liczby całkowitej na inną.

Poniżej znajdziesz plik nagłówkowy dla nowego algorytmu. Plik ten zawiera zarówno algorytm, jak i strukturę danych.

Plik *radixmap.h*:

```
#ifndef _radixmap_h
#include <stdint.h>

typedef union RMElement {
    uint64_t raw;
    struct {
        uint32_t key;
        uint32_t value;
    } data;
} RMElement;

typedef struct RadixMap {
    size_t max;
    size_t end;
    uint32_t counter;
    RMElement *contents;
    RMElement *temp;
} RadixMap;

RadixMap *RadixMap_create(size_t max);

void RadixMap_destroy(RadixMap * map);

void RadixMap_sort(RadixMap * map);

RMElement *RadixMap_find(RadixMap * map, uint32_t key);

int RadixMap_add(RadixMap * map, uint32_t key, uint32_t value);

int RadixMap_delete(RadixMap * map, RMElement * el);

#endif
```

Jak możesz zobaczyć, w pliku znalazła się większość takich samych operacji, jakie mieliśmy w strukturach tablicy dynamicznej i listy. Jednak różnica polega na tym, że powyższe działają jedynie z 32-bitowymi liczbami całkowitymi typu `uin32_t` i stałej wielkości. Przy okazji omawiania algorytmu sortowania pozycyjnego wprowadzę także kolejną koncepcję w języku C, czyli unie.

Unie w języku C

Unia to sposób odwołania się do tego samego fragmentu pamięci na wiele różnych sposobów. Unię definiujesz podobnie jak strukturę, przy czym każdy element współdzieli tę samą przestrzeń z wszystkimi pozostałymi. Unię możesz porównać do obrazu w pamięci, a elementy unii do różnokolorowych szkieł, przez które oglądany jest ten obraz.

Celem zastosowania unii jest zaoszczędzenie pamięci lub konwersja fragmentów pamięci między formatami. W pierwszym przypadku najczęściej są wykorzystywane warianty typów — tworzysz strukturę wraz z tagami dla poszczególnych typów, a następnie wewnątrz unię dla każdego typu. Natomiast w drugim przypadku, konwersji między formatami pamięci, po prostu definiujesz dwie struktury, a następnie uzyskujesz dostęp do odpowiedniej.

Na początek pokażę, jak przygotować wariant typu za pomocą unii w C.

Plik *ex35.c*:

```
#include <stdio.h>

typedef enum {
    TYPE_INT,
    TYPE_FLOAT,
    TYPE_STRING,
} VariantType;

struct Variant {
    VariantType type;
    union {
        int as_integer;
        float as_float;
        char *as_string;
    } data;
};

typedef struct Variant Variant;

void Variant_print(Variant * var)
{
    switch (var->type) {
        case TYPE_INT:
            printf("TYP INT: %d\n", var->data.as_integer);
            break;
        case TYPE_FLOAT:
            printf("TYP FLOAT: %f\n", var->data.as_float);
            break;
        case TYPE_STRING:
            printf("TYP STRING: %s\n", var->data.as_string);
            break;
        default:
            printf("NIEZNANY TYP: %d", var->type);
    }
}
```

```
36
37 int main(int argc, char *argv[])
38 {
39     Variant a_int = {.type = TYPE_INT, .data.as_integer = 100 };
40     Variant a_float = {.type = TYPE_FLOAT, .data.as_float = 100.34 };
41     Variant a_string = {.type = TYPE_STRING,
42         .data.as_string = "YO DUDE!" };
43
44     Variant_print(&a_int);
45     Variant_print(&a_float);
46     Variant_print(&a_string);
47
48     // Oto sposób uzyskania dostępu do typów.
49     a_int.data.as_integer = 200;
50     a_float.data.as_float = 2.345;
51     a_string.data.as_string = "Witaj!";
52
53     Variant_print(&a_int);
54     Variant_print(&a_float);
55     Variant_print(&a_string);
56
57     return 0;
58 }
```

Tego rodzaju implementację znajdziesz w wielu językach dynamicznych. Język definiuje pewien wariant typu bazowego wraz z tagami dla wszystkich bazowych typów języka, a następnie zwykle jest dostępny tag obiektu ogólnego dla typów, które można utworzyć. Zaletą przedstawionego rozwiązania jest to, że wariant wymaga jedynie takiej ilości miejsca, jakiej potrzebuje znacznik `VariantType type` i największy element składowy unii. Wynika to z faktu umieszczania przez C każdego elementu unii `Variant.data` razem, więc mogą się nakładać. Dlatego też język C ustala wielkość unii w taki sposób, aby zmieścić największy element.

W przedstawionym wcześniej pliku *radixmap.c* zdefiniowałem unię `RMElement`, która pokazuje zastosowanie unii do konwersji bloków pamięci między różnymi typami. W omawianym przykładzie przechowujemy liczbę całkowitą o wielkości `uint64_t` na potrzeby sortowania, natomiast dwie liczby całkowite typu `uint32_t` służą do przedstawienia danych w postaci par klucz-wartość. Dzięki użyciu unii zapewniam na dwa różne i niezbędne mi sposoby elegancki dostęp do tego samego bloku pamięci.

Implementacja

Kolejnym krokiem jest przygotowanie rzeczywistej implementacji `RadixMap` dla każdej zdefiniowanej operacji.

Plik *radixmap.c*:

```
1 /*
2 * Poniższy kod został oparty na kodzie Andre Reinald dość mocno zmodyfikowanym
  * przez Zeda A. Shawa.
3 */
4
```

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <lcthw/radixmap.h>
#include <lcthw/dbg.h>

RadixMap *RadixMap_create(size_t max)
{
    RadixMap *map = calloc(sizeof(RadixMap), 1);
    check_mem(map);

    map->contents = calloc(sizeof(RMElement), max + 1);
    check_mem(map->contents);

    map->temp = calloc(sizeof(RMElement), max + 1);
    check_mem(map->temp);

    map->max = max;
    map->end = 0;

    return map;
error:
    return NULL;
}

void RadixMap_destroy(RadixMap * map)
{
    if (map) {
        free(map->contents);
        free(map->temp);
        free(map);
    }
}

#define ByteOf(x,y) (((uint8_t *)x)[(y)])

static inline void radix_sort(short offset, uint64_t max,
uint64_t * source, uint64_t * dest)
{
    uint64_t count[256] = { 0 };
    uint64_t *cp = NULL;
    uint64_t *sp = NULL;
    uint64_t *end = NULL;
    uint64_t s = 0;
    uint64_t c = 0;

    // Zliczenie wystąpień każdej wartości bajta.
    for (sp = source, end = source + max; sp < end; sp++) {
        count[ByteOf(sp, offset)]++;
    }

    // Przekształcenie liczby wystąpień na indeks przez zsumowanie
    // elementów i umieszczenie ich w tej samej tablicy.
```

```
    for (s = 0, cp = count, end = count + 256; cp < end; cp++) {
        c = *cp;
        *cp = s;
        s += c;
    }

    // Wypełnienie dest odpowiednimi wartościami w odpowiednich miejscach.
    for (sp = source, end = source + max; sp < end; sp++) {
        cp = count + ByteOf(sp, offset);
        dest[*cp] = *sp;
        ++(*cp);
    }
}

void RadixMap_sort(RadixMap * map)
{
    uint64_t *source = &map->contents[0].raw;
    uint64_t *temp = &map->temp[0].raw;

    radix_sort(0, map->end, source, temp);
    radix_sort(1, map->end, temp, source);
    radix_sort(2, map->end, source, temp);
    radix_sort(3, map->end, temp, source);
}

RMElement *RadixMap_find(RadixMap * map, uint32_t to_find)
{
    int low = 0;
    int high = map->end - 1;
    RMElement *data = map->contents;

    while (low <= high) {
        int middle = low + (high - low) / 2;
        uint32_t key = data[middle].data.key;

        if (to_find < key) {
            high = middle - 1;
        } else if (to_find > key) {
            low = middle + 1;
        } else {
            return &data[middle];
        }
    }

    return NULL;
}

int RadixMap_add(RadixMap * map, uint32_t key, uint32_t value)
{
    check(key < UINT32_MAX, "Wartość klucza nie może być równa UINT32_MAX.");

    RMElement element = {.data = {.key = key,.value = value} };
    check(map->end + 1 < map->max, "Struktura RadixMap jest pełna.");
```

```
111
112     map->contents[map->end++] = element;
113
114     RadixMap_sort(map);
115
116     return 0;
117
118 error:
119     return -1;
120 }
121
122 int RadixMap_delete(RadixMap * map, RMElement * el)
123 {
124     check(map->end > 0, "Nie ma niczego do usunięcia.");
125     check(el != NULL, "Nie można usunąć elementu NULL.");
126
127     el->data.key = UINT32_MAX;
128
129     if (map->end > 1) {
130         // Nie zawracamy sobie głowy przywróceniem mapy o wielkości jednego elementu.
131         RadixMap_sort(map);
132     }
133
134     map->end--;
135
136     return 0;
137 error:
138     return -1;
139 }
```

Jak zwykle, należy wprowadzić powyższy kod, a następnie upewnić się o jego prawidłowym działaniu za pomocą przedstawionych poniżej testów jednostkowych, w których wyjaśnię, co tak naprawdę dzieje się w kodzie. Zwróć *szczególną uwagę* na funkcję `radix_sort()`, ze względu na charakterystyczny sposób jej implementacji.

Plik *radixmap_tests.c*:

```
1 #include "minunit.h"
2 #include <lcthw/radixmap.h>
3 #include <time.h>
4
5 static int make_random(RadixMap * map)
6 {
7     size_t i = 0;
8
9     for (i = 0; i < map->max - 1; i++) {
10         uint32_t key = (uint32_t) (rand() | (rand() << 16));
11         check(RadixMap_add(map, key, i) == 0, "Nie udało się dodać klucza %u.",
12             key);
13     }
14
15     return i;
16
```

```
17 error:
18     return 0;
19 }
20
21 static int check_order(RadixMap * map)
22 {
23     RMElement d1, d2;
24     unsigned int i = 0;
25
26     // Sygnalizuj błędy tylko wtedy, gdy wystąpią (nie powinno ich być).
27     for (i = 0; map->end > 0 && i < map->end - 1; i++) {
28         d1 = map->contents[i];
29         d2 = map->contents[i + 1];
30
31         if (d1.data.key > d2.data.key) {
32             debug("NIEPOWODZENIE:i=%u, klucz: %u, wartość: %u, osiągnęły
               ↪wartość maksymalną? %d\n", i,
33                 d1.data.key, d1.data.value,
34                 d2.data.key == UINT32_MAX);
35             return 0;
36         }
37     }
38
39     return 1;
40 }
41
42 static int test_search(RadixMap * map)
43 {
44     unsigned i = 0;
45     RMElement *d = NULL;
46     RMElement *found = NULL;
47
48     for (i = map->end / 2; i < map->end; i++) {
49         d = &map->contents[i];
50         found = RadixMap_find(map, d->data.key);
51         check(found != NULL, "Nie znaleziono %u w %u.", d->data.key, i);
52         check(found->data.key == d->data.key,
53             "Otrzymano nieprawidłowy wynik: %p:%u, wyszukując %u w %u", found,
54             found->data.key, d->data.key, i);
55     }
56
57     return 1;
58 error:
59     return 0;
60 }
61
62 // Sprawdzenie działania w przypadku bardzo dużej liczby elementów.
63 static char *test_operations()
64 {
65     size_t N = 200;
66
67     RadixMap *map = RadixMap_create(N);
68     mu_assert(map != NULL, "Nie udało się utworzyć mapy.");
```

```
69      mu_assert(make_random(map), "Nie udało się utworzyć losowo wybranej,
        ↪nieprawdziwej mapy radix.");
70
71      RadixMap_sort(map);
72      mu_assert(check_order(map),
73          "Nie udało się prawidłowo posortować RadixMap.");
74
75      mu_assert(test_search(map), "Test wyszukiwania zakończył się
        ↪niepowodzeniem.");
76      mu_assert(check_order(map),
77          "Mapa RadixMap nie jest posortowana po operacji wyszukiwania.");
78
79      while (map->end > 0) {
80          RMElement *el = RadixMap_find(map,
81              map->contents[map->end / 2].data.key);
82          mu_assert(el != NULL, "Powinien być otrzymany wynik.");
83
84          size_t old_end = map->end;
85
86          mu_assert(RadixMap_delete(map, el) == 0, "Nie usunięto.");
87          mu_assert(old_end - 1 == map->end, "Nieprawidłowa wielkość
            ↪po usunięciu.");
88
89          // Sprawdzenie, czy na końcu znajduje się teraz stara wartość,
90          // natomiast wartość wynosi UINT32_MAX, więc element jest usuwany.
91          mu_assert(check_order(map),
92              "Mapa RadixMap nie jest posortowana po operacji usunięcia."P
93      }
94
95      RadixMap_destroy(map);
96
97      return NULL;
98  }
99
100 char *all_testsP
101 {
102     mu_suite_start();
103     srand(time(NULL));
104
105     mu_run_test(test_operations);
106
107     return NULL;
108 }
109
110 RUN_TESTS(all_tests);
```

Nie muszę chyba wyjaśniać zbyt wiele w odniesieniu do przedstawionych powyżej testów jednostkowych. Ich działanie polega po prostu na umieszczaniu w `RadixMap` losowo wybranych liczb całkowitych, a następnie sprawdzeniu, czy można je niezawodnie pobrać. Tutaj naprawdę nie ma nic szczególnie interesującego.

Większość operacji zaimplementowanych w pliku *radixmap.c* jest łatwa do zrozumienia po spojrzeniu na ich kod. Poniżej przedstawiłem przeznaczenie podstawowych funkcji wraz z ogólnym opisem ich działania.

`RadixMap_create()`. Jak zwykle zaczynamy od alokacji całej pamięci niezbędnej dla struktury zdefiniowanej w pliku nagłówkowych *radixmap.h*. Elementy `temp` i `contents` będą użyte później, w funkcji `radix_sort()`.

`RadixMap_destroy()`. Ta funkcja służy po prostu do usunięcia utworzonej wcześniej struktury.

`radix_sort()`. Oto najważniejsza funkcja tej struktury danych; jej objaśnienie znajdziesz w dalszej części ćwiczenia.

`RadixMap_sort()`. Ta funkcja wykorzystuje `radix_sort()` do rzeczywistego sortowania elementu `contents`. Odbywa się to przez sortowanie między `contents` i `temp` aż do ostatecznego posortowania elementu `contents`. Dokładny sposób działania tej funkcji poznasz w dalszej części ćwiczenia, gdy będzie omawiana funkcja `radix_sort()`.

`RadixMap_find()`. Ta funkcja wykorzystuje algorytm wyszukiwania binarnego do odszukania podanego klucza. Dokładne wyjaśnienie sposobu jej działania przedstawię w dalszej części ćwiczenia.

`RadixMap_add()`. Ta funkcja używa `RadixMap_sort()`, umieszcza na końcu struktury podany klucz i wartość, a następnie sortuje strukturę, aby wszystkie elementy znajdowały się na właściwym miejscu. Po zakończeniu sortowania funkcja `RadixMap_find()` będzie działała prawidłowo, ponieważ mamy do czynienia z wyszukiwaniem binarnym.

`RadixMap_delete()`. Ta funkcja działa podobnie jak `RadixMap_add()`, z wyjątkiem tego, że usuwa elementy struktury przez przypisanie im wartości maksymalnej dla 32-bitowej liczby całkowitej bez znaku, czyli `UINT32_MAX`. Oznacza to, że wspomnianej wartości nie można użyć jako wartości klucza, choć jednocześnie niezwykle ułatwia usuwanie elementów. Wystarczy przypisać elementowi tę wartość, następnie przeprowadzić sortowanie, a element zostanie przeniesiony na koniec. Teraz jest już usunięty.

Przeanalizuj kod wymienionych powyżej funkcji. Do dokładniejszego wyjaśnienia pozostały nam jeszcze funkcje `RadixMap_sort()`, `radix_sort()` i `RadixMap_find()`.

Funkcja RadixMap_find() i wyszukiwanie binarne

Na początek pokażę, w jaki sposób zostało zaimplementowane wyszukiwanie binarne. Wspomniane wyszukiwanie binarne to prosty algorytm, który będzie intuicyjny dla większości osób. Tak naprawdę możesz posłużyć się talią kart i ręcznie zastosować tego rodzaju wyszukiwanie. Oto przedstawiony krok po kroku opis sposobu działania funkcji `RadixMap_find()` oraz przeprowadzania wyszukiwania binarnego.

- Określ poziom minimalny i maksymalny na podstawie wielkości tablicy.
- Pobierz środkowy element między zdefiniowanymi poziomami minimalnym i maksymalnym.
- Jeżeli klucz jest „mniejszy niż", to musi znajdować się poniżej elementu środkowego. Wartość maksymalną ustaw jako o jeden mniejszą niż element środkowy.
- Jeżeli klucz jest „większy niż", to musi znajdować się powyżej elementu środkowego. Wartość minimalną ustaw jako o jeden większą niż element środkowy.
- Jeżeli klucz jest równy elementowi środkowemu, znalazłeś go. Koniec operacji.
- Kontynuuj operację, aż minimum i maksimum nałożą się na siebie. Nie znajdziesz klucza, jeśli opuścisz pętlę.

Tak naprawdę polega to na odgadywaniu, gdzie znajduje się klucz. Odbywa się to przez pobranie środkowego elementu i porównanie go do wartości minimum i maksimum. Ponieważ dane są posortowane, więc wiadomo, że klucz musi znajdować się powyżej lub poniżej sprawdzanego elementu. Jeżeli jest poniżej, o połowę zmniejszyłeś przeszukiwaną przestrzeń. Operację kontynuujesz aż do znalezienia elementu lub nałożenia się wartości minimum i maksimum, co oznacza wyczerpanie przeszukiwanej przestrzeni.

RadixMap_sort() i radix_sort()

Sortowanie pozycyjne jest łatwe do zrozumienia, jeśli najpierw spróbujesz je przeprowadzić ręcznie. W algorytmie wykorzystywany jest fakt przechowywania liczb w sekwencji cyfr od najmniej do najbardziej znaczącej. Następnie liczby są umieszczane w kubełkach (ang. *bucket*) i pogrupowane według cyfr, a po przetworzeniu wszystkich cyfr liczby są już posortowane. Na początku tego rodzaju operacja wydaje się magiczna i szczerze mówiąc, analiza kodu wydaje się to potwierdzać, więc spróbuj tę operację przeprowadzić ręcznie.

W tym celu zacznij od zapisania w losowej kolejności serii trzycyfrowych liczb. Przyjmujemy założenie, że mamy następujący zbiór liczb: 223, 912, 275, 100, 633, 120 i 380.

- Zacznij od pogrupowania liczb w kubełkach według cyfr oznaczających jednostki: [380, 100, 120], [912], [633, 223], [275].
- Teraz trzeba przejść przez wszystkie kubełki po kolei i ułożyć liczby w kolejności według cyfr oznaczających dziesiątki: [100], [912], [120, 223], [633], [275], [380].
- W tym momencie każdy kubełek zawiera liczby posortowane według cyfr oznaczających jednostki, a później dziesiątki. Konieczne jest więc ponowne przejście przez wszystkie kubełki po kolei i ułożenie liczb względem cyfr oznaczających setki: [100, 120], [223, 275], [380], [633], [912].
- Na tym etapie wszystkie kubełki mają liczby posortowane według cyfr oznaczających setki, dziesiątki i jednostki. Po pobraniu kolejnych grup liczb otrzymasz ostatecznie posortowaną listę: 100, 120, 223, 275, 380, 633, 912.

Wykonaj kilkakrotnie powyższą operację, aby dokładnie poznać jej przebieg. To naprawdę jest sprytny, niewielki algorytm. Co najważniejsze, działa na liczbach o dowolnej wielkości, więc można sortować nawet bardzo duże liczby, ponieważ jednorazowo uwzględniany jest tylko jeden bajt.

W omawianym przykładzie cyfry (nazywane także wartościami) są poszczególnymi 8-bitowymi bajtami, a więc potrzeba 256 kubełków do przechowywania rozłożenia liczb według ich cyfr. Potrzebujemy również jakoś przechowywać liczby, ale w sposób niewymagający zbyt dużej ilości miejsca. Jeżeli popatrzysz na kod funkcji `radix_sort()`, zauważysz, że pierwszym zadaniem jest utworzenie histogramu `count` przeznaczonego do monitorowania liczby wystąpień poszczególnych cyfr dla danej wartości przesunięcia (`offset`).

Gdy poznamy liczby wystąpień dla każdej cyfry (w sumie 256), można je wykorzystać jako punkty rozkładu w tablicy docelowej. Na przykład jeśli mamy 10 bajtów o wartości `0x00`, to wiadomo, że można je umieścić w pierwszych dziesięciu slotach tablicy docelowej. W ten sposób otrzymujemy indeks pokazujący miejsce w tablicy docelowej, który jest używany w drugiej pętli `for` w funkcji `radix_sort()`.

Skoro znane jest już położenie cyfr w tablicy docelowej, wystarczy po prostu przejść przez wszystkie cyfry w tablicy źródłowej dla danego przesunięcia (`offset`) i umieścić je kolejno w przeznaczonych dla nich slotach. Makro `ByteOf()` pomaga w zachowaniu przejrzystości kodu źródłowego, ponieważ do prawidłowego działania rozwiązania stosujemy nieco sztuczek związanych ze wskaźnikiem. Jednak efektem jest umieszczenie wszystkich liczb całkowitych w kubełkach dla ich cyfr po zakończeniu działania ostatniej pętli `for`.

Interesujący jest tutaj sposób użycia rozwiązania w funkcji `RadixMap_sort()` w celu sortowania tych 64-bitowych liczb całkowitych na podstawie pierwszych 32 bitów. Czy pamiętasz, że mamy klucz i wartość w unii dla elementu `RMElement`? Oznacza to, że w celu posortowania tablicy względem kluczy konieczne jest posortowanie jedynie pierwszych 4 bajtów (32 bity podzielone przez 8 bitów na bajt) każdej liczby całkowitej.

Jeżeli spojrzysz na kod funkcji `RadixMap_sort()`, to zobaczysz, że pobieramy wskaźnik do elementów `contents` i `temp` dla tablic źródłowej i docelowej, a następnie czterokrotnie wywołujemy funkcję `radix_sort()`. W trakcie każdego wywołania zamieniamy miejscami tablicę źródłową i docelową, a później przechodzimy do kolejnego kroku. Po zakończeniu operacji funkcja `radix_sort()` kończy swoje zadanie, a ostateczna kopia danych jest posortowana w tablicy `contents`.

Jak można usprawnić kod?

Przedstawiona w ćwiczeniu implementacja ma poważną wadę, polegającą na czterokrotnym przetwarzaniu całej tablicy w trakcie każdej operacji wstawienia danych. Wprawdzie przetwarzanie jest przeprowadzane naprawdę szybko, ale znacznie lepiej będzie, jeśli uda się ograniczyć sortowanie do tego, co powinno być posortowane.

Mamy dwa sposoby, na jakie można zaimplementować wymienione usprawnienie.

- Użycie wyszukiwania binarnego do odnalezienia minimalnego położenia dla nowego elementu, a następnie przeprowadzenie sortowania od tego miejsca aż do końca. Odszukujesz minimum, wstawiasz nowy element na końcu, a następnie sortujesz elementy jedynie od minimum. W większości przypadków takie rozwiązanie pozwala na znaczne zmniejszenie sortowanej liczby elementów.

- Monitorowanie największego klucza z aktualnie pozostających w użyciu, a następnie sortowanie jedynie takiej ilości cyfr, aby wystarczyły na obsługę wspomnianego klucza. Istnieje również możliwość monitorowania najmniejszej liczby, a następnie sortowania jedynie cyfr niezbędnych dla tego zakresu. W tym celu musisz zacząć uwzględniać kolejność liczb całkowitych dla procesora (szeregowanie).

Spróbuj wprowadzić wymienione optymalizacje, ale dopiero po zmodyfikowaniu testu jednostkowego przez dodanie pewnych informacji o czasie wykonywania operacji. W ten sposób będziesz mógł sprawdzić, czy rzeczywiście poprawiłeś wydajność działania implementacji.

Zadania dodatkowe

- Zaimplementuj algorytmy sortowania szybkiego, sortowania przez kopcowanie i przez scalanie, a następnie dodaj polecenia `#define` pozwalające na wybór rodzaju sortowania lub utwórz drugi zestaw funkcji przeznaczonych do wywoływania. Przedstawione przeze mnie techniki wykorzystaj do zapoznania się z poświęconymi poszczególnym algorytmom sortowania artykułami w Wikipedii, a następnie zaimplementuj te algorytmy za pomocą pseudokodu.
- Porównaj wydajność implementacji pierwotnej z zawierającą wprowadzone optymalizacje.
- Przygotowane funkcje sortowania wykorzystaj do opracowania funkcji `DArray_sort_add()`, która dodaje elementy do tablicy dynamicznej, a następnie sortuje ją.
- Utwórz funkcję `DArray_find()` wykorzystującą algorytm wyszukiwania binarnego z `RadixMap_find()` i `DArray_compare()` do wyszukiwania elementów w posortowanej tablicy dynamicznej.

ĆWICZENIE 36

Bezpieczniejsze ciągi tekstowe

Począwszy od tego ćwiczenia, zaczniesz używać struktury `bstring`, dowiesz się, dlaczego stosowanie ciągów tekstowych C to niewiarygodnie kiepski pomysł, a następnie zmodyfikujesz kod biblioteki `liblcthw`, przystosowując go do pracy ze strukturą `bstring`.

Dlaczego stosowanie ciągów tekstowych C to niewiarygodnie kiepski pomysł?

Kiedy programiści rozmawiają o problemach związanych z językiem C, to koncepcja ciągu tekstowego jest przedstawiana jako jedna z największych wad języka. Dotąd dość intensywnie korzystaliśmy z ciągów tekstowych, nawet wspomniałem o ich niedociągnięciach, ale nie podałem powodów, dla których stanowią one tak poważną wadę. Spróbuję teraz wszystko wyjaśnić. Po dekadach używania ciągów tekstowych C zebrałem wystarczająco dużo informacji, aby z pełnym przekonaniem stwierdzić, że ich stosowanie to po prostu zły pomysł.

Nie ma możliwości potwierdzenia, że dowolny ciąg tekstowy C jest poprawny:

- Ciąg tekstowy C jest nieprawidłowy, jeśli nie jest zakończony znakiem `'\0'`.
- Każda pętla przetwarzająca nieprawidłowy ciąg tekstowy C będzie działała w nieskończoność (lub po prostu doprowadzi do przepełnienia bufora).
- Długość ciągu tekstowego C pozostaje nieznana, więc jedynym sposobem na sprawdzenie, czy został zakończony prawidłowo, jest wykonanie pętli.
- Dlatego też nie ma możliwości sprawdzenia poprawności ciągu tekstowego C bez ryzyka utworzenia pętli działającej w nieskończoność.

To jest bardzo prosta logika. Nie można utworzyć pętli sprawdzającej poprawność ciągu tekstowego C, ponieważ jeśli ciąg tekstowy okaże się nieprawidłowy, spowoduje działanie pętli w nieskończoność. Dlatego też jedynym rozwiązaniem jest *podanie jego wielkości*. Gdy wielkość ciągu tekstowego jest znana, można uniknąć niebezpieczeństwa powstania pętli działającej w nieskończoność. Gdy spojrzysz na dwie funkcje pochodzące z ćwiczenia 27., zobaczysz przedstawiony poniżej kod.

Plik *ex36.c*:

```
1 void copy(char to[], char from[])
2 {
3     int i = 0;
4
5     // Działanie pętli while nie zakończy się, jeśli na końcu zabraknie znaku '\0'.
6     while ((to[i] = from[i]) != '\0') {
7         ++i;
8     }
```

```
 9 }
10
11 int safercopy(int from_len, char *from, int to_len, char *to)
12 {
13     int i = 0;
14     int max = from_len > to_len - 1 ? to_len - 1 : from_len;
15
16     // Wielkość to_len musi wynosić przynajmniej 1 bajt.
17     if (from_len < 0 || to_len <= 0)
18         return -1;
19
20     for (i = 0; i < max; i++) {
21         to[i] = from[i];
22     }
23
24     to[to_len - 1] = '\0';
25
26     return i;
27 }
```

Wyobraź sobie, że do funkcji `copy()` chcesz dodać operację sprawdzającą, czy podany ciąg tekstowy jest prawidłowy. W jaki sposób można to zrobić? Tworzysz pętlę sprawdzającą, czy ciąg tekstowy kończy się znakiem `'\0'`. Moment, jeśli ciąg tekstowy nie kończy się wymienionym znakiem, to pętla będzie działała w nieskończoność, prawda? Szach-mat.

Niezależnie od tego, jakie działania podejmiesz, nie możesz sprawdzić, czy ciąg tekstowy C jest prawidłowy, jeśli nie znasz jego wielkości. W przypadku funkcji `safercopy()` wspomniana wielkość jest przekazywana wraz z ciągiem tekstowym. Dlatego też funkcja nie powoduje niebezpieczeństwa powstania pętli działającej w nieskończoność, ponieważ zawsze będzie można ją zakończyć. Nawet jeśli zostanie podana niepoprawna wielkość ciągu tekstowego.

Zadanie biblioteki Better String Library polega na utworzeniu struktury zawsze dołączającej do ciągu tekstowego informacje o jego wielkości. Ponieważ wielkość zawsze jest znana podczas użycia struktury `bstring`, wszystkie operacje opierające się na tej strukturze będą bezpieczniejsze. Działanie pętli będzie zakończone, zawartość może być zweryfikowana, a więc unikamy największej wady ciągu tekstowego C. Wspomniana biblioteka jest dostarczana wraz z dużą liczbą operacji możliwych do przeprowadzania na ciągach tekstowych, takich jak podział, formatowanie i wyszukiwania. Te operacje będą wykonywane w prawidłowy i bezpieczniejszy sposób.

Oczywiście struktura `bstring` także może mieć błędy, ale ponieważ istnieje już od dłuższego czasu, więc prawdopodobnie są one minimalne. Nadal są znajdowane błędy w bibliotece `glibc`, więc co programista może na to poradzić, prawda?

Użycie bstrlib

Dostępnych jest kilka bibliotek usprawnionych ciągów tekstowych, ale szczególnie polubiłem `bstrlib`, ponieważ mieści się w pojedynczym pliku i oferuje większość funkcjonalności potrzebnej do pracy z ciągami tekstowymi. W tym ćwiczeniu będą nam potrzebne dwa pliki, *bstrlib.c* i *bstrlib.h*, pochodzące z biblioteki Better String Library.

Oto zapis operacji, jakie przeprowadziłem w katalogu projektu *liblcthw*.

Sesja dla ćwiczenia 36.:

```
$ mkdir bstrlib
$ cd bstrlib/
$ unzip ~/Downloads/bstrlib-05122010.zip
Archive: /Users/zedshaw/Downloads/bstrlib-05122010.zip
...
$ ls
bsafe.c bstraux.c bstrlib.h
bstrwrap.h license.txt test.cpp
bsafe.h bstraux.h bstrlib.txt
cpptest.cpp porting.txt testaux.c
bstest.c bstrlib.c bstrwrap.cpp
gpl.txt security.txt
$ mv bstrlib.h bstrlib.c ../src/lcthw/
$ cd ../
$ rm -rf bstrlib
# Wprowadzenie zmian przedstawionych nieco niżej.
$ vim src/lcthw/bstrlib.c
$ make clean all
...
$
```

Powyżej umieściłem komentarz o konieczności wprowadzenia zmian. Wynikają one z przeniesienia pliku *bstrlib.c* do nowego miejsca oraz usunięcia błędu w systemie OS X. Oto plik z zapisem różnic.

Plik *ex36.diff*:

```
25c25
< #include "bstrlib.h"
---
> #include <lcthw/bstrlib.h>
2759c2759
< #ifdef __GNUC__
---
> #if defined(__GNUC__) && !defined(__APPLE__)
```

Jak możesz zobaczyć, dodałem polecenie `#include <lcthw/bstrlib.h>` oraz zmodyfikowałem jedno z poleceń `#ifdef` w wierszu 2759.

Poznajemy bibliotekę

To ćwiczenie jest krótkie i ma jedynie na celu przygotować Cię do użycia biblioteki Better String Library w pozostałych ćwiczeniach. W kolejnych dwóch ćwiczeniach wykorzystamy *bstrlib.c* do utworzenia struktury danych `hashmap`.

Powinieneś poznać bibliotekę przez przeanalizowanie jej plików nagłówkowego i implementacji, a następnie opracowanie testów jednostkowych (*tests/bstr_tests.c*) sprawdzających działanie wymienionych poniżej funkcji.

- `bfromcstr()`. Utworzenie struktury `bstring` na podstawie stałej w stylu C.
- `blk2bstr()`. Podobnie jak wyżej, ale z podaniem wielkości bufora.
- `bstrcpy()`. Kopiowanie struktury `bstring`.
- `bassign()`. Przypisanie struktury `bstring` do innej struktury `bstring`.
- `bassigncstr()`. Przypisanie strukturze `bstring` zawartości ciągu tekstowego C.
- `bassignblk()`. Przypisanie strukturze `bstring` zawartości ciągu tekstowego C, ale z podaniem wielkości bufora.
- `bdestroy()`. Usunięcie struktury `bstring`.
- `bconcat()`. Dołączenie jednej struktury `bstring` do innej struktury `bstring`.
- `bstricmp()`. Porównanie dwóch struktur `bstring`, wartość zwrotna jest taka sama jak w przypadku funkcji `strcmp()`.
- `biseq()`. Sprawdzenie, czy dwie struktury `bstring` są takie same.
- `binstr()`. Sprawdzenie, czy jedna struktura `bstring` znajduje się w innej.
- `bfindreplace()`. Wyszukanie jednej struktury `bstring` w drugiej, a następnie zastąpienie jej trzecią.
- `bsplit()`. Podział `bstring` na `bstrList`.
- `bformat()`. Przeprowadzenie formatowania ciągu tekstowego, co jest niezwykle użyteczne.
- `blength()`. Pobranie wielkości struktury `bstring`.
- `bdata()`. Pobranie danych ze struktury `bstring`.
- `bchar()`. Pobranie znaku ze struktury `bstring`.

Opracowane przez Ciebie testy jednostkowe powinny sprawdzać wszystkie wymienione powyżej operacje oraz kilka innych, które uznasz za szczególnie interesujące po analizie pliku nagłówkowego.

ĆWICZENIE 37

Struktura Hashmap

Struktura `Hashmap` (nazywana czasem tablicą mieszającą, tablicą haszującą lub słownikiem) jest często używana w programowaniu dynamicznym do przechowywania danych w postaci klucz-wartość. Działanie struktury polega na obliczeniu wartości hash klucza w celu wygenerowania liczby całkowitej, która następnie jest używana do odszukania kubełka (ang. *bucket*) bądź też pobrania lub ustawienia wartości. To niezwykle szybka i praktyczna struktura danych, ponieważ działa z niemalże dowolnym rodzajem danych, a ponadto jest łatwa do zaimplementowania.

Oto przykład użycia struktury `Hashmap` (inaczej słownika) w Pythonie:

```
fruit_weights = {'Jabłka': 10, 'Pomarańcze': 100, 'Winogrono': 1.0}

for key, value in fruit_weights.items():
    print key, "=", value
```

Praktycznie każdy nowoczesny język programowania oferuje podobną strukturę, a liczni programiści tworzą oparty na niej kod, choć tak naprawdę nie znają faktycznej zasady jej działania. Dzięki utworzeniu struktury danych `Hashmap` w C będę mógł zademonstrować sposób jej działania. Pracę rozpoczynamy od pliku nagłówkowego, aby można było przejść do omówienia struktury danych.

Plik *hashmap.h*:

```
#ifndef _lcthw_Hashmap_h
#define _lcthw_Hashmap_h

#include <stdint.h>
#include <lcthw/darray.h>

#define DEFAULT_NUMBER_OF_BUCKETS 100

typedef int (*Hashmap_compare) (void *a, void *b);
typedef uint32_t(*Hashmap_hash) (void *key);

typedef struct Hashmap {
    DArray *buckets;
    Hashmap_compare compare;
    Hashmap_hash hash;
} Hashmap;

typedef struct HashmapNode {
    void *key;
    void *data;
    uint32_t hash;
} HashmapNode;
```

```
typedef int (*Hashmap_traverse_cb) (HashmapNode * node);
Hashmap *Hashmap_create(Hashmap_compare compare, Hashmap_hash);
void Hashmap_destroy(Hashmap * map);

int Hashmap_set(Hashmap * map, void *key, void *data);
void *Hashmap_get(Hashmap * map, void *key);

int Hashmap_traverse(Hashmap * map, Hashmap_traverse_cb traverse_cb);

void *Hashmap_delete(Hashmap * map, void *key);

#endif
```

Struktura danych składa się z elementu `Hashmap` zawierającego dowolną liczbę struktur `HashmapNode`. Przyglądając się elementowi `Hashmap`, można dostrzec, że jego struktura przedstawia się następująco:

`DArray *buckets`. Tablica dynamiczna o zdefiniowanej na stałe wielkości 100 kubełków. Każdy kubełek będzie z kolei zawierał tablicę `DArray` przechowującą pary `HashmapNode`.

`Hashmap_compare compare`. To funkcja porównania używana przez `Hashmap` do wyszukiwania elementów na podstawie ich kluczy. Powinna działać podobnie jak inne funkcje porównania; domyślnie używa `bstrcmp()`, więc klucze są po prostu strukturami `bstring`.

`Hashmap_hash hash`. Funkcja generująca wartość hash. Odpowiada za pobranie klucza, przetworzenie jego zawartości i wygenerowanie pojedynczej wartości indeksu typu `uin32_t`. Wkrótce będziesz miał okazję bliżej poznać tę funkcję.

Przedstawione powyżej funkcje dostarczają informacji o sposobie przechowywania danych, ale jeszcze nie utworzyliśmy żadnych. Pamiętaj, że mamy tutaj do czynienia z mapowaniem na dwóch poziomach:

- Dostępne jest 100 kubełków tworzących poziom pierwszy. Elementy znajdują się w tych kubełkach na podstawie wygenerowanych dla nich wartości hash.
- Każdy kubełek to tablica dynamiczna (`DArray`) zawierająca struktury `HashmapNode`, które są po prostu wstawiane na końcu tablicy podczas operacji ich dodawania.

Struktura `HashmapNode` składa się z trzech wymienionych poniżej elementów:

`void *key`. To jest klucz dla danej pary `klucz=wartość`.

`void *value`. To jest wartość.

`uint32_t hash`. To jest obliczona wartość hash, dzięki której wyszukanie danego węzła będzie szybsze. Wystarczy sprawdzić wartość hash i przejść dalej, jeśli nie będzie dopasowana. Klucz można sprawdzić po dopasowaniu wartości hash.

Pozostała część pliku nagłówkowego nie zawiera nic nowego, więc możemy od razu przejść do pliku implementacji *hashmap.c*.

Plik *hashmap.c*:

```
#undef NDEBUG
#include <stdint.h>
#include <lcthw/hashmap.h>
#include <lcthw/dbg.h>
#include <lcthw/bstrlib.h>

static int default_compare(void *a, void *b)
{
    return bstrcmp((bstring) a, (bstring) b);
}

/**
* Prosty algorytm Boba Jenkinsa przeznaczony do generowania wartości hash.
* Algorytm pochodzi z artykułu w Wikipedii.
*/
static uint32_t default_hash(void *a)
{
    size_t len = blength((bstring) a);
    char *key = bdata((bstring) a);
    uint32_t hash = 0;
    uint32_t i = 0;

    for (hash = i = 0; i < len; ++i) {
        hash += key[i];
        hash += (hash << 10);
        hash ^= (hash >> 6);
    }

    hash += (hash << 3);
    hash ^= (hash >> 11);
    hash += (hash << 15);

    return hash;
}

Hashmap *Hashmap_create(Hashmap_compare compare, Hashmap_hash hash)
{
    Hashmap *map = calloc(1, sizeof(Hashmap));
    check_mem(map);

    map->compare = compare == NULL ? default_compare : compare;
    map->hash = hash == NULL ? default_hash : hash;
    map->buckets = DArray_create(
        sizeof(DArray *), DEFAULT_NUMBER_OF_BUCKETS);
    map->buckets->end = map->buckets->max; // Symulujemy rozszerzenie tablicy.
    check_mem(map->buckets);

    return map;

error:
```

```
51      if (map) {
52          Hashmap_destroy(map);
53      }
54
55      return NULL;
56  }
57
58  void Hashmap_destroy(Hashmap * map)
59  {
60      int i = 0;
61      int j = 0;
62
63      if (map) {
64          if (map->buckets) {
65              for (i = 0; i < DArray_count(map->buckets); i++) {
66                  DArray *bucket = DArray_get(map->buckets, i);
67                  if (bucket) {
68                      for (j = 0; j < DArray_count(bucket); j++) {
69                          free(DArray_get(bucket, j));
70                      }
71                      DArray_destroy(bucket);
72                  }
73              }
74              DArray_destroy(map->buckets);
75          }
76
77          free(map);
78      }
79  }
80
81  static inline HashmapNode *Hashmap_node_create(int hash, void *key,
82          void *data)
83  {
84      HashmapNode *node = calloc(1, sizeof(HashmapNode));
85      check_mem(node);
86
87      node->key = key;
88      node->data = data;
89      node->hash = hash;
90
91      return node;
92
93  error:
94      return NULL;
95  }
96
97  static inline DArray *Hashmap_find_bucket(Hashmap * map, void *key,
98          int create,
99          uint32_t * hash_out)
100 {
101     uint32_t hash = map->hash(key);
102     int bucket_n = hash % DEFAULT_NUMBER_OF_BUCKETS;
103     check(bucket_n >= 0, "Znaleziono nieprawidłowy kubełek: %d", bucket_n);
```

```
104     // Przechowujemy w celu zwrotu, aby komponent wywołujący mógł użyć tych danych.
105     *hash_out = hash;
106
107     DArray *bucket = DArray_get(map->buckets, bucket_n);
108
109     if (!bucket && create) {
110         // Nowy kubełek, trzeba przeprowadzić konfigurację.
111         bucket = DArray_create(
112             sizeof(void *), DEFAULT_NUMBER_OF_BUCKETS);
113         check_mem(bucket);
114         DArray_set(map->buckets, bucket_n, bucket);
115     }
116
117     return bucket;
118
119 error:
120     return NULL;
121 }
122
123 int Hashmap_set(Hashmap * map, void *key, void *data)
124 {
125     uint32_t hash = 0;
126     DArray *bucket = Hashmap_find_bucket(map, key, 1, &hash);
127     check(bucket, "Błąd, nie można utworzyć kubełka.");
128
129     HashmapNode *node = Hashmap_node_create(hash, key, data);
130     check_mem(node);
131
132     DArray_push(bucket, node);
133
134     return 0;
135
136 error:
137     return -1;
138 }
139
140 static inline int Hashmap_get_node(Hashmap * map, uint32_t hash,
141         DArray * bucket, void *key)
142 {
143     int i = 0;
144
145     for (i = 0; i < DArray_end(bucket); i++) {
146         debug("SPRÓBUJ: %d", i);
147         HashmapNode *node = DArray_get(bucket, i);
148         if (node->hash == hash && map->compare(node->key, key) == 0) {
149             return i;
150         }
151     }
152
153     return -1;
154 }
155
156 void *Hashmap_get(Hashmap * map, void *key)
```

```
157 {
158     uint32_t hash = 0;
159     DArray *bucket = Hashmap_find_bucket(map, key, 0, &hash);
160     if (!bucket) return NULL;
161
162     int i = Hashmap_get_node(map, hash, bucket, key);
163     if (i == -1) return NULL;
164
165     HashmapNode *node = DArray_get(bucket, i);
166     check(node != NULL,
167         "Nie udało się pobrać węzła z kubełka, choć powinien istnieć.");
168
169     return node->data;
170
171 error: // Celowe przejście.
172     return NULL;
173 }
174
175 int Hashmap_traverse(Hashmap * map, Hashmap_traverse_cb traverse_cb)
176 {
177     int i = 0;
178     int j = 0;
179     int rc = 0;
180
181     for (i = 0; i < DArray_count(map->buckets); i++) {
182         DArray *bucket = DArray_get(map->buckets, i);
183         if (bucket) {
184             for (j = 0; j < DArray_count(bucket); j++) {
185                 HashmapNode *node = DArray_get(bucket, j);
186                 rc = traverse_cb(node);
187                 if (rc != 0)
188                     return rc;
189             }
190         }
191     }
192
193     return 0;
194 }
195
196 void *Hashmap_delete(Hashmap * map, void *key)
197 {
198     uint32_t hash = 0;
199     DArray *bucket = Hashmap_find_bucket(map, key, 0, &hash);
200     if (!bucket)
201         return NULL;
202
203     int i = Hashmap_get_node(map, hash, bucket, key);
204     if (i == -1)
205         return NULL;
206
207     HashmapNode *node = DArray_get(bucket, i);
208     void *data = node->data;
209     free(node);
```

```
210
211     HashmapNode *ending = DArray_pop(bucket);
212
213     if (ending != node) {
214         // W porządku, wygląda na to, że to nie jest ostatni węzeł, więc dokonujemy
            // zamiany.
215         DArray_set(bucket, i, ending);
216     }
217
218     return data;
219 }
```

Nie ma nic bardzo skomplikowanego w powyższej implementacji, ale funkcje `default_hash()` i `Hashmap_find_bucket()` wymagają pewnego wyjaśnienia. Kiedy używasz funkcji `Hashmap_create()`, możesz przekazać dowolne funkcje pozwalające na porównywanie i tworzenie wartości hash. Jeżeli jednak tego nie zrobisz, to zostaną użyte `default_compare()` i `default_hash()`.

Przede wszystkim zwróć uwagę na to, jak funkcja `default_hash()` wykonuje swoje zadanie. Jest to prosta funkcja tworząca wartość hash Jenkinsa — nazwa pochodzi od twórcy tego algorytmu, Boba Jenkinsa. Algorytm tej funkcji znalazłem w artykule w Wikipedii, zatytułowanym „Jenkins hash function". Działanie funkcji polega na przejściu przez wszystkie bajty klucza w celu utworzenia wartości hash (struktura `bstring`), a następnie pracy z bitami, aby wynikiem była pojedyncza wartość typu `uint32_t`. W tym celu są wykonywane pewne operacje, między innymi `XOR`.

Istnieje wiele różnych funkcji generujących wartości hash, oferujące różne właściwości. Po wybraniu dowolnej z nich musisz ustalić sposób jej działania i wyszukiwania prawidłowych kubełków. Poniżej przedstawiam sposób działania funkcji `Hashmap_find_bucket()`:

- Na początku mamy wywołanie `map->hash(key)` w celu pobrania wartości hash dla klucza.
- Następnie przeprowadzane jest wyszukiwanie kubełka za pomocą `% DEFAULT_NUMBER_OF_BUCKETS`, a więc każda wartość hash zawsze znajdzie pewien kubełek, niezależnie od jego wielkości.
- Kolejnym krokiem jest pobranie kubełka, który również jest tablicą dynamiczną. Jeżeli kubełek nie będzie znaleziony, nastąpi jego utworzenie. To jednak już zależy od ustawień zmiennych dotyczących tworzenia kubełka.
- Po znalezieniu kubełka `DArray` dla odpowiedniej wartości hash następuje jego zwrot, a zmienna `hash_out` jest używana w celu przekazania komponentowi wywołującemu znalezionej wartości hash.

Wszystkie pozostałe funkcje wykorzystują później `Hashmap_find_bucket()` do wykonywania własnych zadań:

- Ustawienie pary klucz-wartość wymaga odszukania kubełka, utworzenia struktury `HashmapNode`, a następnie jej dodania do kubełka.
- Pobranie klucza wymaga wyszukania kubełka, a następnie odszukania struktury `HashmapNode` dopasowanej do wartości hash i klucza.

- Usunięcie elementu wymaga wyszukania kubełka, odszukania struktury `Hashmap ↪Node`, a następnie jej usunięcia przez umieszczenie w jego miejsce ostatniego węzła.

Jedyną inną funkcją, którą powinieneś dokładnie przeanalizować, jest `Hashmap_traverse()`. Jej działanie polega na przejściu przez wszystkie kubełki — w przypadku każdego kubełka zawierającego możliwe wartości wywoływana jest dla każdej z nich funkcja `traverse_cb()`. W ten sposób przeprowadzane jest skanowanie całej struktury `Hashmap` pod kątem jej wartości.

Testy jednostkowe

Przechodzimy teraz do testów jednostkowych dla wszystkich zaimplementowanych operacji.

Plik *hashmap_tests.c*:

```
#include "minunit.h"
#include <lcthw/hashmap.h>
#include <assert.h>
#include <lcthw/bstrlib.h>

Hashmap *map = NULL;
static int traverse_called = 0;
struct tagbstring test1 = bsStatic("dane testowe 1");
struct tagbstring test2 = bsStatic("dane testowe 2");
struct tagbstring test3 = bsStatic("dane testowe 3");
struct tagbstring expect1 = bsStatic("WARTOŚĆ 1");
struct tagbstring expect2 = bsStatic("WARTOŚĆ 2");
struct tagbstring expect3 = bsStatic("WARTOŚĆ 3");

static int traverse_good_cb(HashmapNode * node)
{
    debug("KLUCZ: %s", bdata((bstring) node->key));
    traverse_called++;
    return 0;
}

static int traverse_fail_cb(HashmapNode * node)
{
    debug("KLUCZ: %s", bdata((bstring) node->key));
    traverse_called++;

    if (traverse_called == 2) {
        return 1;
    } else {
        return 0;
    }
}

char *test_create()
{
```

```
36      map = Hashmap_create(NULL, NULL);
37      mu_assert(map != NULL, "Nie udało się utworzyć mapy.");
38
39      return NULL;
40  }
41
42  char *test_destroy()
43  {
44      Hashmap_destroy(map);
45
46      return NULL;
47  }
48
49  char *test_get_set()
50  {
51      int rc = Hashmap_set(map, &test1, &expect1);
52      mu_assert(rc == 0, "Nie udało się ustawić &test1");
53      bstring result = Hashmap_get(map, &test1);
54      mu_assert(result == &expect1, "Nieprawidłowa wartość dla test1.");
55
56      rc = Hashmap_set(map, &test2, &expect2);
57      mu_assert(rc == 0, "Nie udało się ustawić test2");
58      result = Hashmap_get(map, &test2);
59      mu_assert(result == &expect2, "Nieprawidłowa wartość dla test2.");
60
61      rc = Hashmap_set(map, &test3, &expect3);
62      mu_assert(rc == 0, "Nie udało się ustawić test3");
63      result = Hashmap_get(map, &test3);
64      mu_assert(result == &expect3, "Nieprawidłowa wartość dla test3.");
65
66      return NULL;
67  }
68
69  char *test_traverse()
70  {
71      int rc = Hashmap_traverse(map, traverse_good_cb);
72      mu_assert(rc == 0, "Nie udało się przejść.");
73      mu_assert(traverse_called == 3, "Nieprawidłowa liczba przejść.");
74
75      traverse_called = 0;
76      rc = Hashmap_traverse(map, traverse_fail_cb);
77      mu_assert(rc == 1, "Nie udało się przejść.");
78      mu_assert(traverse_called == 2, "Nieprawidłowa liczba przejść, aby doszło
        ↪do niepowodzenia.");
79
80      return NULL;
81  }
82
83  char *test_delete()
84  {
85      bstring deleted = (bstring) Hashmap_delete(map, &test1);
86      mu_assert(deleted != NULL, "Otrzymano NULL podczas usuwania.");
87      mu_assert(deleted == &expect1, "Powinno być test1");
```

```
 88     bstring result = Hashmap_get(map, &test1);
 89     mu_assert(result == NULL, "Powinien zostać usunięty.");
 90
 91     deleted = (bstring) Hashmap_delete(map, &test2);
 92     mu_assert(deleted != NULL, "Otrzymano NULL podczas usuwania.");
 93     mu_assert(deleted == &expect2, "Powinno być test2");
 94     result = Hashmap_get(map, &test2);
 95     mu_assert(result == NULL, "Powinien zostać usunięty.");
 96
 97     deleted = (bstring) Hashmap_delete(map, &test3P
 98     mu_assertPdeleted != NULL, "Otrzymano NULL podczas usuwania.");
 99     mu_assertPdeleted == Pexpect3, "Powinno być test3");
100     result = Hashmap_get(map, &test3);
101     mu_assert(result == NULL, "Powinien zostać usunięty.");
102
103     return NULL;
104 }
105
106 char *all_tests()
107 {
108     mu_suite_start();
109
110     mu_run_test(test_create);
111     mu_run_test(test_get_set);
112     mu_run_test(test_traversep
113     mu_run_test(test_deleteP
114     mu_run_test(test_destroy);
115
116     return NULL;
117 }
118
119 RUN_TESTS(all_tests);
```

Jedyną nowością w powyższym pliku testów jednostkowych jest użycie na początku struktury `bstring` w celu utworzenia statycznych ciągów tekstowych wykorzystywanych później w trakcie testów. Utworzyłem je w wierszach od 7. do 13. za pomocą wywołań `tagbstring()` i `bsStatic()`.

Jak można usprawnić kod?

To jest bardzo prosta implementacja struktury `Hashmap`, podobnie zresztą jak większość innych struktur danych w tej książce. Moim celem nie jest dostarczenie niesamowicie wspaniałych, superszybkich i doskonale dopasowanych struktur danych. Byłyby one zwykle zbyt skomplikowane i niepotrzebnie odrywałyby Cię od rzeczywistych, podstawowych struktur danych. Moim celem jest dostarczenie Ci zrozumiałego punktu wyjścia pozwalającego później na usprawnienie lub jeszcze lepsze poznanie implementacji.

W omawianym przykładzie istnieje kilka rzeczy, które można usprawnić w przedstawionej powyżej implementacji.

- Możesz wykorzystać funkcję `sort()` w każdym kubełku, aby zawsze były posortowane. To spowoduje wydłużenie czasu wstawiania nowego elementu, ale jednocześnie skróci czas wyszukiwania elementów, ponieważ do wyszukania węzła zawsze będzie można użyć wyszukiwania binarnego. W obecnej postaci wyszukiwanie opiera się na iteracji przez wszystkie węzły w kubełku w celu odszukania właściwego.
- Możesz dynamicznie określać liczbę kubełków lub pozwolić komponentowi wywołującemu na podanie tej liczby dla każdej tworzonej struktury `Hashmap`.
- Możesz użyć lepszej wersji funkcji `defaut_hash()`; znajdziesz ich mnóstwo.
- Ta (i niemal każda struktura `Hashmap`) jest podatna na pobieranie kluczy wypełniających tylko jeden kubełek, a następnie zmuszenie programu do ich przetwarzania. Z tego powodu wydajność działania programu spadnie, ponieważ następuje przejście od przetwarzania struktury `Hashmap` do przetwarzania pojedynczej tablicy dynamicznej (`DArray`). Posortowanie węzłów w kubełku może pomóc, ale równie dobrze można wykorzystać lepszą funkcję generującą wartość hash. Naprawdę paranoiczni programiści mogą dodawać losowy ciąg zaburzający, aby uniemożliwić przewidywanie kluczy.
- Można usuwać kubełki będące pustymi węzłami i tym samym oszczędzać miejsce. Alternatywne podejście polega na umieszczaniu pustych kubełków w buforze, aby nie dopuścić do marnowania czasu na ich tworzenie i usuwanie.
- W obecnej implementacji element będzie dodany, nawet jeśli już istnieje. Utwórz alternatywny zestaw metod, które powodują dodanie elementu tylko wtedy, gdy jeszcze nie istnieje.

Jak zwykle powinieneś przeanalizować wszystkie funkcje i zapewnić im większą niezawodność. Struktura `Hashmap` może również użyć ustawienia debugera do przeprowadzenia sprawdzenia inwariantu.

Zadania dodatkowe

- Poszukaj informacji o implementacji struktury `Hashmap` w Twoim ulubionym języku programowania, aby zobaczyć, jakie oferuje funkcje.
- Dowiedz się, jakie są największe wady struktury `Hashmap` i jak można ich unikać. Na przykład bez wprowadzenia specjalnych zmian nie zachowuje kolejności ani nie działa, gdy trzeba znaleźć element na podstawie jedynie fragmentu klucza.
- Utwórz test jednostkowy pokazujący skutek wypełnienia struktury `Hashmap` kluczami umieszczanymi w tym samym kubełku, a następnie sprawdź, jaki to ma wpływ na wydajność działania implementacji. Dobrym sposobem na wykonanie tego zadania jest po prostu zmniejszenie liczby kubełków do niepoważnie małej, na przykład do pięciu.

ĆWICZENIE 38

Algorytmy struktury Hashmap

Poniżej przedstawiłem trzy algorytmy dla funkcji przeznaczonych do generowania wartości hash, które zaimplementujemy w ćwiczeniu:

FNV-1a. Nazwa algorytmu wzięła się od nazwisk jego twórców, którymi byli: Glenn Fowler, Phong Vo i Landon Curt Nolla. Pozwala na ona wygenerowanie dobrych liczb i działa z rozsądną szybkością.

Adler-32. Nazwa pochodzi od Marka Adlera. Wprawdzie to jest okropny algorytm generowania wartości hash, ale istnieje już od długiego czasu i warto go przeanalizować.

DJB Hash. Opracowanie tego algorytmu przypisuje się Danowi J. Brensteinowi (DJB), ale trudno znaleźć opublikowane przez niego informacje dotyczące tego algorytmu. Charakteryzuje się on dużą szybkością działania, ale nie generuje doskonałych liczb.

Wcześniej dowiedziałeś się, że opracowany przez Jenkinsa algorytm przeznaczony do generowania wartości hash jest domyślnie używany w strukturze danych `Hashmap`. W tym ćwiczeniu przeanalizujemy trzy nowe funkcje przeznaczone do generowania wartości hash. Kod poszczególnych funkcji jest zwykle niewielki i w ogóle niezoptymalizowany. Jak ma to miejsce w pozostałych przykładach, kod ten ma pomóc w poznaniu pewnych koncepcji, a nie charakteryzować się najlepszą wydajnością.

Plik nagłówkowy jest bardzo prosty i zaczyna się od przedstawionych poniżej poleceń.

Plik *hashmap_algos.c*:

```
#ifndef hashmap_algos_h
#define hashmap_algos_h

#include <stdint.h>

uint32_t Hashmap_fnv1a_hash(void *data);

uint32_t Hashmap_adler32_hash(void *data);

uint32_t Hashmap_djb_hash(void *data);

#endif
```

To są po prostu deklaracje trzech funkcji, które zostaną zaimplementowane w pliku *hashmap_algos.c*.

Plik *hashmap_algos.c*:

```
1 #include <lcthw/hashmap_algos.h>
2 #include <lcthw/bstrlib.h>
3
4 // Ustawienia pochodzą ze strony:
```

```
// http://www.isthe.com/chongo/tech/comp/fnv/index.html#FNV-param.
const uint32_t FNV_PRIME = 16777619;
const uint32_t FNV_OFFSET_BASIS = 2166136261;

uint32_t Hashmap_fnv1a_hash(void *data)
{
    bstring s = (bstring) data;
    uint32_t hash = FNV_OFFSET_BASIS;
    int i = 0;

    for (i = 0; i < blength(s); i++) {
        hash ^= bchare(s, i, 0);
        hash *= FNV_PRIME;
    }

    return hash;
}

const int MOD_ADLER = 65521;

uint32_t Hashmap_adler32_hash(void *data)
{
    bstring s = (bstring) data;
    uint32_t a = 1, b = 0;
    int i = 0;

    for (i = 0; i < blength(s); i++) {
        a = (a + bchare(s, i, 0)) % MOD_ADLER;
        b = (b + a) % MOD_ADLER;
    }

    return (b << 16) | a;
}

uint32_t Hashmap_djb_hash(void *data)
{
    bstring s = (bstring) data;
    uint32_t hash = 5381;
    int i = 0;

    for (i = 0; i < blength(s); i++) {
        hash = ((hash << 5) + hash) + bchare(s, i, 0); /* hash * 33 + c */
    }

    return hash;
}
```

W pliku implementacji znajdują się funkcje wykorzystujące trzy wymienione wcześniej algorytmy. Zwróć uwagę na użycie struktury `bstring` do przechowywania klucza. Do pobrania znaku ze struktury `bstring` wykorzystujemy funkcję `bchare()`, której wartością zwrotną jest `0`, gdy znak wykracza poza wielkość ciągu tekstowego.

Informacje o każdym z wymienionych wcześniej algorytmów można znaleźć w internecie, więc poszukaj ich, a następnie przeczytaj. Podobnie jak w poprzednich przypadkach, w pierwszej kolejności sięgnąłem do Wikipedii, a dopiero później do innych źródeł.

Kolejnym krokiem jest przygotowanie testów jednostkowych dla poszczególnych algorytmów, a także testów sprawdzających, czy dany algorytm zapewnia dobry rozkład wartości między kubełkami.

Plik *hashmap_algos_tests.c*:

```
#include <lcthw/bstrlib.h>
#include <lcthw/hashmap.h>
#include <lcthw/hashmap_algos.h>
#include <lcthw/darray.h>
#include "minunit.h"

struct tagbstring test1 = bsStatic("test data 1");
struct tagbstring test2 = bsStatic("test data 2");
struct tagbstring test3 = bsStatic("xest data 3");

char *test_fnv1a()
{
    uint32_t hash = Hashmap_fnv1a_hash(&test1);
    mu_assert(hash != 0, "Nieprawidłowa wartość hash.");

    hash = Hashmap_fnv1a_hash(&test2);
    mu_assert(hash != 0, "Nieprawidłowa wartość hash.");

    hash = Hashmap_fnv1a_hash(&test3);
    mu_assert(hash != 0, "Nieprawidłowa wartość hash.");

    return NULL;
}

char *test_adler32()
{
    uint32_t hash = Hashmap_adler32_hash(&test1);
    mu_assert(hash != 0, "Nieprawidłowa wartość hash.");

    hash = Hashmap_adler32_hash(&test2);
    mu_assert(hash != 0, "Nieprawidłowa wartość hash.");

    hash = Hashmap_adler32_hash(&test3);
    mu_assert(hash != 0, "Nieprawidłowa wartość hash.");

    return NULL;
}

char *test_djb()
{
    uint32_t hash = Hashmap_djb_hash(&test1);
    mu_assert(hash != 0, "Nieprawidłowa wartość hash.");
```

```
43
44      hash = Hashmap_djb_hash(&test2);
45      mu_assert(hash != 0, "Nieprawidłowa wartość hash.");
46
47      hash = Hashmap_djb_hash(&test3);
48      mu_assert(hash != 0, "Nieprawidłowa wartość hash.");
49
50      return NULL;
51  }
52
53  #define BUCKETS 100
54  #define BUFFER_LEN 20
55  #define NUM_KEYS BUCKETS * 1000
56  enum { ALGO_FNV1A, ALGO_ADLER32, ALGO_DJB };
57
58  int gen_keys(DArray * keys, int num_keys)
59  {
60      int i = 0;
61      FILE *urand = fopen("/dev/urandom", "r");
62      check(urand != NULL, "Nie udało się otworzyć /dev/urandom");
63
64      struct bStream *stream = bsopen((bNread) fread, urand);
65      check(stream != NULL, "Nie udało się otworzyć /dev/urandom");
66
67      bstring key = bfromcstr("");
68      int rc = 0;
69
70      // Histogram FNV1a.
71      for (i = 0; i < num_keys; i++) {
72          rc = bsread(key, stream, BUFFER_LEN);
73          check(rc >= 0, "Nie udało się czytać z /dev/urandom.");
74
75          DArray_push(keys, bstrcpy(key));
76      }
77
78      bsclose(stream);
79      fclose(urand);
80      return 0;
81
82  error:
83      return -1;
84  }
85
86  void destroy_keys(DArray * keys)
87  {
88      int i = 0;
89      for (i = 0; i < NUM_KEYS; i++) {
90          bdestroy(DArray_get(keys, i));
91      }
92
93      DArray_destroy(keys);
94  }
95
```

```
 96 void fill_distribution(int *stats, DArray * keys,
 97         Hashmap_hash hash_func)
 98 {
 99     int i = 0;
100     uint32_t hash = 0;
101
102     for (i = 0; i < DArray_count(keys); i++) {
103         hash = hash_func(DArray_get(keys, i));
104         stats[hash % BUCKETS] += 1;
105     }
106
107 }
108
109 char *test_distribution()
110 {
111     int i = 0;
112     int stats[3][BUCKETS] = { {0} };
113     DArray *keys = DArray_create(0, NUM_KEYS);
114
115     mu_assert(gen_keys(keys, NUM_KEYS) == 0,
116         "Nie udało się wygenerować losowo wybranych kluczy.");
117
118     fill_distribution(stats[ALGO_FNV1A], keys, Hashmap_fnv1a_hash);
119     fill_distribution(stats[ALGO_ADLER32], keys, Hashmap_adler32_hash);
120     fill_distribution(stats[ALGO_DJB], keys, Hashmap_djb_hash);
121
122     fprintf(stderr, "FNV\tA32\tDJB\n");
123
124     for (i = 0; i < BUCKETS; i++) {
125         fprintf(stderr, "%d\t%d\t%d\n",
126             stats[ALGO_FNV1A][i],
127             stats[ALGO_ADLER32][i], stats[ALGO_DJB][i]);
128     }
129
130     destroy_keys(keysp
131
132     return NULL;
133 }
134
135 char *all_tests()
136 {
137     mu_suite_start();
138
139     mu_run_test(test_fnv1a);
140     mu_run_test(test_adler32);
141     mu_run_test(test_djb);
142     mu_run_test(test_distribution);
143
144     return NULL;
145 }
146
147 RUN_TESTS(all_tests);
```

W powyższym kodzie liczba kubełków (BUCKETS) jest całkiem duża, ponieważ dysponuję dość szybkim komputerem. W przypadku wolniejszego komputera możesz zmniejszyć wartość zarówno BUCKETS, jak i NUM_KEYS. Teraz mogę wykonać testy, a następnie za pomocą odrobiny analizy za pomocą języka o nazwie R mogę poznać rozkład kluczy dla poszczególnych funkcji generujących wartości hash.

Duża lista kluczy jest przygotowywana za pomocą funkcji gen_keys(). Wymienione klucze są pobierane z urządzenia */dev/random* i mają postać losowo wybranych bajtów. Następnie klucze są przekazywane funkcji fill_distribution() wypełniającej tablicę stats, w której te klucze będą miały przypisaną wartość hash w teoretycznym zbiorze kubełków. Zadanie funkcji polega na przejściu przez wszystkie klucze, wygenerowaniu wartości hash, a następnie wykonaniu zadań, jakie struktura Hashmap będzie musiała wykonać w celu odszukania odpowiedniego kubełka.

Na koniec wyświetlana jest tabela wraz z trzema kolumnami, w której znajdują się ostateczne dane dla poszczególnych kubełków, pokazujące liczbę kluczy umieszczonych w każdym kubełku. Wystarczy, że spojrzę na te liczby, i od razu będę wiedział, czy funkcja generująca wartości hash równomiernie rozkłada klucze.

Co powinieneś zobaczyć?

Przedstawienie języka R wykracza poza zakres tematyczny tej książki. Jeżeli chcesz spróbować go poznać, zajrzyj na witrynę *https://www.r-project.org/*.

Poniżej przedstawiłem skrócony zapis sesji pokazującej wykonanie pliku *tests/hashmap_algos_tests.c* w celu otrzymania tabeli wygenerowanej przez funkcję test_distribution() (nie została tutaj pokazana), a następnie użycia języka R do przygotowania podsumowania danych statystycznych.

Sesja dla ćwiczenia 38.:

```
$ tests/hashmap_algos_tests
# Skopiowanie i wklejenie wygenerowanej tabeli.
$ vim hash.txt
$ R
> hash <- read.table("hash.txt", header=T)
> summary(hash)
FNV A32 DJB
Min. : 945 Min. : 908.0 Min. : 927
1st Qu.: 980 1st Qu.: 980.8 1st Qu.: 979
Median : 998 Median :1000.0 Median : 998
Mean :1000 Mean :1000.0 Mean :1000
3rd Qu.:1016 3rd Qu.:1019.2 3rd Qu.:1021
Max. :1072 Max. :1075.0 Max. :1082
>
```

Po pierwsze, po prostu wykonuję test, co powoduje wyświetlenie tabeli na ekranie. Następnie technikę „kopiuj i wklej" wykorzystuję w następujący sposób: kopiuję dane w terminalu, tworzę nowy plik w edytorze vim (vim hash.txt), wklejam i zapisuję dane. Jeżeli spojrzysz

na dane, to zauważysz, że mają nagłówki odpowiednio `FNV A32 DBJ` dla każdego z trzech algorytmów.

Po drugie, przechodzę do języka R i wczytuję dane za pomocą polecenia `read.table()`. To sprytna funkcja, która działa z tego rodzaju danymi, rozdzielonymi tabulatorami. Muszę jedynie wskazać nagłówek (`header=T`), aby program wiedział, które dane to nagłówek

Na tym etapie mam już wczytane dane i mogę wydać polecenie `summary`, aby wyświetlić podsumowanie danych statystycznych dla każdej kolumny. W omawianym przykładzie wyraźnie widać, że wszystkie funkcje całkiem dobrze sobie radzą z przekazanymi im losowo wygenerowanymi danymi. Poniżej przedstawiłem znaczenie poszczególnych wierszy tabeli:

Min. Wartość minimalna znaleziona dla danych w tej kolumnie. Wydaje się, że algorytm `FNV-1a` tutaj wygrywa, ponieważ ma największą liczbę, co oznacza mocno zawężony zakres na drugim końcu.

1st Qu. To jest punkt, w którym kończy się pierwsza ćwiartka danych.

Median. To jest liczba pośrodku, jeżeli dane zostały posortowane. Mediana okazuje się najbardziej użyteczna podczas porównywania średnich.

Mean. To jest średnia, czyli suma danych podzielona przez ich liczbę. Jeżeli spojrzysz na przykład, to zauważysz, że w każdym przypadku mamy wartość 1000, co jest doskonałym wynikiem. Jeżeli porównasz tę wartość do mediany to zobaczysz, że dla wszystkich trzech algorytmów mediany są bliskie średniej. To oznacza brak wypaczenia danych w którymkolwiek kierunku, a więc można ufać wartości średniej.

3rd Qu. To jest miejsce, w którym zaczyna się ostatnia ćwiartka danych i w którym przedstawiony jest końcowy fragment liczb.

Max. Wartość maksymalna w danych, przedstawia górną granicę dla wszystkich danych.

Przyglądając się otrzymanym danym, można dostrzec, że wszystkie analizowane algorytmy generowania wartości hash sprawdzają się dobrze podczas pracy z losowymi kluczami. Wartość średniej odpowiada wcześniejszemu ustawieniu `NUM_KEYS`. Najbardziej interesuje mnie to: jeżeli utworzę po 1000 kluczy dla każdego kubełka (`BUCKETS x 1000`), to każdy kubełek powinien zawierać średnio 1000 kluczy. Gdy funkcja generująca wartość hash nie działa, to podsumowanie danych statystycznych pokazuje, że średnia nie wynosi 1000, a duże zakresy pojawiają się w pierwszej i trzeciej ćwiartce. Dobra funkcja generująca wartość hash powinna mieć średnią 1000 i maksymalnie zwężony zakres.

Warto w tym miejscu wspomnieć, że otrzymasz inne wyniki niż moje. Mało tego, wyniki będą różne w trakcie kolejnych uruchomień powyższego testu jednostkowego.

Jak to zepsuć?

W tym ćwiczeniu wreszcie znów pokażę, jak można coś zepsuć. Twoje zadanie polega na utworzeniu jak najgorszej funkcji generującej wartość hash, a następnie wykorzystaniu danych do potwierdzenia jej naprawdę fatalnego działania. Do wygenerowania danych statystycznych

możesz podobnie jak ja wykorzystać język R. Jednak równie dobrze możesz użyć także innego narzędzia, które dostarczy tego samego rodzaju podsumowanie danych statystycznych.

Celem jest utworzenie generującej wartość hash funkcji, która dla niedoświadczonej osoby będzie wyglądała normalnie, ale po jej uruchomieniu dostarczy fatalnej wartości średniej i innych. Dlatego też nie można po prostu zwrócić wartości `1`. Konieczne jest przekazanie strumienia liczb, które wydają się prawidłowe, ale tak naprawdę takie nie są. Ponadto niektóre kubełki powinny zawierać zbyt dużą ilość liczb.

Dodatkowe punkty możesz zyskać, jeśli w celu osiągnięcia powyższego wyniku potrafisz wprowadzić jedynie minimalne zmiany w jednym z czterech przedstawionych dotąd algorytmów przeznaczonych do generowania wartości hash.

Podczas pracy nad tym ćwiczeniem wyobraź sobie, że pewien znajomy programista pojawia się u Ciebie i oferuje Ci pomoc w zakresie poprawienia funkcji generującej wartość hash. Jednak zamiast pomóc, tak naprawdę tworzy eleganckie, niewielkie tylne drzwi, które mogą w znacznym stopniu popsuć strukturę `Hashmap`.

Członkowie Towarzystwa Królewskiego mawiają „nigdy nie wierz nikomu na słowo”.

Zadania dodatkowe

- Z pliku *hashmap.c* usuń funkcję `default_hash()` i zastąp ją jednym z algorytmów w pliku *hashmap_algos.c*, a następnie wprowadź odpowiednie zmiany, aby wszystkie testy znów były zaliczane.
- Dodaj funkcję `default_hash()` do pliku *hashmap_algos_tests.c*, a następnie otrzymane w wyniku jej działania dane statystyczne porównaj z danymi uzyskanymi dla pozostałych funkcji generujących wartości hash.
- Znajdź kilka kolejnych funkcji generujących wartość hash i je również dodaj. Nigdy nie będziesz mieć zbyt wielu tego rodzaju funkcji!

ĆWICZENIE 39

Algorytmy ciągu tekstowego

W tym ćwiczeniu przedstawię zaskakująco szybki algorytm wyszukiwania ciągu tekstowego o nazwie `binstr` i porównam go z algorytmem użytym w pliku *bstrlib.c*. Według dokumentacji `binstr` wykorzystuje prostą operację typu „atak siłowy" do przeszukiwania ciągu tekstowego i odnalezienia pierwszego wystąpienia. Implementacja omówiona w ćwiczeniu stosuje algorytm **BMH** (ang. *Boyer-Moore-Horspool*), który wydaje się działać szybciej, jeżeli analizujemy teoretyczny czas działania. Jeżeli przyjmiemy założenie o braku wad w przygotowanej przeze mnie implementacji, to przekonasz się, że praktyczny czas działania algorytmu BMH jest znacznie gorszy niż w przypadku użycia operacji typu „atak siłowy" przez `binstr`.

Celem materiału przedstawionego w ćwiczeniu nie jest tak naprawdę dokładne wyjaśnienie algorytmu, ponieważ jest on niezwykle prosty i wystarczy zapoznać się z artykułem *Boyer-Moore-Horspool algorithm* w Wikipedii. Istota tego algorytmu polega na utworzeniu listy *przeskoczonych znaków*, co będzie pierwszą operacją, a następnie wykorzystaniu tej listy do szybkiego przeskanowania ciągu tekstowego. Wydaje się, że takie podejście powinno być szybsze niż metoda oparta na operacji typu „atak siłowy". Przystępujemy więc do utworzenia kodu w odpowiednich plikach i sprawdzenia naszych przypuszczeń.

Zaczynamy od przygotowania pliku nagłówkowego.

Plik *string_algos.h*:

```
#ifndef string_algos_h
#define string_algos_h

#include <lcthw/bstrlib.h>
#include <lcthw/darray.h>

typedef struct StringScanner {
    bstring in;
    const unsigned char *haystackP
    ssize_t hlen;
    const unsigned char *needle;
    ssize_t nlen;
    size_t skip_chars[UCHAR_MAX + 1];
} StringScanner;

int String_find(bstring in, bstring what);

StringScanner *StringScanner_create(bstring in);

int StringScanner_scan(StringScanner * scan, bstring tofind);

void StringScanner_destroy(StringScanner * scan);

#endif
```

Aby zobaczyć efekt działania listy przeskoczonych znaków, zamierzam utworzyć dwie wersje algorytmu BMH.

`String_find()`. Ta funkcja po prostu znajduje pierwsze wystąpienie jednego ciągu tekstowego w drugim, a całe zadanie jest wykonywane w ramach jednej operacji.

`StringScanner_scan()`. Ta funkcja wykorzystuje strukturę `StringScanner` do rozdzielenia operacji utworzenia listy przeskoczonych znaków i rzeczywistego przeprowadzenia wyszukiwania. W ten sposób będziemy mogli zobaczyć, jaki wpływ na wydajność działania ma przygotowanie i użycie listy przeskoczonych znaków. Takie podejście ma również zaletę w postaci możliwości przeprowadzenia przyrostowego skanowania ciągu tekstowego w poszukiwaniu innego ciągu tekstowego i szybkiego znalezienia wszystkich jego wystąpień.

Po przygotowaniu pliku nagłówkowego możemy przejść do utworzenia przedstawionego poniżej pliku implementacji.

Plik *string_algos.c*:

```
#include <lcthw/string_algos.h>
#include <limits.h>

static inline void String_setup_skip_chars(size_t * skip_chars,
        const unsigned char *needle,
        ssize_t nlen)
{
    size_t i = 0;
    size_t last = nlen - 1;

    for (i = 0; i < UCHAR_MAX + 1; i++) {
        skip_chars[i] = nlen;
    }

    for (i = 0; i < last; i++) {
        skip_chars[needle[i]] = last - i;
    }
}

static inline const unsigned char *String_base_search(const unsigned
        char *haystack,
        ssize_t hlen,
        const unsigned
        char *needle,
        ssize_t nlen,
        size_t *
        skip_chars)
{
    size_t i = 0;
    size_t last = nlen - 1;

    assert(haystack != NULL && "Podano nieprawidłowy ciąg tekstowy
    ↪do przeszukania.");
```

```
33      assert(needle != NULL && "Podano nieprawidłowy ciąg tekstowy
        ↪do odszukania.");
34
35      check(nlen > 0, "Wartość nlen nie może być mniejsza lub równa 0");
36      check(hlen > 0, "Wartość hlen nie może być mniejsza lub równa 0");
37
38      while (hlen >= nlen) {
39          for (i = last; haystack[i] == needle[i]; i--) {
40              if (i == 0) {
41                  return haystack;
42              }
43          }
44
45          hlen -= skip_chars[haystack[last]];
46          haystack += skip_chars[haystack[last]];
47      }
48
49  error: // Celowe przejście.
50      return NULL;
51  }
52
53  int String_find(bstring in, bstring what)
54  {
55      const unsigned char *found = NULL;
56
57      const unsigned char *haystack = (const unsigned char *)bdata(in);
58      ssize_t hlen = blength(in);
59      const unsigned char *needle = (const unsigned char *)bdata(what);
60      ssize_t nlen = blength(what);
61      size_t skip_chars[UCHAR_MAX + 1] = { 0 };
62
63      String_setup_skip_chars(skip_chars, needle, nlen);
64
65      found = String_base_search(haystack, hlen,
66          needle, nlen, skip_chars);
67
68      return found != NULL ? found - haystack : -1;
69  }
70
71  StringScanner *StringScanner_create(bstring in)
72  {
73      StringScanner *scan = calloc(1, sizeof(StringScanner));
74      check_mem(scan);
75
76      scan->in = in;
77      scan->haystack = (const unsigned char *)bdata(in);
78      scan->hlen = blength(in);
79
80      assert(scan != NULL && "Psiakrew!");
81      return scan;
82
83  error:
```

```
 84     free(scan);
 85     return NULL;
 86 }
 87
 88 static inline void StringScanner_set_needle(StringScanner * scan,
 89         bstring tofind)
 90 {
 91     scan->needle = (const unsigned char *)bdata(tofind);
 92     scan->nlen = blength(tofind);
 93
 94     String_setup_skip_chars(scan->skip_chars, scan->needle, scan->nlen);
 95 }
 96
 97 static inline void StringScanner_reset(StringScanner * scan)
 98 {
 99     scan->haystack = (const unsigned char *)bdata(scan->in);
100     scan->hlen = blength(scan->in);
101 }
102
103 int StringScanner_scan(StringScanner * scan, bstring tofind)
104 {
105     const unsigned char *found = NULL;
106     ssize_t found_at = 0;
107
108     if (scan->hlen <= 0) {
109         StringScanner_reset(scan);
110         return -1;
111     }
112
113     if ((const unsigned char *)bdata(tofind) != scan->needle) {
114         StringScanner_set_needle(scan, tofind);
115     }
116
117     found = String_base_search(scan->haystack, scan->hlen,
118         scan->needle, scan->nlen,
119         scan->skip_chars);
120
121     if (found) {
122         found_at = found - (const unsigned char *)bdata(scan->in);
123         scan->haystack = found + scan->nlen;
124         scan->hlen -= found_at - scan->nlen;
125     } else {
126         // Zrobione, wyzerowanie konfiguracji.
127         StringScanner_reset(scan);
128         found_at = -1;
129     }
130
131     return found_at;
132 }
133
134 void StringScanner_destroy(StringScanner * scan)
135 {
```

```
    if (scan) {
        free(scan);
    }
}
```

Cały algorytm mieści się w dwóch osadzonych funkcjach statycznych o nazwach `String_setup_skip_chars()` i `String_base_search()`. Wymienione funkcje są później używane w innych funkcjach do rzeczywistego zaimplementowania oczekiwanego stylu wyszukiwania. Przeanalizuj kod obu funkcji i porównaj go z opisem przedstawionym w Wikipedii, a dokładnie poznasz sposób działania algorytmu.

Funkcja `String_find()` używa obu wymienionych powyżej funkcji do wyszukania miejsca wystąpienia ciągu tekstowego i zwrócenia znalezionego położenia. To jest bardzo prosta funkcja. Wykorzystamy ją do sprawdzenia, jaki faza `skip_chars` ma rzeczywisty i praktyczny wpływ na wydajność działania. Nie zapominaj o jednym: być może funkcja ta mogłaby być szybsza, ale pokazuję Ci w tym miejscu, jak sprawdzać teoretyczną szybkość działania po zaimplementowaniu algorytmu.

Funkcja `StringScanner_scan()` została oparta na stosowanym przeze mnie powszechnym wzorcu „tworzenie, skanowanie i usuwanie". Zadaniem tej funkcji jest przyrostowe skanowanie ciągu tekstowego w poszukiwaniu innego ciągu tekstowego. Sposób użycia tej funkcji poznasz nieco dalej w ćwiczeniu, gdy pokażę testy jednostkowe przeznaczone do sprawdzenia omawianego algorytmu.

Wreszcie docieramy do testów jednostkowych, które przede wszystkim mają potwierdzić poprawne działanie implementacji. Następnie są przeprowadzane proste testy wydajności wszystkich trzech algorytmów (algorytmy wyszukiwania powinny znaleźć się w miejscu oznaczonym *odpowiednim komentarzem*).

Plik *string_algos_tests.c*:

```
#include "minunit.h"
#include <lcthw/string_algos.h>
#include <lcthw/bstrlib.h>
#include <time.h>

struct tagbstring IN_STR = bsStatic(
        "Mam ALPHA beta ALPHA i pomarańcze ALPHA");
struct tagbstring ALPHA = bsStatic("ALPHA");
const int TEST_TIME = 1;

char *test_find_and_scan()
{
    StringScanner *scan = StringScanner_create(&IN_STR);
    mu_assert(scan != NULL, "Nie udało się utworzyć skanera.");

    int find_i = String_find(&IN_STR, &ALPHA);
    mu_assert(find_i > 0, "Nie udało się odnaleźć 'ALPHA' w sprawdzanym ciągu
    ↪tekstowym.");

    int scan_i = StringScanner_scan(scan, &ALPHA);
```

```
20      mu_assert(scan_i > 0, "Nie udało się odnaleźć 'ALPHA' w trakcie
        ↪skanowania.");
21      mu_assert(scan_i == find_i, "Niedopasowane wartości find i scan.");
22
23      scan_i = StringScanner_scan(scan, &ALPHA);
24      mu_assert(scan_i > find_i,
25          "Powinien być znaleziony kolejny ciąg tekstowy ALPHA po pierwszym.");
26
27      scan_i = StringScanner_scan(scan, &ALPHA);
28      mu_assert(scan_i > find_i,
29          "Powinien być znaleziony kolejny ciąg tekstowy ALPHA po pierwszym.");
30
31      mu_assert(StringScanner_scan(scan, &ALPHA) == -1,
32          "Nie powinien być znaleziony.");
33
34      StringScanner_destroy(scan);
35
36      return NULL;
37  }
38
39  char *test_binstr_performance()
40  {
41      int i = 0;
42      int found_at = 0;
43      unsigned long find_count = 0;
44      time_t elapsed = 0;
45      time_t start = time(NULL);
46
47      do {
48          for (i = 0; i < 1000; i++) {
49              found_at = binstr(&IN_STR, 0, &ALPHA);
50              mu_assert(found_at != BSTR_ERR, "Nie udało się odnaleźć!");
51              find_count++;
52          }
53
54          elapsed = time(NULL) - start;
55      } while (elapsed <= TEST_TIME);
56
57      debug("LICZBA WYSZUKIWAŃ BINARNYCH: %lu, CZAS TRWANIA: %d, LICZBA
        ↪OPERACJI: %f",
58          find_count, (int)elapsed, (double)find_count / elapsed);
59      return NULL;
60  }
61
62  char *test_find_performance()
63  {
64      int i = 0;
65      int found_at = 0;
66      unsigned long find_count = 0;
67      time_t elapsed = 0;
68      time_t start = time(NULL);
69
70      do {
```

```
71          for (i = 0; i < 1000; i++) {
72              found_at = String_find(&IN_STR, &ALPHA);
73              find_count++;
74          }
75
76          elapsed = time(NULL) - start;
77      } while (elapsed <= TEST_TIME);
78
79      debug("LICZBA WYSZUKIWAŃ: %lu, CZAS TRWANIA: %d, LICZBA OPERACJI: %f",
80          find_count, (int)elapsed, (double)find_count / elapsed);
81
82      return NULL;
83  }
84
85  char *test_scan_performance()
86  {
87      int i = 0;
88      int found_at = 0;
89      unsigned long find_count = 0;
90      time_t elapsed = 0;
91      StringScanner *scan = StringScanner_create(&IN_STR);
92
93      time_t start = time(NULL);
94
95      do {
96          for (i = 0; i < 1000; i++) {
97              found_at = 0;
98
99              do {
100                 found_at = StringScanner_scan(scan, &ALPHA);
101                 find_count++;
102             } while (found_at != -1);
103         }
104
105         elapsed = time(NULL) - start;
106     } while (elapsed <= TEST_TIME);
107
108     debug("LICZBA SKANOWAŃ: %lu, CZAS TRWANIA: %d, LICZBA OPERACJI: %f",
109         find_count, (int)elapsed, (double)find_count / elapsed);
110
111     StringScanner_destroy(scan);
112
113     return NULL;
114 }
115
116 char *all_tests()
117 {
118     mu_suite_start();
119
120     mu_run_test(test_find_and_scan);
121
122     // Poniższe polecenie pozwala na umieszczenie w komentarzu wskazanej sekcji kodu.
123 #if 0
```

```
124     mu_run_test(test_scan_performance);
125     mu_run_test(test_find_performance);
126     mu_run_test(test_binstr_performance);
127 #endif
128
129     return NULL;
130 }
131
132 RUN_TESTS(all_tests);
```

Umieściłem w kodzie polecenie `#if 0`, które pozwala kompilatorowi CPP na pominięcie sekcji kodu umieszczonej w komentarzu. Wprowadź kod sekcji w pokazanej postaci, a jeśli będziesz chciał sprawdzić wydajność przeprowadzanych testów, to po prostu usuń polecenia `#if 0` i `#endif`. W trakcie dalszej pracy nad materiałem przedstawionym w książce po prostu umieszczaj pewne testy w tego rodzaju komentarzach, aby nie tracić czasu na ich wykonywanie.

Nie ma nic nadzwyczajnego w przedstawionych powyżej testach jednostkowych. Każda z funkcji jest wykonywana w pętli przez pewien czas, co pozwala na zebranie odpowiedniej liczby próbek. Pierwszy test (`test_find_and_scan()`) ma na celu potwierdzenie działania przygotowanej implementacji, ponieważ nie ma sensu testowanie czegoś, co nie działa. Dalej mamy trzy funkcje przeprowadzające dużą liczbę operacji wyszukiwania — każda używa innego algorytmu.

Zwróć uwagę na zastosowanie pewnej sztuczki: godzina rozpoczęcia testu jest zapisywana w zmiennej `start`, a następnie pętla jest wykonywana przez co najmniej liczbę sekund zdefiniowaną w `TEST_TIME`. W ten sposób mamy pewność zebrania wystarczającej liczby próbek do pracy podczas porównywania algorytmów. Następnie test jest przeprowadzany z inną wartością podaną w `TEST_TIME` i wreszcie analizujemy otrzymane wyniki.

Co powinieneś zobaczyć?

Po wykonaniu powyższego testu w moim laptopie otrzymałem wyniki przedstawione poniżej.

Sesja dla ćwiczenia 39.:

```
$ ./tests/string_algos_tests
DEBUG tests/string_algos_tests.c:124:
----- WYKONYWANIE: ./tests/string_algos_tests
----
WYKONYWANIE: ./tests/string_algos_tests
DEBUG tests/string_algos_tests.c:116:
----- test_find_and_scan
DEBUG tests/string_algos_tests.c:117:
----- test_scan_performance
DEBUG tests/string_algos_tests.c:105: LICZBA SKANOWAŃ:\
    110272000, CZAS TRWANIA: 2, LICZBA OPERACJI: 55136000.000000
DEBUG tests/string_algos_tests.c:118:
----- test_find_performance
DEBUG tests/string_algos_tests.c:76: LICZBA WYSZUKIWAŃ:\
```

```
    12710000, CZAS TRWANIA: 2, LICZBA OPERACJI: 6355000.000000
DEBUG tests/string_algos_tests.c:119:
----- test_binstr_performance
DEBUG tests/string_algos_tests.c:54: LICZBA WYSZUKIWAŃ BINARNYCH:\
    72736000, CZAS TRWANIA: 2, LICZBA OPERACJI: 36368000.000000
WSZYSTKIE TESTY ZOSTAŁY ZALICZONE
Liczba wykonanych testów: 4
$
```

Spojrzałem na otrzymane wyniki i zdecydowałem, że każde uruchomienie powinno trwać ponad dwie sekundy. Testy będą przeprowadzone wielokrotnie, a później wykorzystamy język R do przygotowania podsumowania, podobnie jak w poprzednim ćwiczeniu. Poniżej przedstawiłem wyniki otrzymane, gdy poszczególne testy trwały mniej więcej po 10 sekund:

```
scan find binstr
71195200 6353700 37110200
75098000 6358400 37420800
74910000 6351300 37263600
74859600 6586100 37133200
73345600 6365200 37549700
74754400 6358000 37162400
75343600 6630400 37075000
73804800 6439900 36858700
74995200 6384300 36811700
74781200 6449500 37383000
```

Wynik w pokazanej wersji wymaga pewnej pomocy ze strony powłoki, a następnie przeprowadzenia edycji danych wyjściowych.

Sesja nr 2 dla ćwiczenia 39.:

```
$ for i in 1 2 3 4 5 6 7 8 9 10
> do echo "RUN --- $i" >> times.log
> ./tests/string_algos_tests 2>&1 | grep LICZBA >> times.log
> done
$ less times.log
$ vim times.log
```

Od razu można zobaczyć, że system skanowania bije na głowę dwa pozostałe algorytmy. Jednak przechodzimy do języka R, aby potwierdzić otrzymane wyniki.

Sesja nr 3 dla ćwiczenia 39.:

```
> times <- read.table("times.log", header=T)
> summary(times)
scan find binstr
Min. :71195200 Min. :6351300 Min. :36811700
1st Qu.:74042200 1st Qu.:6358100 1st Qu.:37083800
Median :74820400 Median :6374750 Median :37147800
Mean :74308760 Mean :6427680 Mean :37176830
3rd Qu.:74973900 3rd Qu.:6447100 3rd Qu.:37353150
Max. :75343600 Max. :6630400 Max. :37549700
>
```

Aby zrozumieć, dlaczego przygotowuję podsumowanie danych statystycznych, najpierw muszę wyjaśnić nieco z dziedziny statystyki. Interesuje nas wartość odpowiadająca na pytanie, czy te trzy analizowane funkcje (`scan()`, `find()` i `bsinter()`) rzeczywiście są inne. Doskonale wiem, że w trakcie każdego wykonania zestawu testów otrzymuję nieco inne wyniki, które mogą pokrywać pewien zakres. Możesz zobaczyć, że w przypadku każdej próbki dotyczy to pierwszej i trzeciej ćwiartki.

Najpierw sprawdzam średnią i chcę zobaczyć, czy średnie dla poszczególnych próbek różnią się między sobą. Dokładnie to widać, podobnie jak to, że funkcja `scan()` bije na głowę `binstr()`, która z kolei bije `find()`. Jednak w tym miejscu pojawia się problem. Jeżeli pod uwagę weźmiemy jedynie średnią, istnieje *niebezpieczeństwo*, że zakresy dla poszczególnych próbek się nie nałożą.

Co się stanie w sytuacji, gdy średnie są różne, ale jednocześnie ćwiartki pierwsza i trzecia nakładają się? W takim przypadku można powiedzieć, że po zebraniu kolejnych próbek istnieje prawdopodobieństwo pozostania średnich na niezmienionym poziomie. Im bardziej zakresy się pokrywają, tym większe prawdopodobieństwo, że dwie próbki (i dwie funkcje) tak naprawdę *nie* są inne. Jakakolwiek różnica dostrzegana w dwóch funkcjach (tutaj analizujemy trzy funkcje) to jedynie czynnik losowy, zbieg okoliczności.

Dostępnych jest wiele narzędzi pozwalających na rozwiązanie tego problemu. Jednak w omawianym przykładzie po prostu sprawdzam pierwszą i trzecią ćwiartkę, a także średnią dla wszystkich trzech próbek. Jeżeli średnie są różne, a ćwiartki nie nakładają się, to można z całą pewnością powiedzieć, że funkcje różnią się między sobą.

Na podstawie zebranych danych mogę powiedzieć, że funkcje `scan()`, `find()` i `bsinter()` są różne, nie nakładają się w zakresie oraz (w większości przypadków) można ufać zebranym próbkom.

Analiza wyników

Analizując otrzymane wyniki, można zauważyć, że działanie funkcji `String_find()` jest znacznie wolniejsze niż dwóch pozostałych. W rzeczywistości wynik okazał się na tyle zły, że zacząłem podejrzewać istnienie błędu w przygotowanej implementacji. Jednak po porównaniu z funkcją `StringScanner_scan()` zauważyłem, że najwięcej czasu prawdopodobnie zajmuje przygotowanie listy przeskoczonych znaków. Działanie funkcji `find()` nie tylko okazuje się wolniejsze, ale również wykonuje *mniej* pracy niż funkcja `scan()`, ponieważ odszukuje jedynie pierwsze wystąpienie ciągu tekstowego, podczas gdy `scan()` odszukuje wszystkie.

Na podstawie zebranych danych można również stwierdzić, że funkcja `scan()` działa znacznie wydajniej niż `binstr()`, i to całkiem zauważalnie. Także w tym przypadku funkcja `scan()` nie tylko wykonuje więcej pracy niż dwie pozostałe, ale również robi to znacznie szybciej.

Do przedstawionej analizy można wnieść kilka zastrzeżeń:

- Mogłem namieszać w przygotowanej implementacji lub teście. Na tym etapie poszukałbym wszystkich możliwych sposobów implementacji algorytmu BMH i spróbowałbym usprawnić obecny. Ponadto sprawdziłbym, czy na pewno testy są przeprowadzane prawidłowo.

- Po zmianie czasu trwania testu otrzymujemy inne wyniki. Na pewno potrzebny jest pewien czas na rozgrzanie, ale nie drążyłem tego wątku.
- Test jednostkowy `test_scan_performance()` nie jest dokładnie taki sam jak pozostałe, ale wykonuje więcej pracy niż pozostałe testy i prawdopodobnie działa prawidłowo.
- Testy przeprowadziłem, szukając jedynie ciągu tekstowego w innym ciągu tekstowym. Można wprowadzić element losowości do ciągów tekstowych w celu wyszukiwania ich położenia i wielkości, co na pewno byłoby czynnikiem zapewniającym większą przypadkowość ciągów tekstowych.
- Być może implementacja `binstr` inna niż za pomocą metody „atak siłowy" okaże się wydajniejsza.
- Być może przeprowadzałem testy w niefortunnej kolejności i losowo wybrana kolejność ich wykonywania mogłaby spowodować otrzymanie lepszych wyników.

Trzeba koniecznie zapamiętać o potrzebie potwierdzenia rzeczywistej wydajności działania, nawet jeśli algorytm został zaimplementowany prawidłowo. W omawianym przykładzie twierdziłem, że algorytm BMH powinien być lepszy niż `binstr`, co jednak okazało się nieprawdą i zostało obalone za pomocą prostego testu. Gdybym nie przeprowadził tego testu, używałbym gorszego algorytmu, nawet nie wiedząc o tym. Dzięki dostępnym metrykom jestem w stanie rozpocząć usprawnianie implementacji lub po prostu porzucić ją i poszukać innej.

Zadania dodatkowe

- Zobacz czy potrafisz zwiększyć szybkość działania funkcji `Scan_find()`. Dlaczego implementacja przedstawiona w ćwiczeniu jest wolna?
- Spróbuj zmienić czas trwania skanowania i zobacz, jakie to będzie miało przełożenie na otrzymany wynik. Jaki wpływ ma czas trwania testu ma na czas działania funkcji `scan()`? Co możesz powiedzieć o otrzymanych wynikach?
- Zmodyfikuj test jednostkowy, aby najpierw uruchamiał każdą funkcję na krótką chwilę na początku testu w celu przeprowadzenia rozgrzewki, a dopiero później rozpoczynał pomiar czasu. Czy tego rodzaju zmiana ma wpływ na czas trwania testu? Czy ta zmiana wpłynęła na liczbę operacji wykonywanych w ciągu sekundy?
- Wprowadź w teście jednostkowym czynnik losowości dotyczący wyszukiwanych ciągów tekstowych, a następnie zmierz wydajność. Jednym z możliwych podejść jest użycie funkcji `bsplit()` z *bstrlib.c* w celu podziału `IN_STR` w miejscu występowania spacji. Następnie wykorzystaj strukturę `bstrList`, aby uzyskać dostęp do każdego ciągu tekstowego zwróconego przez wymienioną funkcję. Tym samym dowiesz się również, w jaki sposób można używać operacji `bstrList` do przetwarzania ciągu tekstowego.
- Spróbuj przeprowadzić testy w różnej kolejności. Sprawdź, czy otrzymasz wtedy inne wyniki.

ĆWICZENIE 40

Binarne drzewo poszukiwań

Drzewo binarne to najprostsza struktura danych bazująca na drzewie. Mimo że została zastąpiona w wielu językach programowania przez `Hashmap`, nadal okazuje się niezwykle użyteczna w niektórych aplikacjach. Warianty drzewa binarnego mają wiele zastosowań, takich jak indeks w bazie danych, struktura algorytmu wyszukiwania, a nawet grafika.

W tym ćwiczeniu binarne drzewo przeszukiwań będę określał mianem `BSTree`. Najlepszym sposobem na jego opisanie jest przygotowanie innego magazynu w stylu `Hashmap`, przechowującego dane w postaci klucz-wartość. Różnica polega na tym, że zamiast generować wartość hash w celu wyszukania położenia, `BSTree` porównuje klucz do węzłów drzewa, a następnie przechodzi przez drzewo w poszukiwaniu najlepszego miejsca na umieszczenie danych, opierając decyzję na wyniku porównania z innymi węzłami.

Zanim dokładnie wyjaśnię sposób działania powyższego rozwiązania, zaczniemy od przygotowania pliku nagłówkowego *bstree.h*. Dzięki temu zobaczysz użyte tutaj struktury danych. Później wyjaśnię, jak zostało zbudowane drzewo binarne.

Plik *bstree.h*:

```
#ifndef _lcthw_BSTree_h
#define _lcthw_BSTree_h

typedef int (*BSTree_compare) (void *a, void *b);

typedef struct BSTreeNode {
    void *key;
    void *data;

    struct BSTreeNode *left;
    struct BSTreeNode *right;
    struct BSTreeNode *parent;
} BSTreeNode;

typedef struct BSTree {
    int count;
    BSTree_compare compareP
    BSTreeNode *root;
} BSTree;

typedef int (*BSTree_traverse_cb) (BSTreeNode * node);

BSTree *BSTree_create(BSTree_compare compare);
void BSTree_destroy(BSTree * map);

int BSTree_set(BSTree * map, void *key, void *data);
void *BSTree_get(BSTree * map, void *key);
```

```
int BSTree_traverse(BSTree * map, BSTree_traverse_cb traverse_cb);

void *BSTree_delete(BSTree * map, void *key);

#endif
```

W powyższym pliku nagłówkowym zastosowałem ten sam wzorzec, którego używam praktycznie za każdym razem. Mamy więc kontener podstawowy o nazwie `BSTree` wraz z węzłami `BSTreeNode` tworzącymi rzeczywistą zawartość. Nie znudziłeś się jeszcze? Dobrze, nie ma absolutnie żadnego powodu, aby tego rodzaju strukturę tworzyć w skomplikowany sposób.

Najważniejszy jest sposób konfiguracji elementów `BSTreeNode` oraz wykonywania przez nie operacji `set()`, `get()` i `delete()`. Zacznę od omówienia operacji `get()`, ponieważ jest ona najłatwiejsza. Będę w tym miejscu udawał, że przeprowadzam ją ręcznie względem struktury danych.

- Biorę szukany klucz i zaczynam przeglądać strukturę od początku, czyli od węzła głównego. Pierwszym zadaniem jest porównanie wziętego klucza z kluczem węzła.
- Jeżeli klucz ma wartość mniejszą niż `node.key`, to poruszam się w dół drzewa za pomocą wskaźnika `left`.
- Jeżeli klucz ma wartość większą niż `node.key`, to poruszam się w dół drzewa za pomocą wskaźnika `right`.
- Kroki 2. i 3. powtarzam aż do znalezienia dopasowanego `node.key` lub dotarcia do węzła, który nie pozwala na przejście w prawo bądź lewo. W pierwszym przypadku zwracam `node.data`, natomiast w drugim `NULL`.

Powyżej przedstawiłem działanie operacji `get()`, więc teraz przechodzimy do funkcji `set()`. W sumie działanie jest bardzo podobne, z wyjątkiem tego, że szukamy miejsca, w którym umieścimy nowy węzeł.

- Jeżeli nie istnieje węzeł `BSTree.root`, to go tworzymy, i na tym koniec pracy. Tak powstaje pierwszy węzeł.
- Następnie porównujemy posiadany klucz z `node.key`, zaczynając od węzła głównego.
- Jeżeli klucz jest mniejszy lub równy `node.key`, to należy przejść w lewo. Natomiast jeśli klucz jest większy i nierówny `node.key`, to należy przejść w prawo.
- Krok 3. jest powtarzany aż do dotarcia do węzła, dla którego nie istnieje węzeł po prawej lub lewej stronie. Jednak wówczas konieczne jest ustalenie kierunku, w którym trzeba będzie pójść.
- W takim przypadku ustalamy stronę (lewa lub prawa) utworzenia nowego węzła dla posiadanego klucza i danych, a następnie węzeł, z którego przyszliśmy, stanie się węzłem nadrzędnym dla nowo utworzonego. Węzeł nadrzędny będzie używany w trakcie operacji `delete()`.

Przedstawione operacje mają sens, biorąc pod uwagę sposób działania drzewa binarnego. Jeżeli odszukanie węzła wymaga przejścia w lewo lub prawo w zależności od wyniku porównania klucza, to zdefiniowanie nowego węzła działa tak samo, o ile można utworzyć nowy węzeł po lewej lub prawej stronie.

Poświęć nieco czasu na narysowanie kilku drzew na kartce papieru, a następnie przeanalizuj operacje tworzenia i pobierania węzłów, abyś dokładnie poznał ten mechanizm. Dopiero wtedy będziesz gotowy na przejście do kodu przedstawiającego implementację binarnego drzewa poszukiwań, co pozwoli mi na wyjaśnienie operacji `delete()`. Usuwanie węzłów w drzewie jest *bardzo* problematyczne, więc najlepiej to wyjaśnić podczas omawiania wiersz po wierszu kodu odpowiedzialnego za przeprowadzenie tej operacji.

Plik *bstree.c*:

```
#include <lcthw/dbg.h>
#include <lcthw/bstree.h>
#include <stdlib.h>
#include <lcthw/bstrlib.h>

static int default_compare(void *a, void *b)
{
    return bstrcmp((bstring) a, (bstring) b);
}

BSTree *BSTree_create(BSTree_compare compare)
{
    BSTree *map = calloc(1, sizeof(BSTree));
    check_mem(map);

    map->compare = compare == NULL ? default_compare : compare;

    return map;

error:
    if (map) {
        BSTree_destroy(map);
    }
    return NULL;
}

static int BSTree_destroy_cb(BSTreeNode * node)
{
    free(node);
    return 0;
}

void BSTree_destroy(BSTree * map)
{
    if (map) {
        BSTree_traverse(map, BSTree_destroy_cb);
        free(map);
    }
}

static inline BSTreeNode *BSTreeNode_create(BSTreeNode * parent,
        void *key, void *data)
{
```

```
44      BSTreeNode *node = calloc(1, sizeof(BSTreeNode));
45      check_mem(node);
46
47      node->key = key;
48      node->data = data;
49      node->parent = parent;
50      return node;
51
52  error:
53      return NULL;
54  }
55
56  static inline void BSTree_setnode(BSTree * map, BSTreeNode * node,
57          void *key, void *data)
58  {
59      int cmp = map->compare(node->key, key);
60
61      if (cmp <= 0) {
62          if (node->left) {
63              BSTree_setnode(map, node->left, key, data);
64          } else {
65              node->left = BSTreeNode_create(node, key, data);
66          }
67      } else {
68          if (node->right) {
69              BSTree_setnode(map, node->right, key, data);
70          } else {
71              node->right = BSTreeNode_create(node, key, data);
72          }
73      }
74  }
75
76  int BSTree_set(PBSTree * map, void *key, void *data)
77  {
78      if (map->root == NULL) {
79          // To jest pierwszy węzeł, więc go tworzymy i kończymy działanie.
80          map->root = BSTreeNode_create(NULL, key, data);
81          check_mem(map->root);
82      } else {
83          BSTree_setnode(map, map->root, key, data);
84      }
85
86      return 0;
87  error:
88      return -1;
89  }
90
91  static inline BSTreeNode *BSTree_getnode(BSTree * map,
92          BSTreeNode * node, void *key)
93  {
94      int cmp = map->compare(node->key, key);
95
96      if (cmp == 0) {
```

```
 97          return node;
 98      } else if (cmp < 0) {
 99          if (node->left) {
100              return BSTree_getnode(map, node->left, key);
101          } else {
102              return NULL;
103          }
104      } else {
105          if (node->right) {
106              return BSTree_getnode(map, node->right, key);
107          } else {
108              return NULL;
109          }
110      }
111  }
112
113  void *BSTree_get(BSTree * map, void *key)
114  {
115      if (map->root == NULL) {
116          return NULL;
117      } else {
118          BSTreeNode *node = BSTree_getnode(map, map->root, key);
119          return node == NULL ? NULL : node->data;
120      }
121  }
122
123  static inline int BSTree_traverse_nodes(BSTreeNode * node,
124          BSTree_traverse_cb traverse_cb)
125  {
126      int rc = 0;
127
128      if (node->left) {
129          rc = BSTree_traverse_nodes(node->left, traverse_cb);
130          if (rc != P
131              return rc;
132      }
133
134      if (node->right) {
135          rc = BSTree_traverse_nodes(node->right, traverse_cb);
136          if (rc != 0)
137              return rc;
138      }
139
140      return traverse_cb(node);
141  }
142
143  int BSTree_traverse(BSTree * map, BSTree_traverse_cb traverse_cb)
144  {
145      if (map->root) {
146          return BSTree_traverse_nodes(map->root, traverse_cb);
147      }
148
149      return 0;
```

```
150 }
151
152 static inline BSTreeNode *BSTree_find_min(BSTreeNode * node)
153 {
154     while (node->left) {
155         node = node->left;
156     }
157
158     return node;
159 }
160
161 static inline void BSTree_replace_node_in_parent(BSTree * map,
162         BSTreeNode * node,
163         BSTreeNode * new_value)
164 {
165     if (node->parent) {
166         if (node == node->parent->left) {
167             node->parent->left = new_value;
168         } else {
169             node->parent->right = new_value;
170         }
171     } else {
172         // To jest węzeł główny, więc trzeba wprowadzić zmianę.
173         map->root = new_value;
174     }
175
176     if (new_value) {
177         new_value->parent = node->parent;
178     }
179 }
180
181 static inline void BSTree_swap(BSTreeNode * a, BSTreeNode * b)
182 {
183     void *temp = NULL;
184     temp = b->key;
185     b->key = a->key;
186     a->key = temp;
187     temp = b->data;
188     b->data = a->data;
189     a->data = temp;
190 }
191
192 static inline BSTreeNode *BSTree_node_delete(BSTree * map,
193         BSTreeNode * node,
194         void *key)
195 {
196     int cmp = map->compare(node->key, key);
197
198     if (cmp < 0) {
199         if (node->left) {
200             return BSTree_node_delete(map, node->left, key);
201         } else {
202             // Nie znaleziono.
```

```
203             return NULL;
204         }
205     } else if (cmp > 0) {
206         if (node->right) {
207             return BSTree_node_delete(map, node->right, key);
208         } else {
209             // Nie znaleziono.
210             return NULL;
211         }
212     } else {
213         if (node->left && node->right) {
214             // Zastąpienie tego węzła najmniejszym węzłem, który będzie większy
                // od bieżącego.
215             BSTreeNode *successor = BSTree_find_min(node->right);
216             BSTree_swap(successor, node);
217
218             // Pozostajemy ze starym sukcesorem, który prawdopodobnie ma węzeł
                // potomny po prawej stronie.
219             // Dokonujemy więc zastąpienia z użyciem tego węzła potomnego
                // po prawej stronie.
220             BSTree_replace_node_in_parent(map, successor,
221                 successor->right);
222
223             // Węzły zostały wreszcie zastąpione, więc zwracamy sukcesora zamiast
                // bieżącego węzła.
224             return successor;
225         } else if (node->left) {
226             BSTree_replace_node_in_parent(map, node, node->left);
227         } else if (node->right) {
228             BSTree_replace_node_in_parent(map, node, node->right);
229         } else {
230             BSTree_replace_node_in_parent(map, node, NULL);
231         }
232
233         return node;
234     }
235 }
236
237 void *BSTree_delete(BSTree * map, void *key)
238 {
239     void *data = NULL;
240
241     if (map->root) {
242         BSTreeNode *node = BSTree_node_delete(map, map->root, key);
243
244         if (node) {
245             data = node->data;
246             free(node);
247         }
248     }
249
250     return data;
251 }
```

Zanim przejdziemy do omówienia działania funkcji `BSTree_delete()`, najpierw chciałbym wyjaśnić wzorzec przeprowadzania rekurencyjnych wywołań funkcji w rozsądny sposób. Przekonasz się, że wiele struktur danych opartych na drzewie pozwala na łatwy zapis, o ile będzie użyta rekurencja zdefiniowana za pomocą pojedynczej funkcji rekurencji. Część problemu polega jednak na tym, że trzeba skonfigurować pewne dane początkowe dla pierwszej operacji, a *następnie* rekurencyjnie zagłębić się w strukturę danych, co okazuje się niezwykle trudne, gdy ma się do dyspozycji tylko jedną funkcję.

Rozwiązaniem jest użycie dwóch funkcji. Pierwsza odpowiada za konfigurację struktury danych i początkowych warunków rekurencji, natomiast druga funkcja może wówczas wykonać rzeczywistą pracę. Spójrz na funkcję `BSTree_get()`, aby zobaczyć, co mam na myśli.

- Zdefiniowałem warunek początkowy, sprawdzający, czy `map->root` przyjmuje wartość `NULL`, a następnie zwracamy wartość `NULL` i nie przeprowadzamy rekurencji.
- Następnie definiujemy wywołanie do rzeczywistej rekurencji, która odbywa się w funkcji `BSTree_getnode()`. Zdefiniowałem warunek początkowy dla węzła głównego, aby rozpocząć operację z kluczem, a następnie ze strukturą `map`.
- W funkcji `BSTree_getnode()` mamy logikę odpowiedzialną za rzeczywistą rekurencję. Klucze są porównywane za pomocą wywołania `map->compare(node->key, key)`, a następnie w zależności od wyniku przechodzimy w prawo lub w lewo.
- Ponieważ to funkcja samopodobieństwa i nie musi zajmować się obsługą żadnych warunków początkowych (za to odpowiada `BSTree_get()`), więc możemy zastosować bardzo prostą strukturę. Po zakończeniu działania funkcji następuje powrót do komponentu wywołującego, a dane zwrotne są przekazywane funkcji `BSTree_`
  `↪get()`, generującej ostateczny wynik.
- Na koniec funkcja `BSTree_get()` zajmuje się obsługą pobrania elementu `node.data`, ale tylko wtedy, gdy wynik jest inny niż `NULL`.

Tego rodzaju struktura algorytmu rekurencji odpowiada sposobowi, na jaki tworzę struktury dla moich struktur danych rekurencyjnych. Przygotowuję początkową funkcję bazową odpowiedzialną za obsługę warunku początkowego i pewnych przypadków skrajnych, a następnie wywołuję funkcje ściśle rekurencyjną, która wykonuje rzeczywistą pracę. Porównaj to podejście z przygotowaniem struktury bazowej `BSTree` połączonej z rekurencyjnymi strukturami `BSTreeNode`, które wszystkie odwołują się do siebie nawzajem w drzewie. Zastosowanie omówionego wzorca ułatwia pracę z rekurencją oraz zachowanie prostoty.

Następnie spójrz na funkcje `BSTree_set()` i `BSTree_setnode()`, a dostrzeżesz dokładnie ten sam wzorzec. Za pomocą funkcji `BSTree_set()` przeprowadzam konfigurację warunków początkowych oraz przypadków skrajnych. Dość często spotykanym przypadkiem skrajnym jest brak węzła głównego, więc trzeba go wtedy utworzyć, aby móc rozpocząć operację.

Ten wzorzec będzie sprawdzał się w niemalże każdym algorytmie rekurencji, z którym będziesz musiał pracować. Oto podejście stosowane przeze mnie.

- Określenie zmiennych początkowych, jak się zmieniają i jakie są warunki zatrzymania na każdym kroku rekurencji.
- Utworzenie funkcji rekurencyjnej wywołującej samą siebie i mającej argumenty dla każdego warunku zatrzymania oraz zmienną początkową.

- Utworzenie funkcji konfiguracyjnej odpowiedzialnej za ustawienie warunków początkowych dla algorytmu oraz obsługę przypadków skrajnych, a następnie wywołanie przez nią funkcji rekurencyjnej.
- Na końcu przygotowujemy funkcję zwracającą ostateczny wynik i prawdopodobnie zmieniającą go, jeśli funkcja rekurencyjna nie potrafi obsłużyć ostatnich przypadków skrajnych.

Dochodzimy teraz do funkcji `BSTree_delete()` i `BSTree_node_delete()`. Zacznij od spojrzenia na kod funkcji `BSTree_delete()` i zwróć uwagę na jej funkcję konfiguracyjną. Pobiera ona wynikowe dane węzła i usuwa ten znaleziony węzeł. Sytuacja znacznie się komplikuje w funkcji `BSTree_node_delete()`, ponieważ aby usunąć węzeł w dowolnym miejscu drzewa, konieczne jest przeprowadzenie *rotacji* przenoszącej jego węzły potomne w górę, do węzła nadrzędnego. Poniżej przedstawiłem dokładne omówienie działania wymienionych funkcji.

bstree.c:196. Wykonanie funkcji porównania w celu ustalenia kierunku, w którym będziemy się poruszać.

bstree.c:198 – 204. To jest tradycyjna gałąź „mniejszy niż" używana podczas poruszania się w lewo. Obsługujemy sytuację, gdy nie można pójść w lewo, i zwracamy wartość `NULL`, oznaczającą „nie znaleziono". W ten sposób mamy obsłużony przypadek próby usunięcia węzła nieistniejącego w drzewie binarnym.

bstree.c:206 – 211. Kod podobny do powyższego, ale dla gałęzi odpowiedzialnej za przejście w prawo. Przeprowadzamy rekurencję w dół drzewa podobnie jak w pozostałych funkcjach. Jeżeli węzeł nie istnieje, wartością zwrotną będzie `NULL`.

bstree.c:212. W tym miejscu znajdujemy węzeł, ponieważ klucze są takie same (funkcja `compare()` zwraca wartość `0`).

bstree.c:213. Ten węzeł ma gałęzie w prawą i w lewą stronę, a więc mamy do czynienia z wysoce zagnieżdżonym drzewem.

bstree.c:215. W celu usunięcia drzewa należy zacząć od odszukania najmniejszego węzła, który będzie większy od bieżącego. Oznacza to wywołanie funkcji `BSTree_`
`↪find_min()` dla prawego węzła potomnego.

bstree.c:216. Po otrzymaniu szukanego węzła zastępujemy jego klucz i dane wartościami pochodzącymi z węzła bieżącego. Tym samym bierzemy węzeł znajdujący się niżej i umieszczamy jego zawartość w węźle bieżącym. Unikamy więc konieczności podjęcia próby przestawienia węzła za pomocą jego wskaźników.

bstree.c:220 – 221. Wartością `successor` będzie teraz martwa gałąź zawierająca wartości węzła bieżącego. Wprawdzie można ją po prostu usunąć, ale nadal istnieje prawdopodobieństwo, że ma wartość prawego węzła. To oznacza konieczność przeprowadzenia pojedynczej rotacji, aby ten prawy węzeł został przeniesiony w górę i całkowicie odłączony od węzła przeznaczonego do usunięcia.

bstree.c:224. Na tym etapie węzeł został usunięty z drzewa, jego wartości zostały zastąpione wartościami węzła bieżącego, a wszystkie węzły potomne uległy przesunięciu w górę do węzła nadrzędnego dla węzła, który został usunięty. Można więc zwrócić wartość `successor`, jakby była węzłem.

bstree.c:225. Wiemy, że w bieżącej gałęzi węzeł ma inny węzeł po lewej stronie, ale nie po prawej. Dlatego też chcemy go zastąpić jego węzłem potomnym znajdującym się po lewej stronie.

bstree.c:226. Ponownie używamy funkcji `BSTree_replace_node_in_parent()` w celu przeprowadzenia operacji zastąpienia i rotacji lewego węzła potomnego w górę.

bstree.c:227. Ta gałąź polecenia `if` oznacza, że mamy prawy węzeł potomny, ale nie lewy. Dlatego też chcemy przeprowadzić operację rotacji prawego węzła potomnego w górę.

bstree.c:228. Ponownie używamy funkcji `BSTree_replace_node_in_parent()` do przeprowadzenia rotacji, ale tym razem prawego węzła.

bstree.c:230. Na koniec jedyna rzecz, jaka pozostała do obsługi, to sytuacja, gdy znaleziony węzeł nie ma węzłów potomnych (po lewej lub prawej stronie). W takim przypadku po prostu zastępujemy ten węzeł wartością `NULL`, używając tej samej funkcji, jak we wszystkich pozostałych przypadkach.

bstree.c:233. Ostatecznie bieżący węzeł został usunięty z drzewa i zastąpiony elementem potomnym, który będzie pasował w tym drzewie. Informacje dotyczące usuniętego węzła są przekazywane komponentowi wywołującemu, aby mógł zająć się zwolnieniem zasobów wcześniej zajmowanych przez usunięty węzeł.

Operacja jest bardzo skomplikowana i szczerze mówiąc, w ogóle nie zabieram się do usuwania węzłów w niektórych strukturach danych opartych na drzewach. Traktuję je bardziej jako stałe elementy w tworzonym oprogramowaniu. Jeżeli mam potrzebę przeprowadzania wielu operacji wstawiania i usuwania danych, to zamiast drzewa binarnego używam `Hashmap`.

Teraz możemy przystąpić do przygotowania testów jednostkowych dla naszej struktury drzewa binarnego.

Plik *bstree_tests.c*:

```
#include "minunit.h"
#include <lcthw/bstree.h>
#include <assert.h>
#include <lcthw/bstrlib.h>
#include <stdlib.h>
#include <time.h>

BSTree *map = NULL;
static int traverse_called = 0;
struct tagbstring test1 = bsStatic("dane testowe 1");
struct tagbstring test2 = bsStatic("dane testowe 2");
struct tagbstring test3 = bsStatic("dane testowe 3");
struct tagbstring expect1 = bsStatic("WARTOŚĆ 1");
struct tagbstring expect2 = bsStatic("WARTOŚĆ 2");
struct tagbstring expect3 = bsStatic("WARTOŚĆ 3");

static int traverse_good_cb(BSTreeNode * node)
{
    debug("KLUCZ: %s", bdata((bstring) node->key));
```

```
20      traverse_called++;
21      return 0;
22  }
23
24  static int traverse_fail_cb(BSTreeNode * node)
25  {
26      debug("KLUCZ: %s", bdata((bstring) node->key));
27      traverse_called++;
28
29      if (traverse_called == 2) {
30          return 1;
31      } else {
32          return 0;
33      }
34  }
35
36  char *test_create()
37  {
38      map = BSTree_create(NULL);
39      mu_assert(map != NULL, "Nie udało się utworzyć mapy.");
40
41      return NULL;
42  }
43
44  char *test_destroy()
45  {
46      BSTree_destroy(map);
47
48      return NULL;
49  }
50
51  char *test_get_set()
52  {
53      int rc = BSTree_set(map, &test1, &expect1);
54      mu_assert(rc == 0, "Nie udało się ustawić &test1");
55      bstring result = BSTree_get(map, &test1);
56      mu_assert(result == &expect1, "Nieprawidłowa wartość dla test1.");
57
58      rc = BSTree_set(map, &test2, &expect2);
59      mu_assert(rc == 0, "Nie udało się ustawić test2");
60      result = BSTree_get(map, &test2);
61      mu_assert(result == &expect2, "Nieprawidłowa wartość dla test2.");
62
63      rc = BSTree_set(map, &test3, &expect3);
64      mu_assert(rc == 0, "Nie udało się ustawić test3");
65      result = BSTree_get(map, &test3);
66      mu_assert(result == &expect3, "Nieprawidłowa wartość dla test3.");
67
68      return NULL;
69  }
70
71  char *test_traverse()
72  {
```

```
 73     int rc = BSTree_traverse(map, traverse_good_cb);
 74     mu_assert(rc == 0, "Nie udało się przejść.");
 75     mu_assert(traverse_called == 3, "Nieprawidłowa liczba przejść.");
 76
 77     traverse_called = 0;
 78     rc = BSTree_traverse(map, traverse_fail_cb);
 79     mu_assert(rc == 1, "Nie udało się przejść.");
 80     mu_assert(traverse_called == 2, "Nieprawidłowa liczba przejść for fail.");
 81
 82     return NULL;
 83 }
 84
 85 char *test_delete()
 86 {
 87     bstring deleted = (bstring) BSTree_delete(map, &test1);
 88     mu_assert(deleted != NULL, "Otrzymano NULL podczas usuwania.");
 89     mu_assert(deleted == &expect1, "Powinno być test1.");
 90     bstring result = BSTree_get(map, &test1);
 91     mu_assert(result == NULL, "Powinien zostać usunięty.");
 92
 93     deleted = (bstring) BSTree_delete(map, &test1);
 94     mu_assert(deleted == NULL, "Powinno być NULL podczas usuwania.");
 95
 96     deleted = (bstring) BSTree_delete(map, &test2);
 97     mu_assert(deleted != NULL, "Otrzymano NULL podczas usuwania.");
 98     mu_assert(deleted == &expect2, "Powinno być test2.");
 99     result = BSTree_get(map, &test2);
100     mu_assert(result == NULL, "Powinien zostać usunięty.");
101
102     deleted = (bstring) BSTree_delete(map, &test3);
103     mu_assert(deleted != NULL, "Otrzymano NULL podczas usuwania.");
104     mu_assert(deleted == &expect3, "Powinno być test3.");
105     result = BSTree_get(map, &test3);
106     mu_assert(result == NULL, "Powinien zostać usunięty.");
107
108     // Sprawdzenie próby usunięcia nieistniejących elementów.
109     deleted = (bstring) BSTree_delete(map, &test3);
110     mu_assert(deleted == NULL, "Powinno być NULL.");
111
112     return NULL;
113 }
114
115 char *test_fuzzing()
116 {
117     BSTree *store = BSTree_create(NULL);
118     int i = 0;
119     int j = 0;
120     bstring numbers[100] = { NULL };
121     bstring data[100] = { NULL };
122     srand((unsigned int)time(NULL));
123
124     for (i = 0; i < 100; i++) {
125         int num = rand();
```

```
126         numbers[i] = bformat("%d", num);
127         data[i] = bformat("data %d", num);
128         BSTree_set(store, numbers[i], data[i]);
129     }
130
131     for (i = 0; i < 100; i++) {
132         bstring value = BSTree_delete(store, numbers[i]);
133         mu_assert(value == data[i],
134             "Nie udało się usunąć odpowiedniej liczby elementów.");
135
136         mu_assert(BSTree_delete(store, numbers[i]) == NULL,
137             "Nic nie powinno być.");
138
139         for (j = i + 1; j < 99 - i; j++) {
140             bstring value = BSTree_get(store, numbers[j]);
141             mu_assert(value == data[j],
142                 "Nie udało się pobrać odpowiedniej liczby elementów.");
143         }
144
145         bdestroy(value);
146         bdestroy(numbers[i]);
147     }
148
149     BSTree_destroy(store);
150
151     return NULL;
152 }
153
154 char *all_tests()
155 {
156     mu_suite_start();
157
158     mu_run_test(test_create);
159     mu_run_test(test_get_set);
160     mu_run_test(test_traverse);
161     mu_run_test(test_delete);
162     mu_run_test(test_destroy);
163     mu_run_test(test_fuzzing);
164
165     return NULL;
166 }
167
168 RUN_TESTS(all_tests);
```

Skieruj swoją uwagę na funkcję `test_fuzzing()`, która pokazuje interesującą technikę przeznaczoną do testowania skomplikowanych struktur danych. Bardzo trudno jest utworzyć zbiór kluczy pokrywających wszystkie gałęzie w `BSTree_node_delete()`, więc istnieje niebezpieczeństwo pominięcia pewnych przypadków skrajnych. Dlatego też lepsze rozwiązanie polega na utworzeniu funkcji `fuzz()` odpowiadającej za wykonanie wszystkich operacji, choć to będzie zrobione okropnie i charakteryzuje się dużą nieprzewidywalnością. W przedstawionym przypadku wstawiamy zbiór losowych kluczy, a następnie usuwamy je, próbując pobrać pozostałe po każdej operacji usunięcia.

Tego rodzaju podejście chroni nas przed przetestowaniem jedynie elementów, o których wiemy, że działają, i pominięciem tych nieznanych. Dzięki dostarczeniu strukturom danych losowo wygenerowanych śmieci odkrywasz kwestie, których w ogóle się nie spodziewałeś, i zyskujesz możliwość usunięcia wszelkich błędów.

Jak można usprawnić kod?

Jeszcze *nie* wprowadzaj wymienionych poniżej zmian. W kolejnym ćwiczeniu wykorzystamy ten test jednostkowych do poznania pewnych sztuczek dotyczących poprawy wydajności działania. Dopiero po zakończeniu lektury ćwiczenia 41. powróć tutaj i wprowadź przedstawione poniżej zmiany.

- Jak zwykle powinieneś przejść przez wszystkie strategie programowania defensywnego i dodać asercje dla warunków, które nie powinny wystąpić. Na przykład nie powinieneś otrzymać wartości `NULL` z funkcji rekurencji, więc przygotuj pod tym kątem odpowiednią asercję.
- Funkcja `traverse()` przechodzi przez drzewo w następującej kolejności: lewo, prawo i bieżący węzeł. Utwórz funkcję `traverse()`, która będzie przechodzić w odwrotnej kolejności.
- Dla każdego węzła przeprowadzane jest pełne porównanie ciągu tekstowego, ale można użyć funkcji generujących wartość hash dla struktur `Hashmap` i tym samym przyspieszyć operację. Można więc wygenerować wartość hash dla klucza i przechowywać ją w `BSTreeNode`. Następnie w każdej funkcji konfiguracyjnej można wcześniej generować wartość hash i przekazywać ją funkcji rekurencyjnej. Za pomocą tej wartości hash później znacznie szybciej przeprowadzamy porównanie każdego węzła, podobnie jak w przypadku struktur `Hashmap`.

Zadania dodatkowe

- Istnieje alternatywny sposób przygotowania omówionej w ćwiczeniu struktury danych bez użycia rekurencji. Artykuł w Wikipedii wskazuje alternatywne podejścia nieoparte na rekurencji, ale wykonujące takie samo zadanie. Dlaczego podejście alternatywne miałoby być lepsze lub gorsze?
- Poszukaj informacji o innych, choć podobnych strukturach drzewa. Mamy jeszcze drzewa AVL — nazywane tak od nazwisk rosyjskich matematyków Adelsona-Velskiego oraz Landisa (Gieorgija Adelsona-Wielskiego i Jewgienija Łandisa) — drzewa czerwono-czarne i pewne struktury niebazujące na drzewie, takie jak lista z przeskokami.

ĆWICZENIE 41

Projekt devpkg

Jesteś już gotowy, aby zająć się pracą nad nowym projektem o nazwie *devpkg*. W ramach tego projektu utworzymy oprogramowanie devpkg, które opracowałem specjalnie na potrzeby książki. Następnie zajmiesz się rozbudową projektu na kilka ważnych sposobów i usprawnisz kod, przede wszystkim przez utworzenie dla niego testów jednostkowych.

Dla tego ćwiczenia przygotowałem klip wideo oraz projekt w serwisie GitHub (*https://github.com/*), do którego możesz zajrzeć, gdy napotkasz trudności. Powinieneś podjąć próbę wykonania przedstawionego tutaj projektu, opierając się na podanych wskazówkach, ponieważ w taki właśnie sposób będziesz w przyszłości uczył się programować na podstawie książek. Do większości książek informatycznych nie są dołączane klipy wideo, więc wszelkie szczegóły związane z projektem trzeba ustalać na bazie dostępnego opisu.

Jeżeli podczas wykonywania projektu utkniesz, to obejrzyj klip wideo i zajrzyj do zamieszczonego przeze mnie projektu w serwisie GitHub, a następnie porównaj swój kod z moim.

Co to jest devpkg?

Devpkg to prosty program w języku C, przeznaczony do instalacji innego oprogramowania. Opracowałem go specjalnie na potrzeby książki, aby pokazać, jak przedstawia się struktura rzeczywistego oprogramowania oraz jak można wielokrotnie używać biblioteki utworzone przez innych. W projekcie wykorzystałem bibliotekę **APR** (ang. *Apache Portable Runtime*), oferującą mnóstwo użytecznych funkcji C działających na wielu platformach, w tym także Windows. Poza tym program pobiera kod z internetu (lub plików lokalnych), a następnie wykonuje zwykłe polecenia `./configure`, `make` i `make install`, jak ma to miejsce podczas instalacji innych programów.

Celem w tym ćwiczeniu jest utworzenie devpkg zupełnie od początku, wykonanie przedstawionych *wyzwań* oraz dokładne poznanie mechanizmu działania devpkg na podstawie jego kodu źródłowego.

Co chcemy zbudować?

Chcemy zbudować narzędzie oferujące wymienione poniżej polecenia:

- **`devpkg -S`**. Przygotowanie nowej instalacji w komputerze.
- **`devpkg -I`**. Instalacja oprogramowania z podanego adresu URL.
- **`devpkg -L`**. Wyświetlenie całego oprogramowania, które zostało zainstalowane.
- **`devpkg -F`**. Pobranie kodu źródłowego w celu ręcznej kompilacji oprogramowania.
- **`devpkg -B`**. Kompilacja kodu źródłowego i instalacja oprogramowania, nawet jeśli jest już zainstalowane.

Chcemy, aby program devpkg mógł pobrać dowolny adres URL, określić rodzaj projektu znajdującego się pod podanym adresem, pobrać kod źródłowy, zainstalować i zarejestrować to pobrane oprogramowanie. Dobrze będzie przetworzyć także prostą listę zależności, aby można było zainstalować całe oprogramowanie, jakie może być wymagane przez dany projekt.

Projekt

Aby osiągnąć założone cele, program devpkg ma stosunkowo prosty projekt.

Użycie poleceń zewnętrznych. Większość zadań wykonamy za pomocą poleceń zewnętrznych, takich jak `curl`, `git` i `tar`. Dzięki temu zmniejsza się ilość kodu niezbędnego programowi devpkg do wykonania zadań.

Plik prostej bazy danych. Wprawdzie dość łatwo można by ten aspekt znacznie skomplikować, ale na początku prac nad projektem wystarczy nam umieszczony w */usr/local/.devpkg/db* prosty plik bazy danych przeznaczonej do monitorowania zainstalowanego oprogramowania.

Zawsze używamy katalogu */usr/local*. Także w tym przypadku można by zastosować znacznie bardziej złożone podejście, ale na obecnym etapie przyjmujemy założenie, że zawsze będziemy używać katalogu */usr/local*. To jest standardowy katalog przeznaczony do instalacji większości oprogramowania w systemie UNIX.

`./configure`, `make`, `make install`. Przyjmujemy założenie, że większość oprogramowania można zainstalować za pomocą poleceń `./configure`, `make` i `make install`, przy czym `./configure` być może jest opcjonalne. Jeżeli oprogramowanie wymaga innej procedury instalacji, dostępne są pewne opcje pozwalające na modyfikację poleceń. Gdy podejście standardowe wystarcza, devpkg po prostu działa.

Użytkownik może mieć uprawnienia *root*. Przyjmujemy założenie, że użytkownik może otrzymać uprawnienia *root* dzięki zastosowaniu mechanizmu `sudo`. Jednak chcemy, aby te uprawnienia otrzymał dopiero na końcu operacji.

Na początku program zachowuje niewielkie rozmiary i działa wystarczająco sprawnie. Po przygotowaniu tej wersji będziemy mogli przystąpić do jego późniejszej modyfikacji.

Biblioteki Apache Portable Runtime

Jedną z rzeczy, którą trzeba będzie zrobić, jest wykorzystanie bibliotek APR w celu uzyskania dobrego zestawu procedur przenośnych, które będą potrzebne do pracy budowanego programu. Wprawdzie biblioteki APR nie są niezbędne i prawdopodobnie można by przygotować program bez nich, ale takie podejście wymagałoby utworzenia większej ilości kodu. Ponadto zmuszam Cię do użycia bibliotek APR, abyś mógł się przyzwyczaić do dołączania i stosowania bibliotek opracowanych przez innych programistów. Warto również dodać, że biblioteki APR działają też w systemie *Windows*, więc nabyte umiejętności będzie można wykorzystać podczas pracy na innej platformie.

Powinieneś pobrać biblioteki *apr-1.5.2* i *apr-util-1.5.4*, a także przejrzeć dokumentację dostępną na witrynie głównej APR pod adresem *http://apr.apache.org/*.

Poniżej przedstawiłem skrypt powłoki odpowiedzialny za instalację niezbędnych komponentów. Zawartość skryptu powinieneś wprowadzić ręcznie do pliku, a następnie uruchomić go, aby bez żadnych błędów zainstalować wymagane biblioteki APR.

Sesja dla ćwiczenia 41.:

```
set -e

# Przejście w bezpieczne miejsce.
cd /tmp

# Pobranie kodu źródłowego bazowej biblioteki APR 1.5.2.
curl -L -O http://archive.apache.org/dist/apr/apr-1.5.2.tar.gz

# Wyodrębnienie plików i przejście do katalogu kodu źródłowego.
tar -xzvf apr-1.5.2.tar.gz
cd apr-1.5.2

# Kompilacja: ./configure, make, make install.
./configure
make
sudo make install

# Wyzerowanie i operacje porządkujące.
cd /tmp
rm -rf apr-1.5.2 apr-1.5.2.tar.gz

# Powtórzenie operacji dla biblioteki apr-util.
curl -L -O http://archive.apache.org/dist/apr/apr-util-1.5.4.tar.gz

# Wyodrębnienie plików.
tar -xzvf apr-util-1.5.4.tar.gz
cd apr-util-1.5.4

# Kompilacja: configure, make, make install.
./configure --with-apr=/usr/local/apr
# Potrzebny jest parametr dodatkowy na etapie konfiguracji, ponieważ w przeciwnym
# razie biblioteka apr-util nie potrafi znaleźć bazowej biblioteki apr. Nie wiadomo
# dlaczego...


make
sudo make install

# Operacje porządkujące.
cd /tmp
rm -rf apr-util-1.5.4* apr-1.5.2*
```

Kazałem Ci ręcznie wprowadzić zawartość powyższego skryptu, ponieważ w zasadzie wykonuje on zadania, do których realizacji budujemy devpkg, ale z dodatkowymi opcjami i możliwościami w zakresie sprawdzania. Tak naprawdę wszystko mógłbyś zrobić bezpośrednio w powłoce, z wykorzystaniem mniejszej ilości kodu, ale wówczas to nie byłby zbyt dobry program do umieszczenia w książce dotyczącej programowania w języku C.

Po prostu uruchom powyższy skrypt i poprawiaj go do chwili, aż będzie działał prawidłowo. Wtedy otrzymasz biblioteki niezbędne do ukończenia pracy nad projektem.

Przygotowanie projektu

Aby rozpocząć pracę, trzeba przygotować kilka prostych plików projektu. Poniżej przedstawiłem polecenia, które zwykle wydaję w celu utworzenia nowego projektu.

Sesja dla ćwiczenia 41.:

```
$ mkdir devpkg
$ cd devpkg
$ touch README Makefile
```

Pozostałe zależności

Powinieneś mieć już biblioteki *apr-1.5.2* i *apr-util-1.5.4*, więc pozostało dołączenie kilku dodatkowych plików używanych w charakterze podstawowych zależności:

- *dbg.h* z ćwiczenia 20.
- *bstrlib.h* i *bstrlib.c* z witryny *http://bstring.sourceforge.net/*. Pobierz archiwum w formacie ZIP, rozpakuj je i skopiuj dwa wymienione pliki.
- Wydaj polecenie `make bstrilib.o`; jeśli nie zadziała, zapoznaj się z przedstawionymi poniżej informacjami, jak można naprawić `bstring`.

OSTRZEŻENIE Na niektórych platformach próba kompilacji pliku *bstring.c* powoduje wygenerowanie błędu podobnego do poniższego:

```
bstrlib.c:2762: error: expected declaration\
specifiers or '...' before numeric constant
```

Wynika to z dodania przez autora niewłaściwego polecenia `define`, które nie zawsze działa. Dlatego też musisz zmienić wiersz 2759 pliku i zastąpić polecenie `#idfef __GNUC__` poniższym:

```
#if defined(__GNUC__) && !defined(__APPLE__)
```

a kompilacja zakończy się powodzeniem w systemie OS X.

Po zakończeniu tego etapu powinieneś mieć pliki *Makefile*, *README*, *dbh.h*, *bstrlib.h* i *bstrlib.c* gotowe do dalszej pracy nad projektem.

Plik Makefile

Dobrym miejscem na rozpoczęcie pracy jest plik *Makefile*, ponieważ określa on sposób kompilacji projektu oraz to, jakie będziemy tworzyć pliki źródłowe.

Plik *Makefile*:

```
PREFIX?=/usr/local
CFLAGS=-g -Wall -I${PREFIX}/apr/include/apr-1
CFLAGS+=-I${PREFIX}/apr/include/apr-util-1
LDFLAGS=-L${PREFIX}/apr/lib -lapr-1 -pthread -laprutil-1

all: devpkg

devpkg: bstrlib.o db.o shell.o commands.o

install: all
    install -d $(DESTDIR)/$(PREFIX)/bin/
    install devpkg $(DESTDIR)/$(PREFIX)/bin/

clean:
    rm -f *.o
    rm -f devpkg
    rm -rf *.dSYM
```

W powyższym pliku nie znajdziesz nic nadzwyczajnego, z czym byś się już wcześniej nie spotkał. Jednym wyjątkiem może być dziwna składnia `?=`, która oznacza: „Ustaw `PREFIX` dokładnie, jak podano, chyba że ta zmienna została już zdefiniowana".

OSTRZEŻENIE Jeżeli używasz jednej z najnowszych wersji Ubuntu i pojawią się komunikaty o błędach dotyczące `apr_off_t` lub `off64_t`, to dodaj `-D_LARGE_FILE64_SOURCE=1` do opcji `CFLAGS`. Kolejną zmianą do wprowadzenia może być konieczność dodania */usr/local/apr/lib* do pliku w katalogu */etc/ld.conf.so.d/*, a następnie wydanie polecenia `ldconfig`, aby zostały wybrane odpowiednie biblioteki.

Pliki kodu źródłowego

Na podstawie pliku *Makefile* widzimy, że program devpkg ma pięć zależności:

bstrlib.o. Biblioteka skompilowana na podstawie plików *bstrlib.c* i *bstrlib.h*; zależność tę mamy już spełnioną.

db.o. Biblioteka skompilowana na podstawie plików *db.c* i *db.h*, będzie zawierała kod dla procedur obsługi naszej małej bazy danych.

shell.o. Biblioteka skompilowana na podstawie plików *shell.c* i *shell.h*, będzie zawierała kilka funkcji ułatwiających wykonywanie innych poleceń, takich jak `curl`.

commands.o. Biblioteka skompilowana na podstawie plików *commands.c* i *commands.h*, będzie zawierała wszystkie polecenia, dzięki którym program devpkg jest tak użyteczny.

devpkg. To nie jest wyraźnie wymieniona zależność, choć `devpkg` jest celem (`TARGET`) w pliku *Makefile*. Kod źródłowy znajduje się w pliku *devpkg.c* i zawiera funkcję `main()` przeznaczoną do obsługi całego programu.

Twoim zadaniem jest teraz utworzenie poszczególnych plików, wprowadzenie przeznaczonego dla nich kodu i zapewnienie prawidłowego ich działania.

OSTRZEŻENIE Być może czytasz ten opis i myślisz sobie: „Jejku, jakim sprytnym człowiekiem jest ten Zed, że tak po prostu usiadł do komputera i przygotował te pliki. Ja chyba nigdy bym tego nie potrafił". Muszę Ci w tym miejscu powiedzieć, że nie użyłem żadnych nadzwyczajnych umiejętności do opracowania programu devpkg w jego obecnej postaci. Mój sposób działania przedstawia się następująco:

- Przygotowuję niewielki plik *README*, aby dokładnie wiedzieć, co chcę osiągnąć.
- Przygotowuję prosty skrypt powłoki (podobny do przedstawionego wcześniej w ćwiczeniu) w celu ustalenia wszystkich niezbędnych komponentów programu.
- Tworzę jeden plik *.c*, a następnie eksperymentuję z nim przez kilka dni, wypróbowując różne idee i koncepcje.
- Gdy mam działające rozwiązanie i zakończę proces debugowania, przystępuję do podziału jednego dużego pliku na wymienione cztery mniejsze.
- Po przygotowaniu plików zmieniam nazwy funkcji i struktur danych, dopracowuję je, aby stały się znacznie bardziej logiczne i eleganckie.
- Kiedy po powyższych zmianach mam *dokładnie takie samo działające rozwiązanie jak wcześniej*, ale oparte na nowej strukturze, wówczas zaczynam dodawać nowe funkcje, takie jak opcje `-F` i `-B` w omawianym programie.

Przedstawiony w książce materiał poznajesz w kolejności, w jakiej masz się go nauczyć. Nie powinieneś jednak uważać, że to jest kolejność, którą zawsze stosuję podczas tworzenia oprogramowania. Czasami doskonale znam temat i stosuję nieco dłuższą fazę planowania. Z kolei w innych przypadkach po prostu zaczynam eksperymentować z ideą i sprawdzam, co z tego wyniknie. Jeszcze innym razem tworzę coś, usuwam to i planuję coś lepszego. Wszystko zależy od tego, co podpowiada mi doświadczenie lub gdzie zabiera mnie inspiracja.

Jeżeli natrafisz na rzekomego eksperta próbującego przekonać Cię o istnieniu tylko jednego właściwego sposobu rozwiązania problemu programistycznego, to pamiętaj, że on Cię po prostu okłamuje. Najlepiej będzie, jeśli postarasz się stosować różne podejścia i taktyki.

Funkcje bazy danych

Musi istnieć sposób na zarejestrowanie adresu URL, z którego pochodzi zainstalowane oprogramowanie, na wyświetlenie tego adresu URL i sprawdzenie, czy dane oprogramowanie na pewno zostało już wcześniej zainstalowane, co pozwala na jego pominięcie w trakcie kolejnej operacji sprawdzania i instalacji. W omawianym programie do tych celów wykorzystamy bazę danych w postaci pliku jednorodnego, a także bibliotekę *bstrlib.h*.

Zaczynamy od utworzenia pliku nagłówkowego *db.h*. Dzięki temu możesz zobaczyć, co trzeba będzie zaimplementować.

Plik *db.h*

```
#ifndef _db_h
#define _db_h

#define DB_FILE "/usr/local/.devpkg/db"
#define DB_DIR "/usr/local/.devpkg"

int DB_init();
int DB_list();
int DB_update(const char *url);
int DB_find(const char *url);

#endif
```

Następnie implementujemy w pliku *db.c* niezbędne nam funkcje, używając polecenia `make`, aby przeprowadzić czystą kompilację.

Plik *db.c*:

```
 1 #include <unistd.h>
 2 #include <apr_errno.h>
 3 #include <apr_file_io.h>
 4
 5 #include "db.h"
 6 #include "bstrlib.h"
 7 #include "dbg.h"
 8
 9 static FILE *DB_open(const char *path, const char *mode)
10 {
11     return fopen(path, mode);
12 }
13
14 static void DB_close(FILE * db)
15 {
16     fclose(db);
17 }
18
19 static bstring DB_load()
20 {
21     FILE *db = NULL;
```

```
22      bstring data = NULL;
23
24      db = DB_open(DB_FILE, "r");
25      check(db, "Nie udało się otworzyć bazy danych: %s", DB_FILE);
26
27      data = bread((bNread) fread, db);
28      check(data, "Nie udało się odczytać pliku bazy danych: %s", DB_FILE);
29
30      DB_close(db);
31      return data;
32
33  error:
34      if (db)
35          DB_close(db);
36      if (data)
37          bdestroy(data);
38      return NULL;
39  }
40
41  int DB_update(const char *url)
42  {
43      if (DB_find(url)) {
44          log_info("Oprogramowanie jest już zainstalowane: %s", url);
45  }
46
47      FILE *db = DB_open(DB_FILE, "a+");
48      check(db, "Nie udało się otworzyć pliku bazy danych: %s", DB_FILE);
49
50      bstring line = bfromcstr(url);
51      bconchar(line, '\n');
52      int rc = fwrite(line->data, blength(line), 1, db);
53      check(rc == 1, "Nie udało się dodać danych do pliku bazy danych.");
54
55      return 0;
56  error:
57      if (db)
58          DB_close(db);
59      return -1;
60  }
61
62  int DB_find(const char *url)
63  {
64      bstring data = NULL;
65      bstring line = bfromcstr(url);
66      int res = -1;
67
68      data = DB_load();
69      check(data, "Nie udało się wczytać: %s", DB_FILE);
70
71      if (binstr(data, 0, line) == BSTR_ERR) {
72          res = 0;
73      } else {
74          res = 1;
```

```
    }

error: // Celowe przejście.
    if (data)
        bdestroy(data);
    if (line)
        bdestroy(line);

    return res;
}

int DB_init()
{
    apr_pool_t *p = NULL;
    apr_pool_initialize();
    apr_pool_create(&p, NULL);

    if (access(DB_DIR, W_OK | X_OK) == -1) {
        apr_status_t rc = apr_dir_make_recursive(DB_DIR,
                APR_UREAD | APR_UWRITE
                | APR_UEXECUTE |
                APR_GREAD | APR_GWRITE
                | APR_GEXECUTE, p);
        check(rc == APR_SUCCESS, "Nie udało się utworzyć katalogu bazy danych: %s",
              DB_DIR);
    }

    if (access(DB_FILE, W_OK) == -1) {
        FILE *db = DB_open(DB_FILE, "w");
        check(db, "Nie można otworzyć bazy danych: %s", DB_FILE);
        DB_close(db);
    }

    apr_pool_destroy(p);
    return 0;

error:
    apr_pool_destroy(p);
    return -1;
}

int DB_list()
{
    bstring data = DB_load();
    check(data, "Nie udało się wczytać: %s", DB_FILE);

    printf("%s", bdata(data));
    bdestroy(data);
    return 0;

error:
    return -1;
}
```

Wyzwanie 1. Przegląd kodu

Zanim będziemy kontynuować pracę, dokładnie odczytaj każdy wiersz kodu źródłowego tworzonych plików i upewnij się, że wprowadziłeś je *dokładnie*, jak pokazałem w książce. Odczytaj także wiersz po wierszu od końca, aby nabrać w tym praktyki. Ponadto prześledź każde wywołanie funkcji i upewnij się o użyciu wywołań `check()` do sprawdzenia danych wyjściowych funkcji. Wreszcie poszukaj informacji o *każdej* nieznanej Ci funkcji — w dokumentacji dostępne w witrynie APR lub w plikach kodu źródłowego *bstrlib.h* i *bstrlib.c*.

Funkcje powłoki

Kluczową decyzją projektową w programie devpkg jest użycie zewnętrznych narzędzi, takich jak `curl`, `tar` i `git`, do wykonywania większości zadań. Wprawdzie można znaleźć biblioteki pozwalające na wewnętrzne wykonanie tego rodzaju zadań, ale to bezcelowe, jeżeli potrzebujemy jedynie podstawowych funkcjonalności wymienionych narzędzi. Naprawdę nie jest wstydem wykonywanie w programie innych poleceń wbudowanych w systemie UNIX.

W omawianym programie do uruchamiania narzędzi wykorzystamy funkcje zdefiniowane w pliku *apr_thread_proc.h*, choć przygotujemy również pewnego rodzaju prosty system szablonów. Mianowicie użyjemy struktury `Shell` do przechowywania wszystkich informacji niezbędnych do uruchomienia danego narzędzia, przy czym pozostawimy miejsca na liście argumentów, co pozwoli nam na dostarczanie własnych wartości.

Spójrz na plik *shell.h*, aby poznać strukturę i używane polecenia. Jak możesz zobaczyć, za pomocą słowa kluczowego `extern` wskazujemy, że inne pliki mogą uzyskać dostęp do zmiennych zdefiniowanych w *shell.c*.

Plik *shell.h*:

```
#ifndef _shell_h
#define _shell_h

#define MAX_COMMAND_ARGS 100

#include <apr_thread_proc.h>

typedef struct Shell {
    const char *dir;
    const char *exe;

    apr_procattr_t *attr;
    apr_proc_t proc;
    apr_exit_why_e exit_why;
    int exit_code;

    const char *args[MAX_COMMAND_ARGSP
} Shell;

int Shell_run(apr_pool_t * p, Shell * cmd);
int Shell_exec(Shell cmd, ...);
```

```
extern Shell CLEANUP_SH;
extern Shell GIT_SH;
extern Shell TAR_SH;
extern Shell CURL_SH;
extern Shell CONFIGURE_SH;
extern Shell MAKE_SH;
extern Shell INSTALL_SH;

#endif
```

Upewnij się o wprowadzeniu kodu pliku *shell.h* dokładnie tak, jak przedstawiłem w książce, a także o zachowaniu takich samych nazw i liczby zmiennych `extern Shell`. Będą one wykorzystywane przez funkcje `Shell_run()` i `Shell_exec()` do wykonywania poleceń powłoki. W pliku *shell.c* zdefiniowałem obie wymienione funkcje i utworzyłem rzeczywiste zmienne.

Plik *shell.c*:

```
#include "shell.h"
#include "dbg.h"
#include <stdarg.h>

int Shell_exec(Shell template, ...)
{
    apr_pool_t *p = NULL;
    int rc = -1;
    apr_status_t rv = APR_SUCCESS;
    va_list argp;
    const char *key = NULL;
    const char *arg = NULL;
    int i = 0;

    rv = apr_pool_create(&p, NULL);
    check(rv == APR_SUCCESS, "Nie udało się utworzyć puli.");

    va_start(argp, template);

    for (key = va_arg(argp, const char *);
            key != NULL; key = va_arg(argp, const char *)) {
        arg = va_arg(argp, const char *);

        for (i = 0; template.args[i] != NULL; i++) {
            if (strcmp(template.args[i], key) == 0) {
                template.args[i] = arg;
                break; // Znaleziono.
            }
        }
    }

    rc = Shell_run(p, &template);
    apr_pool_destroy(p);
    va_end(argp);
    return rc;

```

```
37 error:
38     if (p) {
39         apr_pool_destroy(p);
40     }
41     return rc;
42 }
43
44 int Shell_run(apr_pool_t * p, Shell * cmd)
45 {
46     apr_procattr_t *attr;
47     apr_status_t rv;
48     apr_proc_t newproc;
49
50     rv = apr_procattr_create(&attr, p);
51     check(rv == APR_SUCCESS, "Nie udało się utworzyć atrybutu procedury.");
52
53     rv = apr_procattr_io_set(attr, APR_NO_PIPE, APR_NO_PIPE,
54             APR_NO_PIPE);
55     check(rv == APR_SUCCESS, "Nie udało się ustawić operacji wejścia-wyjścia
       ↪polecenia.");
56
57     rv = apr_procattr_dir_set(attr, cmd->dir);
58     check(rv == APR_SUCCESS, "Nie udało się uzyskać uprawnień roota dla %s",
       ↪cmd->dir);
59
60     rv = apr_procattr_cmdtype_set(attr, APR_PROGRAM_PATH);
61     check(rv == APR_SUCCESS, "Nie udało się ustawić typu polecenia.");
62
63     rv = apr_proc_create(&newproc, cmd->exe, cmd->args, NULL, attr, p);
64     check(rv == APR_SUCCESS, "Nie udało się wykonać polecenia.");
65
66     rv = apr_proc_wait(&newproc, &cmd->exit_code, &cmd->exit_why,
67         APR_WAIT);
68     check(rv == APR_CHILD_DONE, "Nie udało się zaczekać.");
69
70     check(cmd->exit_code == 0, "Wykonywanie %s zakończyło się niepowodzeniem.",
       ↪cmd->exe);
71     check(cmd->exit_why == APR_PROC_EXIT, "Polecenie %s zostało przerwane lub
       ↪uległo awari.",
72             cmd->exe);
73
74     return 0;
75
76 error:
77     return -1;
78 }
79
80 Shell CLEANUP_SH = {
81     .exe = "rm",
82     .dir = "/tmp",
83     .args = {"rm", "-rf", "/tmp/pkg-build", "/tmp/pkg-src.tar.gz",
84     "/tmp/pkg-src.tar.bz2", "/tmp/DEPENDS", NULL}
85 };
```

```
 86
 87 Shell GIT_SH = {
 88     .dir = "/tmp",
 89     .exe = "git",
 90     .args = {"git", "clone", "URL", "pkg-build", NULL}
 91 };
 92
 93 Shell TAR_SH = {
 94     .dir = "/tmp/pkg-build",
 95     .exe = "tar",
 96     .args = {"tar", "-xzf", "FILE", "--strip-components", "1", NULL}
 97 };
 98
 99 Shell CURL_SH = {
100     .dir = "/tmp",
101     .exe = "curl",
102     .args = {"curl", "-L", "-o", "TARGET", "URL", NULL}
103 };
104
105 Shell CONFIGURE_SH = {
106     .exe = "./configure",
107     .dir = "/tmp/pkg-build",
108     .args = {"configure", "OPTS", NULL}
109     ,
110 };
111
112 Shell MAKE_SH = {
113     .exe = "make",
114     .dir = "/tmp/pkg-build",
115     .args = {"make", "OPTS", NULL}
116 };
117
118 Shell INSTALL_SH = {
119     .exe = "sudo",
120     .dir = "/tmp/pkg-build",
121     .args = {"sudo", "make", "TARGET", NULL}
122 };
```

Przeanalizuj kod źródłowy pliku *shell.c* od końca do początku (to jest powszechnie stosowany układ kodu źródłowego C), a zobaczysz, jak utworzyłem rzeczywiste zmienne struktury `Shell`, które w pliku nagłówkowym *shell.h* zdefiniowałem jako `extern`. Wprawdzie wymienione zmienne znajdują się w pliku *shell.c*, ale pozostają dostępne dla całego programu. W ten właśnie sposób można utworzyć zmienne globalne pozostające w pojedynczym pliku *.o*, a jednocześnie gotowe do użycia w pozostałych komponentach programu. Nie powinieneś tworzyć zbyt wielu tego rodzaju zmiennych, choć okazują się użyteczne do wykonywania zadań takich jak w omawianym programie.

Kontynuując analizę pliku, docieramy do funkcji `Shell_run()` będącej funkcją bazową przeznaczoną do wykonywania poleceń zgodnie z informacjami dostarczanymi przez strukturę `Shell`. Wykorzystujemy tutaj wiele funkcji zdefiniowanych w pliku nagłówkowym *apr_thread_proc.h*, więc zapoznaj się z nimi, aby dokładnie poznać sposób działania naszej funkcji bazowej. Wydaje się, że to dużo pracy w porównaniu z użyciem po prostu wywołania funkcji

`system()`, ale otrzymujemy większą kontrolę nad przebiegiem wykonywania programu. Na przykład w strukturze `Shell` mamy atrybut `.dir`, wymuszający na programie przejście do określonego katalogu przed wykonaniem polecenia.

Na końcu mamy funkcję `Shell_exec()`, która pobiera zmienną liczbę argumentów. Tego rodzaju funkcję widziałeś już wcześniej, ale mimo wszystko upewnij się o przeanalizowaniu funkcji wskazanych w pliku *stdarg.h*. W wyzwaniu przeznaczonym dla tej sekcji będziesz zajmował się analizą funkcji `Shell_exec()`.

Wyzwanie 2. Przeanalizuj funkcję Shell_exec()

Wyzwaniem dotyczącym plików *shell.h* i *shell.c* (poza oczywiście pełną analizą kodu, podobnie jak w poprzednim wyzwaniu) jest dokładna analiza funkcji `Shell_exec()` i poznanie sposobu jej działania. Powinieneś dokładnie znać działanie każdego wiersza funkcji, obu pętli `for` oraz sposobu zastępowania argumentów.

Po przeanalizowaniu pliku dodaj nowy element do struktury `Shell` pozwalający na pobranie liczby argumentów, które muszą być zastąpione. Uaktualnij wszystkie polecenia, aby otrzymywały poprawne informacje o liczbie argumentów, oraz przeprowadź sprawdzenie mające na celu potwierdzenie, że wspomniane argumenty zostały zastąpione. W przypadku wystąpienia błędu program powinien kończyć działanie.

Funkcje poleceń programu

Przechodzimy teraz do przygotowania rzeczywistych poleceń wykonujących zadania, do których został przeznaczony program. Polecenia te będą używały funkcji pochodzących z bibliotek APR oraz plików *db.h* i *shell.h* w celu realizacji faktycznych operacji pobrania i kompilacji wskazanego oprogramowania. To będzie najbardziej skomplikowany zestaw plików, więc zajmuj się nimi niezwykle dokładnie. Podobnie jak wcześniej, zacznij od utworzenia pliku nagłówkowego *commands.h*, a następnie zaimplementuj jego funkcje w pliku *commands.c*.

Plik *commands.h*:

```
#ifndef _commands_h
#define _commands_h

#include <apr_pools.h>

#define DEPENDS_PATH "/tmp/DEPENDS"
#define TAR_GZ_SRC "/tmp/pkg-src.tar.gz"
#define TAR_BZ2_SRC "/tmp/pkg-src.tar.bz2"
#define BUILD_DIR "/tmp/pkg-build"
#define GIT_PAT "*.git"
#define DEPEND_PAT "*DEPENDS"
#define TAR_GZ_PAT "*.tar.gz"
#define TAR_BZ2_PAT "*.tar.bz2"
#define CONFIG_SCRIPT "/tmp/pkg-build/configure"

enum CommandType {
    COMMAND_NONE, COMMAND_INSTALL, COMMAND_LIST, COMMAND_FETCH,
```

```
    COMMAND_INIT, COMMAND_BUILD
};

int Command_fetch(apr_pool_t * p, const char *url, int fetch_only);
int Command_install(apr_pool_t * p, const char *url,
        const char *configure_opts, const char *make_opts,
        const char *install_opts);

int Command_depends(apr_pool_t * p, const char *path);

int Command_build(apr_pool_t * p, const char *url,
        const char *configure_opts, const char *make_opts,
        const char *install_opts);

#endif
```

W powyższym pliku nie znajdziesz zbyt wiele nowych rzeczy, z którymi nie spotkałeś się już wcześniej. Zwróć uwagę na istnienie pewnych definicji ciągów tekstowych używanych w całym programie. Najbardziej interesujący kod devpkg znajduje się właśnie w pliku *commands.c*.

Plik *commands.c*:

```
 1 #include <apr_uri.h>
 2 #include <apr_fnmatch.h>
 3 #include <unistd.h>
 4
 5 #include "commands.h"
 6 #include "dbg.h"
 7 #include "bstrlib.h"
 8 #include "db.h"
 9 #include "shell.h"
10
11 int Command_depends(apr_pool_t * p, const char *path)
12 {
13     FILE *in = NULL;
14     bstring line = NULL;
15
16     in = fopen(path, "r");
17     check(in != NULL, "Nie udało się otworzyć pobranych zależności: %s", path);
18
19     for (line = bgets((bNgetc) fgetc, in, '\n');
20             line != NULL;
21             line = bgets((bNgetc) fgetc, in, '\n'))
22     {
23         btrimws(line);
24         log_info("Przetwarzanie zależności: %s", bdata(line));
25         int rc = Command_install(p, bdata(line), NULL, NULL, NULL);
26         check(rc == 0, "Nie udało się zainstalować: %s", bdata(line));
27         bdestroy(line);
28     }
29
```

```
30      fclose(in);
31      return 0;
32
33  error:
34      if (line) bdestroy(line);
35      if (in) fclose(in);
36      return -1;
37  }
38
39  int Command_fetch(apr_pool_t * p, const char *url, int fetch_only)
40  {
41      apr_uri_t info = {.port = 0 };
42      int rc = 0;
43      const char *depends_file = NULL;
44      apr_status_t rv = apr_uri_parse(p, url, &info);
45
46      check(rv == APR_SUCCESS, "Nie udało się przetworzyć adresu URL: %s", url);
47
48      if (apr_fnmatch(GIT_PAT, info.path, 0) == APR_SUCCESS) {
49          rc = Shell_exec(GIT_SH, "URL", url, NULL);
50          check(rc == 0, "Wykonanie polecenia git zakończyło się
            ↪niepowodzeniem.");
51      } else if (apr_fnmatch(DEPEND_PAT, info.path, 0) == APR_SUCCESS) {
52          check(!fetch_only, "Nie ma sensu pobieranie pliku zależności.");
53
54          if (info.scheme) {
55              depends_file = DEPENDS_PATH;
56              rc = Shell_exec(CURL_SH, "URL", url, "TARGET", depends_file,
57                      NULL);
58              check(rc == 0, "Wykonanie polecenia curl zakończyło się
                ↪niepowodzeniem.");
59          } else {
60              depends_file = info.path;
61          }
62
63          // Rekurencyjne przetwarzanie listy devpkg.
64          log_info("Kompilacja zgodnie z plikiem DEPENDS: %s", url);
65          rv = Command_depends(p, depends_file);
66          check(rv == 0, "Nie udało się przetworzyć DEPENDS: %s", url);
67
68          // To wskazuje na brak operacji do wykonania.
69          return 0;
70
71      } else if (apr_fnmatch(TAR_GZ_PAT, info.path, 0) == APR_SUCCESS) {
72          if (info.scheme) {
73              rc = Shell_exec(CURL_SH,
74                      "URL", url, "TARGET", TAR_GZ_SRC, NULL);
75              check(rc == 0, "Nie udało się pobrać kodu źródłowego: %s", url);
76          }
77
78          rv = apr_dir_make_recursive(BUILD_DIR,
79                  APR_UREAD | APR_UWRITE |
80                  APR_UEXECUTE, p);
```

```
81          check(rv == APR_SUCCESS, "Nie udało się utworzyć katalogu %s",
82                  BUILD_DIR);
83
84          rc = Shell_exec(TAR_SH, "FILE", TAR_GZ_SRC, NULL);
85          check(rc == 0, "Nie udało się rozpakować %s", TAR_GZ_SRC);
86      } else if (apr_fnmatch(TAR_BZ2_PAT, info.path, 0) == APR_SUCCESS) {
87          if (info.scheme) {
88              rc = Shell_exec(CURL_SH, "URL", url, "TARGET", TAR_BZ2_SRC,
89                      NULL);
90              check(rc == 0, "Wykonanie polecenia curl zakończyło się
                ↪niepowodzeniem.");
91          }
92
93          apr_status_t rc = apr_dir_make_recursive(BUILD_DIR,
94                  APR_UREAD | APR_UWRITE
95                  | APR_UEXECUTE, p);
96
97          check(rc == 0, "Nie udało się utworzyć katalogu %s", BUILD_DIR);
98          rc = Shell_exec(TAR_SH, "FILE", TAR_BZ2_SRC, NULL);
99          check(rc == 0, "Nie udało się rozpakować %s", TAR_BZ2_SRC);
100     } else {
101         sentinel("Nie wiadomo, jak obsłużyć %s", url);
102     }
103
104     // Wskazuje na konieczność rzeczywistego przeprowadzenia instalacji.
105     return 1;
106 error:
107     return -1;
108 }
109
110 int Command_build(apr_pool_t * p, const char *url,
111         const char *configure_opts, const char *make_opts,
112         const char *install_opts)
113 {
114     int rc = 0;
115
116     check(access(BUILD_DIR, X_OK | R_OK | W_OK) == 0,
117             "Katalog kompilacji nie istnieje: %s", BUILD_DIR);
118
119     // Rzeczywiste przeprowadzenie instalacji.
120     if (access(CONFIG_SCRIPT, X_OK) == 0) {
121         log_info("Istnieje skrypt konfiguracji, przetwarzam go.");
122         rc = Shell_exec(CONFIGURE_SH, "OPTS", configure_opts, NULL);
123         check(rc == 0, "Nie udało się przeprowadzić konfiguracji.");
124     }
125
126     rc = Shell_exec(MAKE_SH, "OPTS", make_opts, NULL);
127     check(rc == 0, "Nie udało się przeprowadzić kompilacji.");
128
129     rc = Shell_exec(INSTALL_SH,
130             "TARGET", install_opts ? install_opts : "install",
131             NULL);
132     check(rc == 0, "Nie udało się zainstalować.");
```

```
133
134     rc = Shell_exec(CLEANUP_SH, NULL);
135     check(rc == 0, "Nie udało się przeprowadzić operacji porządkujących
        ↪po kompilacji.");
136
137     rc = DB_update(url);
138     check(rc == 0, "Nie udało się dodać tego pakietu do bazy danych.");
139
140     return 0;
141
142 error:
143     return -1;
144 }
145
146 int Command_install(apr_pool_t * p, const char *url,
147         const char *configure_opts, const char *make_opts,
148         const char *install_opts)
149 {
150     int rc = 0;
151     check(Shell_exec(CLEANUP_SH, NULL) == 0,
152             "Nie udało się przeprowadzić operacji porządkujących przed
                ↪kompilacją.");
153
154     rc = DB_find(url);
155     check(rc != -1, "Błąd podczas sprawdzania bazy danych.");
156
157     if (rc == 1) {
158         log_info("Pakiet %s jest już zainstalowany.", url);
159         return 0;
160     }
161
162     rc = Command_fetch(p, url, 0);
163
164     if (rc == 1) {
165         rc = Command_build(p, url, configure_opts, make_opts,
166                 install_opts);
167         check(rc == 0, "Nie udało się przeprowadzić kompilacji: %s", url);
168     } else if (rc == 0) {
169         // Nie ma konieczności przeprowadzenia instalacji.
170         log_info("Instalacja zależności zakończyła się powodzeniem: %s", url);
171     } else {
172         // Wystąpił błąd
173         sentinel("Instalacja zakończyła się powodzeniem: %s", url);
174     }
175
176     Shell_exec(CLEANUP_SH, NULL);
177     return 0;
178
179 error:
180     Shell_exec(CLEANUP_SH, NULL);
181     return -1;
182 }
```

Po wprowadzeniu powyższego kodu źródłowego i jego kompilacji możesz przystąpić do analizy kodu. Jeżeli sprostałeś przedstawionym dotąd wyzwaniom, powinieneś wiedzieć, że funkcje zdefiniowane w pliku *shell.c* są używane do uruchamiania narzędzi powłoki, oraz znać metodę zastępowania argumentów. Jeżeli jednak pominąłeś poprzednie wyzwania, wróć do wcześniejszego fragmentu ćwiczenia i upewnij się o *dokładnym* zrozumieniu mechanizmu działania funkcji `Shell_exec()`.

Wyzwanie 3. Krytyka mojego rozwiązania

Podobnie jak wcześniej, dokładnie przeanalizuj pełny kod źródłowy i upewnij się, że jest dokładnie taki sam jak w książce. Następnie przejrzyj każdą funkcję i sprawdź, czy na pewno znasz mechanizm jej działania i przeznaczenie. Powinieneś również prześledzić wywoływanie poszczególnych funkcji przez inne funkcje w tym oraz pozostałych plikach. Na koniec upewnij się, że rozumiesz sposób działania wszystkich funkcji wywoływanych z bibliotek APR.

Po prawidłowym utworzeniu pliku i jego przeanalizowaniu przejrzyj go raz jeszcze i przyjmij założenie, że jestem idiotą. Skrytykuj przygotowany przeze mnie projekt i zobacz, czy potrafisz go usprawnić. *Nie wprowadzaj* rzeczywistych zmian w kodzie, lecz jedynie utwórz niewielki plik *notes.txt* i zamieść w nim swoje przemyślenia o zmianach, które chciałbyś przeprowadzić.

Funkcja main() w devpkg

Ostatni i najważniejszy plik (choć prawdopodobnie najprostszy) to *devpkg.c* zawierający funkcję `main()`. Nie ma dla niego pliku nagłówkowego, ponieważ zawiera odwołania do wszystkich pozostałych. Tworzymy tutaj program wykonywalny devpkg i umieszczamy odwołania do pozostałych plików *.o* wskazanych w pliku *Makefile*. Wprowadź przedstawiony poniżej kod źródłowy i upewnij się, że jest poprawny.

Plik *devpkg.c*:

```
 1 #include <stdio.h>
 2 #include <apr_general.h>
 3 #include <apr_getopt.h>
 4 #include <apr_strings.h>
 5 #include <apr_lib.h>
 6
 7 #include "dbg.h"
 8 #include "db.h"
 9 #include "commands.h"
10
11 int main(int argc, const char const *argv[])
12 {
13     apr_pool_t *p = NULL;
14     apr_pool_initialize();
15     apr_pool_create(&p, NULL);
16
17     apr_getopt_t *opt;
18     apr_status_t rv;
19
20     char ch = '\0';
```

```
21      const char *optarg = NULL;
22      const char *config_opts = NULL;
23      const char *install_opts = NULL;
24      const char *make_opts = NULL;
25      const char *url = NULL;
26      enum CommandType request = COMMAND_NONE;
27
28      rv = apr_getopt_init(&opt, p, argc, argv);
29
30      while (apr_getopt(opt, "I:Lc:m:i:d:SF:B:", &ch, &optarg) ==
31              APR_SUCCESS) {
32          switch (ch) {
33              case 'I':
34                  request = COMMAND_INSTALL;
35                  url = optarg;
36                  break;
37
38              case 'L':
39                  request = COMMAND_LIST;
40                  break;
41
42              case 'c':
43                  config_opts = optarg;
44                  break;
45
46              case 'm':
47                  make_opts = optarg;
48                  break;
49
50              case 'i':
51                  install_opts = optarg;
52                  break;
53
54              case 'S':
55                  request = COMMAND_INIT;
56                  break;
57
58              case 'F':
59                  request = COMMAND_FETCH;
60                  url = optarg;
61                  break;
62
63              case 'B':
64                  request = COMMAND_BUILD;
65                  url = optarg;
66                  break;
67          }
68      }
69
70      switch (request) {
71          case COMMAND_INSTALL:
72              check(url, "Konieczne jest podanie przynajmniej adresu URL.");
73              Command_install(p, url, config_opts, make_opts, install_opts);
```

```
74              break;
75
76          case COMMAND_LIST:
77              DB_list();
78              break;
79
80          case COMMAND_FETCH:
81              check(url != NULL, "Musisz podać adres URL.");
82              Command_fetch(p, url, 1);
83              log_info("Pobrano do %s oraz w /tmp/", BUILD_DIR);
84              break;
85
86          case COMMAND_BUILD:
87              check(url, "Konieczne jest podanie przynajmniej adresu URL.");
88              Command_build(p, url, config_opts, make_opts, install_opts);
89              break;
90
91          case COMMAND_INIT:
92              rv = DB_init();
93              check(rv == 0, "Nie udało się utworzyć bazy danych.");
94              break;
95
96          default:
97              sentinel("Podano nieprawidłowe polecenie.");
98      }
99
100     return 0;
101
102 error:
103     return 1;
104 }
```

Wyzwanie 4. Pliki README i testów jednostkowych

Wyzwaniem dla przedstawionego powyżej pliku jest dokładne poznanie mechanizmu przetwarzania argumentów, poznanie ich znaczenia, a następnie utworzenie pliku *README* wraz z informacjami o sposobie użycia programu. Podczas przygotowywania pliku *README* utwórz także prosty plik *test.sh*, wykonujący *./devpkg* w celu sprawdzenia, czy każde polecenie faktycznie działa względem rzeczywistego kodu. Wykorzystaj opcję -e na początku skryptu, aby przerwać jego działanie w przypadku wystąpienia pierwszego błędu.

Na koniec uruchom program w debugerze i upewnij się o jego prawidłowym działaniu, zanim przejdziesz do ostatniego wyzwania.

Ostatnie wyzwanie

Ostatnie wyzwanie ma charakter miniegzaminu i obejmuje trzy zadania:

- Porównaj utworzony przez siebie kod z przygotowanym przeze mnie, który znajdziesz w repozytorium GitHub. Zaczynając od oceny 100%, odejmuj 1% za każdy nieprawidłowy wiersz kodu.

- Powróć do utworzonego wcześniej pliku *notes.txt* i zaimplementuj zaproponowane w nim usprawnienia kodu i funkcjonalności programu devpkg.
- Utwórz alternatywną wersję devpkg, używając ulubionego języka programowania lub tego, który uważasz za najlepszy. Porównaj obie wersje, a następnie spróbuj usprawnić tę przygotowaną w C na podstawie tego, czego się nauczyłeś podczas przygotowywania wersji w drugim języku.

Aby porównać swój kod z moim, wydaj poniższe polecenia:

```
$ git clone git@github.com:zedshaw/learn-c-the-hard-way-lectures.git
$ diff -r devpkg learn-c-the-hard-way-lectures/devpkg/devpkg
```

Pierwsze polecenie spowoduje sklonowanie w katalogu o nazwie *devpkgzed* mojej wersji programu, co pozwoli później na użycie narzędzia `diff` w celu porównania Twojego kodu z moim. Pliki, z którymi pracowałeś w ćwiczeniu, pochodzą bezpośrednio z wymienionego projektu. Jeżeli otrzymasz inne wiersze kodu, będzie to oznaczać błąd.

Pamiętaj, że nie istnieje rzeczywiste zaliczenie lub niepowodzenie w omówionym projekcie. To jedynie wyzwanie dla Ciebie, aby podczas realizacji projektu postępować jak najdokładniej i najskrupulatniej.

ĆWICZENIE 42

Stos i kolejka

Na tym etapie książki powinieneś już wiedzieć dużo na temat struktur danych wykorzystywanych do budowy wszystkich pozostałych struktur danych. Mając do dyspozycji pewnego rodzaju elementy — `List`, `DArray`, `Hashmap` i `Tree` — na ich podstawie możesz utworzyć niemalże wszystko inne. Praktycznie wszystkie pozostałe komponenty będą korzystały z pewnego wariantu wymienionych struktur. Jeżeli nie, to mamy do czynienia z pewną egzotyczną strukturą danych, której prawdopodobnie nie potrzebujesz.

Stos (`Stack`) i kolejka (`Queue`) to bardzo proste struktury danych będące tak naprawdę wariantami struktur `List`. Obie używają struktury `List`, zachowując dyscyplinę lub konwencję nakazującą umieszczanie elementów zawsze na jednym z końców listy. W przypadku struktury danych `Stack` mamy umieszczanie elementu na stosie i usuwanie ze stosu. Natomiast w strukturze Queue zawsze umieszczamy element na początku, a usuwamy z końca.

Obie wymienione struktury danych można zaimplementować za pomocą wyłącznie CPP i dwóch plików nagłówkowych. Przygotowane przeze mnie pliki nagłówkowe są krótkie (21 wierszy kodu) i definiują wszystkie operacje stosu oraz listy bez zbędnych poleceń `define`.

Poniżej przedstawię plik testów jednostkowych, a Twoim zadaniem będzie implementacja pliku nagłówkowego pozwalającego na zaliczenie tych testów. Nie musisz tworzyć żadnych plików implementacji, na przykład *stack.c* lub *queue.c*. Przygotuj jedynie pliki nagłówkowe *stack.k* lub *queue.h* w takiej postaci, aby testy jednostkowe były zaliczane.

Plik *stack_tests.c*:

```
#include "minunit.h"
#include <lcthw/stack.h>
#include <assert.h>

static Stack *stack = NULL;
char *tests[] = { "dane testowe1", "dane testowe2", "dane testowe3" };

#define NUM_TESTS 3

char *test_create()
{
    stack = Stack_create();
    mu_assert(stack != NULL, "Nie udało się utworzyć stosu.");

    return NULL;
}

char *test_destroy()
{
    mu_assert(stack != NULL, "Nie udało się utworzyć stosu #2");
    Stack_destroy(stack);
```

```
22
23      return NULL;
24  }
25
26  char *test_push_pop()
27  {
28      int i = 0;
29      for (i = 0; i < NUM_TESTS; i++) {
30          Stack_push(stack, tests[i]);
31          mu_assert(Stack_peek(stack) == tests[i], "Nieprawidłowa następna
            ↪wartość.");
32      }
33
34      mu_assert(Stack_count(stack) == NUM_TESTS, "Nieprawidłowa wartość podczas
        ↪operacji wstawiania danych.");
35
36      STACK_FOREACH(stack, cur) {
37          debug("WARTOŚĆ: %s", (char *)cur->value);
38      }
39
40      for (i = NUM_TESTS - 1; i >= 0; i--) {
41          char *val = Stack_pop(stack);
42          mu_assert(val == tests[i], "Nieprawidłowa wartość podczas operacji
            ↪usuwania danych.");
43      }
44
45      mu_assert(Stack_count(stack) == 0, "Nieprawidłowa wartość po operacji
        ↪usunięcia danych.");
46
47      return NULL;
48  }
49
50  char *all_tests()
51  {
52      mu_suite_start();
53
54      mu_run_test(test_create);
55      mu_run_test(test_push_pop);
56      mu_run_test(test_destroy);
57
58      return NULL;
59  }
60
61  RUN_TESTS(all_tests);
```

Przechodzimy teraz do pliku *queue_tests.c*, który jest niemalże taki sam, ale przeznaczony dla struktury danych `Queue`.

Plik *queue_tests.c*:

```
1  #include "minunit.h"
2  #include <lcthw/queue.h>
3  #include <assert.h>
```

```
 4
 5 static Queue *queue = NULL;
 6 char *tests[] = { "dane testowe1", "dane testowe2", "dane testowe3" };
 7
 8 #define NUM_TESTS 3
 9
10 char *test_create()
11 {
12     queue = Queue_create();
13     mu_assert(queue != NULL, "Nie udało się utworzyć kolejki.");
14
15     return NULL;
16 }
17
18 char *test_destroy()
19 {
20     mu_assert(queue != NULL, "Nie udało się utworzyć kolejki #2");
21     Queue_destroy(queue);
22
23     return NULL;
24 }
25
26 char *test_send_recv()
27 {
28     int i = 0;
29     for (i = 0; i < NUM_TESTS; i++) {
30         Queue_send(queue, tests[i]);
31         mu_assert(Queue_peek(queue) == tests[0], "Nieprawidłowa następna
           ↪wartość.");
32     }
33
34     mu_assert(Queue_count(queue) == NUM_TESTS, "Nieprawidłowa wartość podczas
       ↪operacji przekazywania danych.");
35
36     QUEUE_FOREACH(queue, cur) {
37         debug("WARTOŚĆ: %s", (char *)cur->value);
38     }
39
40     for (i = 0; i < NUM_TESTS; i++) {
41         char *val = Queue_recv(queue);
42         mu_assert(val == tests[i], "Prawidłowa wartość podczas operacji
           ↪odbierania danych.");
43     }
44
45     mu_assert(Queue_count(queue) == 0, "Prawidłowa wartość po operacji odebrania
       ↪danych.");
46
47     return NULL;
48 }
49
50 char *all_tests()
51 {
52     mu_suite_start();
```

```
53
54      mu_run_test(test_create);
55      mu_run_test(test_send_recv);
56      mu_run_test(test_destroyP
57
58      return NULL;
59  }
60
61  RUN_TESTS(all_tests);
```

Co powinieneś zobaczyć?

Testy jednostkowe powinny zostać zaliczone bez konieczności ich modyfikowania, a także bez żadnych błędów pamięci. Poniżej przedstawiłem wynik bezpośredniego wykonania testów `stack_tests`.

Sesja dla ćwiczenia 42.:

```
$ ./tests/stack_tests
DEBUG tests/stack_tests.c:60:
-----
WYKONYWANIE: ./tests/stack_tests
----
WYKONYWANIE: ./tests/stack_tests
DEBUG tests/stack_tests.c:53:
----- test_create
DEBUG tests/stack_tests.c:54:
----- test_push_pop
DEBUG tests/stack_tests.c:37: WARTOŚĆ: dane testowe3
DEBUG tests/stack_tests.c:37: WARTOŚĆ: dane testowe2
DEBUG tests/stack_tests.c:37: WARTOŚĆ: dane testowe1
DEBUG tests/stack_tests.c:55:
----- test_destroy
WSZYSTKIE TESTY ZOSTAŁY ZALICZONE
Liczba wykonanych testów: 3
$
```

Testy `queue_test` praktycznie powodują wygenerowanie tego samego rodzaju danych wyjściowych, a więc nie muszę ich tutaj pokazywać.

Jak można usprawnić kod?

Jedynym rzeczywistym usprawnieniem może być przejście z użycia listy (`List`) na tablicę dynamiczną (`DArray`). Struktura danych `Queue` jest znacznie trudniejsza do implementacji za pomocą `DArray`, ponieważ działa na obu stronach listy węzłów.

Wadą przedstawionego podejścia opartego wyłącznie na plikach nagłówkowych jest to, że nie można w łatwy sposób zająć się dostrojeniem wydajności przygotowanego rozwiązania. Tego rodzaju technika to praktycznie przygotowanie protokołu, określającego, jak używać

struktury danych `List` w określonym stylu. Jeżeli podczas dostrajania wydajności uda się zapewnić szybkie działanie struktury danych `List`, to powinno mieć to przełożenie na szybkość działania struktur `Stack` i `Queue`.

Zadania dodatkowe

- Zaimplementuj strukturę `Stack` za pomocą `DArray` zamiast `List`, ale bez zmiany testu jednostkowego. Oznacza to konieczność utworzenia własnej wersji `STACK_FOREACH()`.

ĆWICZENIE 43

Prosty silnik dla danych statystycznych

To jest prosty algorytm, który wykorzystuję w celu otrzymania podsumowania danych statystycznych bez konieczności przechowywania wszystkich próbek. Stosuję go w każdym oprogramowaniu wymagającym obsługi pewnych danych statystycznych, takich jak średnia, odchylenie standardowe i suma, ale niepozwalającym na przechowywanie wszystkich próbek. Zamiast tego przechowuję jedynie wyniki obliczeń, na które tak naprawdę składa się jedynie pięć liczb.

Odchylenie standardowe i średnia

Do pracy potrzebujemy przede wszystkim sekwencji próbek. To może być cokolwiek, począwszy od czasu potrzebnego na wykonanie zadania, przez liczbę określającą, ile razy uzyskano dostęp do czegokolwiek, aż po wyrażoną w gwiazdkach ocenę witryny internetowej. To tak naprawdę nie ma znaczenia, o ile otrzymujemy strumień liczb i chcemy poznać przedstawione poniżej podsumowanie danych statystycznych dotyczące tych liczb:

Suma. Suma całkowita otrzymana po dodaniu wszystkich liczb.

Suma kwadratów. Suma kwadratów poszczególnych liczb.

Ilość (n). Liczba otrzymanych próbek.

Minimum. Najmniejsza z otrzymanych próbek.

Maksimum. Największa z otrzymanych próbek.

Średnia. To będzie prawdopodobnie średnia, niekoniecznie dokładna, ponieważ mamy do czynienia z medianą, ale i tak stanowi akceptowane przybliżenie średniej.

Odchylenie standardowe. Wartość obliczona na podstawie wzoru `$sqrt(suma_kwadratów - (suma * średnia)) / (n - 1)))$`, gdzie `sqrt` oznacza zdefiniowaną w pliku nagłówkowym *math.h* funkcję kwadratu.

Powyższe obliczenia można potwierdzić za pomocą kodu w języku R, ponieważ wiadomo, że będą przez R obliczone prawidłowo.

Sesja dla ćwiczenia 43.:

```
> s <- runif(n=10, max=10)
> s
 [1] 6.1061334 9.6783204 1.2747090 8.2395131 0.3333483 6.9755066 1.0626275
 [8] 7.6587523 4.9382973 9.5788115
> summary(s)
Min. 1st Qu. Median Mean 3rd Qu. Max.
0.3333 2.1910 6.5410 5.5850 8.0940 9.6780
> sd(s)
```

```
 9 [1] 3.547868
10 > sum(s)
11 [1] 55.84602
12 > sum(s * s)
13 [1] 425.1641
14 > sum(s) * mean(s)
15 [1] 311.8778
16 > sum(s * s) - sum(s) * mean(s)
17 [1] 113.2863
18 > (sum(s * s) - sum(s) * mean(s)) / (length(s) - 1)
19 [1] 12.58737
20 > sqrt((sum(s * s) - sum(s) * mean(s)) / (length(s) - 1))
21 [1] 3.547868
22 >
```

Nie musisz znać języka R. Aby sprawdzić poprawność otrzymanych wyników powinno wystarczyć przedstawione poniżej wyjaśnienie.

Wiersze od 1. do 4. Użyłem funkcji `runif()` w celu otrzymania losowego, jednakowego rozkładu liczb, a następnie je wyświetliłem. Liczby te będą później wykorzystane w teście jednostkowym.

Wiersze od 5. do 7. To jest podsumowanie i pokazuje wartości obliczone przez R dla podanych liczb.

Wiersze od 8. do 9. To jest odchylenie standardowe obliczone za pomocą funkcji `sd()`.

Wiersze od 10. do 11. W tym miejscu rozpoczynamy ręczne obliczenia, na początek sprawdzamy sumę.

Wiersze od 12. do 13. Kolejnym elementem wzoru odchylenia standardowego jest `sumsq`, tę wartość otrzymujemy przez `sum(s * s)`, co nakazuje R pomnożenie całej listy `s` przez samą siebie, a następnie obliczenie sumy. Potęga języka R polega na możliwości przeprowadzania operacji matematycznych na całych strukturach danych, takich jak przedstawiona.

Wiersze od 14. do 15. Patrząc na wzór, widzimy konieczność pomnożenia sumy przez średnią, stąd `sum(s) * mean(s)`.

Wiersze od 16. do 17. Następnie łączymy sumę kwadratów z otrzymanym wynikiem, stąd `sum(s * s) - sum(s) * mean(s)`.

Wiersze od 18. do 19. Teraz wynik trzeba podzielić przez `$n-1$`, stąd `(sum(s * s) - sum(s) * mean(s)) / (length(s) - 1)`.

Wiersze od 20. do 21. Na koniec wykonujemy funkcję `sqrt()` i otrzymujemy wartość `3.547868` odpowiadającą liczbie, jaką język R nam obliczył.

Implementacja

Powyżej przedstawiłem sposób obliczenia odchylenia standardowego. Teraz możemy więc utworzyć prosty kod implementujący te obliczenia.

Plik *stats.h*:

```
#ifndef lcthw_stats_h
#define lcthw_stats_h

typedef struct Stats {
    double sum;
    double sumsq;
    unsigned long n;
    double min;
    double max;
} Stats;

Stats *Stats_recreate(double sum, double sumsq, unsigned long n,
        double min, double max);

Stats *Stats_create();

double Stats_mean(Stats * st);

double Stats_stddev(Stats * st);

void Stats_sample(Stats * st, double s);

void Stats_dump(Stats * st);

#endif
```

Jak możesz zobaczyć, niezbędne do przeprowadzenia obliczenia zostały umieszczone w strukturze, a następnie zdefiniowałem funkcje odpowiedzialne za próbkowanie i pobranie liczb. Implementacja jest tak naprawdę ćwiczeniem ze znajomości matematyki.

Plik *stats.c*:

```
 1 #include <math.h>
 2 #include <lcthw/stats.h>
 3 #include <stdlib.h>
 4 #include <lcthw/dbg.h>
 5
 6 Stats *Stats_recreate(double sum, double sumsq, unsigned long n,
 7         double min, double max)
 8 {
 9     Stats *st = malloc(sizeof(Stats));
10     check_mem(st);
11
12     st->sum = sum;
13     st->sumsq = sumsq;
14     st->n = n;
15     st->min = min;
16     st->max = max;
17
18     return st;
19
```

```
20 error:
21     return NULL;
22 }
23
24 Stats *Stats_create()
25 {
26     return Stats_recreate(0.0, 0.0, 0L, 0.0, 0.0);
27 }
28
29 double Stats_mean(Stats * st)
30 {
31     return st->sum / st->n;
32 }
33
34 double Stats_stddev(Stats * st)
35 {
36     return sqrt((st->sumsq - (st->sum * st->sum / st->n)) /
37     (st->n - 1));
38 }
39
40 void Stats_sample(Stats * st, double s)
41 {
42     st->sum += sP
43     st->sumsq += s * s;
44
45     if (st->n == 0) {
46         st->min = sP
47         st->max = s;
48     } else {
49         if (st->min > s)
50             st->min = s;
51         if (st->max < s)
52             st->max = s;
53     }
54
55     st->n += 1;
56 }
57
58 void Stats_dump(Stats * st)
59 {
60     fprintf(stderr,
61         "sum: %f, sumsq: %f, n: %ld, "
62         "min: %f, max: %f, mean: %f, stddev: %f",
63         st->sum, st->sumsq, st->n, st->min, st->max, Stats_mean(st),
64         Stats_stddev(st));
65 }
```

Oto omówienie poszczególnych funkcji zdefiniowanych w pliku *stats.c*.

Stats_recreate(). Liczby są wczytywane z pewnego rodzaju bazy danych, a ta funkcja pozwala na ponowne utworzenie struktury `Stats`.

Stats_create(). To jest po prostu wywołanie `Stats_recreate()` wraz z wszystkimi wartościami `0`.

Stats_mean(). Użycie sumy (sum) i wartości n w efekcie daje średnią.

Stats_stddev(). Mamy tutaj implementację opracowanego wcześniej wzoru. Jedyna różnica polega na tym, że tutaj obliczamy średnią za pomocą st->sum / st->n, zamiast wywoływać Stats_mean().

Stats_sample(). Ta funkcja zajmuje się obsługą liczb w strukturze Stats. Po podaniu pierwszej wartości powoduje przypisanie n wartości 0 oraz odpowiednie obliczenie min i max. Później każde wywołanie powoduje zwiększenie wartości sum, sumsq i n, a następnie sprawdzenie, czy nowa próbka będzie nową wartością min lub max.

Stats_dump(). Prosta funkcja debugowania, która pobiera dane statystyczne, aby można je było wyświetlić.

Ostatnią rzeczą, którą trzeba zrobić, jest potwierdzenie poprawności przeprowadzanych operacji matematycznych. Liczby i obliczenia pochodzące z sesji pracy z językiem R wykorzystamy do utworzenia testu jednostkowego potwierdzającego otrzymanie prawidłowych wyników.

Plik *stats_tests.c*:

```
#include "minunit.h"
#include <lcthw/stats.h>
#include <math.h>

const int NUM_SAMPLES = 10;
double samples[] = {
    6.1061334, 9.6783204, 1.2747090, 8.2395131, 0.3333483,
    6.9755066, 1.0626275, 7.6587523, 4.9382973, 9.5788115
};

Stats expect = {
    .sumsq = 425.1641,
    .sum = 55.84602,
    .min = 0.333,
    .max = 9.678,
    .n = 10,
};

double expect_mean = 5.584602;
double expect_stddev = 3.547868;

#define EQ(X,Y,N) (round((X) * pow(10, N)) == round((Y) * pow(10, N)))

char *test_operations()
{
    int i = 0;
    Stats *st = Stats_create();
    mu_assert(st != NULL, "Nie udało się utworzyć struktury Stats.");

    for (i = 0; i < NUM_SAMPLES; i++) {
        Stats_sample(st, samples[i]);
```

```
32      }
33
34      Stats_dump(st);
35
36      mu_assert(EQ(st->sumsq, expect.sumsq, 3), "Nieprawidłowa wartość sumsq.");
37      mu_assert(EQ(st->sum, expect.sum, 3), "Nieprawidłowa wartość sum.");
38      mu_assert(EQ(st->min, expect.min, 3), "Nieprawidłowa wartość min.");
39      mu_assert(EQ(st->max, expect.max, 3), "Nieprawidłowa wartość max.");
40      mu_assert(EQ(st->n, expect.n, 3), "Nieprawidłowa wartość max.");
41      mu_assert(EQ(expect_mean, Stats_mean(st), 3), "Nieprawidłowa wartość mean.");
42      mu_assert(EQ(expect_stddev, Stats_stddev(st), 3),
43          "Nieprawidłowa wartość stddev.");
44
45      return NULL;
46  }
47
48  char *test_recreate()
49  {
50      Stats *st = Stats_recreate(
51          expect.sum, expect.sumsq, expect.n, expect.min, expect.max);
52
53      mu_assert(st->sum == expect.sum, "Wartość sum nie jest równa.");
54      mu_assert(st->sumsq == expect.sumsq, "Wartość sumsq nie jest równa.");
55      mu_assert(st->n == expect.n, "Wartość n nie jest równa.");
56      mu_assert(st->min == expect.min, "Wartość min nie jest równa.");
57      mu_assert(st->max == expect.max, "Wartość max nie jest równa.");
58      mu_assert(EQ(expect_mean, Stats_mean(st), 3), "Nieprawidłowa wartość.");
59      mu_assert(EQ(expect_stddev, Stats_stddev(st), 3),
60          "Nieprawidłowa wartość stddev.");
61
62      return NULL;
63  }
64
65  char *all_tests()
66  {
67      mu_suite_start();
68
69      mu_run_test(test_operations);
70      mu_run_test(test_recreate);
71
72      return NULL;
73  }
74
75  RUN_TESTS(all_tests);
```

W powyższym teście jednostkowym nie znajdziesz nic nowego, być może z wyjątkiem makra `EQ()`. Ponieważ zaliczam się do leniwych osób i nie chcę w standardowy sposób sprawdzać, czy dwie wartości typu `double` są podobne, więc utworzyłem makro. Problem z wartościami typu `double` polega na tym, że w ich przypadku równość oznacza uzyskanie absolutnie identycznych wyników. Jednak tutaj używamy dwóch różnych systemów z dwoma nieco różnymi błędami zaokrąglenia. Rozwiązaniem przyjętym przeze mnie jest sprawdzenie, czy podane liczby są „równe do X miejsc po przecinku dziesiętnym”.

Do tego celu wykorzystuję makro `EQ()`, liczba zostaje podniesiona do potęgi 10, a następnie za pomocą funkcji `round()` otrzymujemy liczbę całkowitą. To jest prosty sposób zaokrąglenia do `N` miejsc po przecinku dziesiętnym i porównywania wyników jako liczb całkowitych. Jestem przekonany o istnieniu miliarda innych sposobów wykonania tego zadania; wybrany przeze mnie sprawdza się dobrze.

Oczekiwany wynik znajduje się w strukturze `Stats` i po prostu sprawdzamy, czy otrzymana liczba jest bardzo przybliżona do obliczonej przez R.

Jak można użyć tego rozwiązania?

Odchylenie standardowe i średnią możesz wykorzystać do ustalenia, czy nowa próbka jest interesująca lub czy można jej użyć do zebrania danych statystycznych dotyczących danych statystycznych. Pierwsze z wymienionych zadań jest łatwe do zrozumienia, a więc wyjaśnię je pokrótce na przykładzie liczby określającej czas, przez który użytkownik pozostaje zalogowany.

Wyobraź sobie monitorowanie ilości czasu spędzanego przez użytkownika w serwerze oraz monitorowanie użycia danych statystycznych do analizy tego czasu. Podczas każdego logowania rejestrujesz długość danej sesji pracy i wywołujesz funkcję `Stats_sample()`. Interesują nas użytkownicy pozostający zalogowani w serwerze zarówno zbyt długo, jak i zbyt krótko.

Zamiast definiować określone poziomy, po prostu porównujemy długość sesji użytkownika z zakresem `średnia (+ lub -) 2 * odchylenie standardowe`. Otrzymujemy średnią oraz dwukrotność odchylenia standardowego, a czas, przez który użytkownik pozostaje zalogowany, uznajemy za interesujący, gdy wykracza poza te dwa zakresy. Ponieważ dane statystyczne zbieramy za pomocą algorytmu zmiennego, więc obliczenia są niezwykle szybkie, a oprogramowanie może oznaczać użytkowników wykraczających poza zakres.

To oczywiście nie oznacza użytkowników zachowujących się nieprawidłowo, lecz jedynie wskazuje na potencjalne problemy, którym warto się przyjrzeć, aby wiedzieć, co się dzieje. Oznaczenie odbywa się na podstawie zachowania wszystkich użytkowników, co pozwala na uniknięcie problemu wyboru pewnej liczby zupełnie oderwanej od rzeczywistości.

Ogólna zasada jest następująca: w przypadku wzoru `średnia (+ lub -) 2 * odchylenie standardowe` można oszacować, że 90% wartości zmieści się w zakresie, a wszystkie spoza niego są interesujące.

Drugim sposobem wykorzystania tego rodzaju danych statystycznych jest potraktowanie ich jako metadanych i wykorzystanie do innych obliczeń w strukturze `Stats`. Praktycznie standardowo wykonujesz funkcję `Stats_sample()` dla `min`, `max`, `n`, `mean` i `stddev` dla danej próbki. W ten sposób otrzymujesz dwupoziomowy pomiar i możesz porównać próbki próbek.

To niezrozumiałe, prawda? Kontynuujemy omawianie wcześniejszego przykładu, ale przyjmujemy założenie, że mamy 100 serwerów, z których każdy obsługuje inną aplikację. Zaczynasz monitorować czas, przez który użytkownicy pozostają zalogowani w poszczególnych serwerach aplikacji. Chcesz porównać wszystkie 100 aplikacji i oznaczyć użytkowników, którzy logują się zbyt często we wszystkich z nich. Najłatwiejszym sposobem jest obliczenie danych

statystycznych nowego logowania za każdym razem, gdy użytkownik się loguje, a następnie dodanie elementu *tej* struktury `Stats` do drugiej struktury.

W takim przypadku otrzymujemy serię danych statystycznych, które można określić następująco:

Średnia średnich. Pełna struktura `Stats` dostarczająca informacje o średniej i odchyleniu standardowym średnich dla wszystkich serwerów. Każdy serwer lub użytkownik spoza zakresu jest przydatny w analizie na poziomie globalnym.

Średnia odchyleń standardowych. Kolejna struktura `Stats` generująca dane statystyczne dotyczące *wszystkich* serwerów. Możesz przeanalizować poszczególne serwery i sprawdzić, czy w przypadku któregokolwiek z nich występuje nietypowo szeroki zakres liczb w porównaniu z ich odchyleniem standardowym względem średniej odchyleń standardowych.

Wprawdzie można wygenerować wszystkie dane statystyczne, ale poniżej wymieniłem te najbardziej użyteczne. Jeżeli chcesz monitorować serwery pod kątem nieregularnego czasu, przez który użytkownik pozostaje zalogowany, to możesz podjąć następujące działania.

- Użytkownik Janek loguje się do serwera A i wylogowuje się z niego. Pobierz dane statystyczne dla serwera A i uaktualnij je.
- Pobierz dane statystyczne średniej średnich, a następnie pobierz średnią serwera A i dodaj ją jako próbkę. Tej operacji nadajemy nazwę `m_of_m`.
- Pobierz dane statystyczne średniej odchylenia standardowego, a następnie pobierz odchylenie standardowe serwera A i dodaj je jako próbkę. Tej operacji nadajemy nazwę `m_of_s`.
- Jeżeli średnia serwera A jest spoza zakresu `m_of_m.mean + 2 * m_of_m.stddev`, to oznacz tę sytuację jako potencjalny problem.
- Jeżeli odchylenie standardowe serwera A jest spoza zakresu `m_of_s.mean + 2 * m_of_s.stddev`, to oznacz tę sytuację jako potencjalny problem.
- Jeżeli czas, przez który użytkownik Janek pozostaje zalogowany, jest spoza zakresu serwera A lub `m_of_m` dla serwera A, oznacz tę sytuację jako interesującą.

Używając tego rodzaju obliczenia średniej średnich i średniej odchyleń standardowych, można efektywnie monitorować wiele danych statystycznych przy minimalnych wymaganiach w zakresie przetwarzania i miejsca w pamięci masowej.

Zadania dodatkowe

- Skonwertuj funkcje `Stats_stddev()` i `Stats_mean()` na typ `static inline` w pliku *stats.h* zamiast w pliku *stats.c*.
- Użyj tego kodu w celu przygotowania testu wydajności dla *string_algos_test.c*. Test powinien być opcjonalny i wykonywać test bazowy w charakterze serii próbek, a następnie przekazywać wyniki.

- Utwórz wersję tego kodu w innym znanym Ci języku programowania. Opierając się na przedstawionych przeze mnie danych, potwierdź poprawność przygotowanej implementacji.
- Utwórz niewielki program pobierający plik pełen liczb i generujący dla niego omówione w ćwiczeniu dane statystyczne.
- Zmodyfikuj program w taki sposób, aby akceptował tabele danych zawierające wiersz nagłówka, a następnie w wierszach pozostałe liczby rozdzielone dowolną liczbą spacji. Program powinien wyświetlać dane statystyczne dla każdej kolumny, a każda kolumna powinna mieć nagłówek.

ĆWICZENIE 44

Bufor cykliczny

Bufor cykliczny (ang. *ring buffer*) jest niezwykle użyteczny podczas przetwarzania asynchronicznych operacji wejścia-wyjścia. Pozwala na otrzymywanie danych po jednej stronie w losowych odstępach czasu oraz o losowej wielkości, natomiast po drugiej stronie dostarcza spójne fragmenty danych w ustalonych odstępach czasu oraz wielkości. Tego rodzaju bufor jest odmianą struktury danych kolejki, ale skoncentrowaną na blokach bajtów zamiast na liście wskaźników. W tym ćwiczeniu przedstawię kod struktury `RingBuffer`, a następnie Twoim zadaniem będzie utworzenie dla niej pełnych testów jednostkowych.

Plik *ringbuffer.h*:

```
#ifndef _lcthw_RingBuffer_h
#define _lcthw_RingBuffer_h

#include <lcthw/bstrlib.h>

typedef struct {
    char *buffer;
    int length;
    int start;
    int end;
} RingBuffer;

RingBuffer *RingBuffer_create(int length);

void RingBuffer_destroy(RingBuffer * buffer);

int RingBuffer_read(RingBuffer * buffer, char *target, P amount);

int RingBuffer_write(RingBuffer * buffer, char *data, int length);

int RingBuffer_empty(RingBuffer * buffer);

int RingBuffer_full(RingBuffer * buffer);

int RingBuffer_available_data(RingBuffer * buffer);

int RingBuffer_available_space(RingBuffer * buffer);

bstring RingBuffer_gets(RingBuffer * buffer, int amount);

#define RingBuffer_available_data(B) (\
    ((B)->end + 1) % (B)->length - (B)->start - 1)

#define RingBuffer_available_space(B) (\
    (B)->length - (B)->end - 1)

```

```
#define RingBuffer_full(B) (RingBuffer_available_data((B))\
    - (B)->length == 0)

#define RingBuffer_empty(B) (\
    RingBuffer_available_data((B)) == 0)

#define RingBuffer_puts(B, D) RingBuffer_write(\
    (B), bdata((D)), blength((D)))

#define RingBuffer_get_all(B) RingBuffer_gets(\
    (B), RingBuffer_available_data((B)))

#define RingBuffer_starts_at(B) (\
    (B)->buffer + (B)->start)

#define RingBuffer_ends_at(B) (\
    (B)->buffer + (B)->end)

#define RingBuffer_commit_read(B, A) (\
    (B)->start = ((B)->start + (A)) % (B)->length)

#define RingBuffer_commit_write(B, A) (\
    (B)->end = ((B)->end + (A)) % (B)->length)

#endif
```

Przyglądając się tej strukturze danych, możesz dostrzec zmienne `buffer`, `start` i `end`. Działanie tej struktury polega po prostu na przesuwaniu zmiennych `start` i `end` po buforze, aby powracać na jego początek po dotarciu do końca. W ten sposób otrzymujemy iluzję istnienia w małej przestrzeni urządzenia pozwalającego na nieskończony odczyt danych. W kodzie znalazł się zbiór makr przeznaczonych do przeprowadzania różnych obliczeń na tej podstawie.

Poniżej przedstawiłem implementację bufora cyklicznego, która stanowi znacznie lepsze wyjaśnienie mechanizmu jego działania.

Plik *ringbuffer.c*:

```
#undef NDEBUG
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <lcthw/dbg.h>
#include <lcthw/ringbuffer.h>

RingBuffer *RingBuffer_create(int length)
{
    RingBuffer *buffer = calloc(1, sizeof(RingBuffer));
    buffer->length = length + 1;
    buffer->start = 0;
    buffer->end = 0;
```

```
15      buffer->buffer = calloc(buffer->length, 1);
16
17      return buffer;
18  }
19
20  void RingBuffer_destroy(RingBuffer * bufferP
21  {
22      if (buffer) {
23          free(buffer->buffer);
24          free(buffer);
25      }
26  }
27
28  int RingBuffer_write(RingBuffer * buffer, char *data, int lengthP
29  {
30      if (RingBuffer_available_data(buffer) == 0) {
31          buffer->start = buffer->end = 0;
32      }
33
34      check(length <= RingBuffer_available_space(buffer),
35          "Brak wystarczającej ilości miejsca: %d żądane, %d dostępne",
36          RingBuffer_available_data(buffer), length);
37
38      void *result = memcpy(RingBuffer_ends_at(buffer), data, length);
39      check(result != NULL, "Nie udało się zapisać danych w buforze.");
40
41      RingBuffer_commit_write(buffer, length);
42
43      return length;
44  error:
45      return -1;
46  }
47
48  int RingBuffer_read(RingBuffer * buffer, char *target, int amount)
49  {
50      check_debug(amount <= RingBuffer_available_data(buffer),
51          "Brak wystarczającej ilości miejsca w buforze: jest %d, potrzeba %d",
52          RingBuffer_available_data(buffer), amount);
53
54      void *result = memcpy(target, RingBuffer_starts_at(buffer), amount);
55      check(result != NULL, "Nie udało się zapisać bufora w danych.");
56
57      RingBuffer_commit_read(buffer, amount);
58
59      if (buffer->end == buffer->start) {
60          buffer->start = buffer->end = 0;
61      }
62
63      return amount;
64  error:
65      return -1;
66  }
67
```

```
68 bstring RingBuffer_gets(RingBuffer * buffer, int amount)
69 {
70     check(amount > 0, "Potrzeba więcej niż 0 dla get, otrzymano: %d ",
71         amount);
72     check_debug(amount <= RingBuffer_available_data(buffer),
73         "Brak wystarczającej ilości w buforze.");
74
75     bstring result = blk2bstr(RingBuffer_starts_at(buffer), amount);
76     check(result != NULL, "Nie udało się utworzyć wyniku.");
77     check(blength(result) == amount, "Nieprawidłowa wielkość wyniku.");
78
79     RingBuffer_commit_read(buffer, amount);
80     assert(RingBuffer_available_data(buffer) >= 0
81         && "Błąd podczas odczytu.");
82
83     return result;
84 error:
85     return NULL;
86 }
```

Powyższy fragment kodu przedstawia podstawową implementację struktury `RingBuffer`. Można już odczytywać i zapisywać bloki danych w tym buforze, a także sprawdzać ilość znajdujących się w nim danych oraz ilość pozostałego wolnego miejsca. Wprawdzie istnieją znacznie bardziej skomplikowane implementacje bufora cyklicznego, wykorzystujące sztuczki systemu operacyjnego do utworzenia wyimaginowanego magazynu danych o nieskończonej wielkości, ale implementacje te nie są przenośne.

Ponieważ przedstawiona tutaj struktura `RingBuffer` zajmuje się odczytem i zapisem bloków pamięci, więc upewniamy się, że po każdym spełnieniu warunku `end == start` następuje wyzerowanie wartości wymienionych zmiennych, aby wskazywały początek bufora. Bufor cykliczny w wersji przedstawionej w Wikipedii nie zapisuje bloków danych, więc jego obsługa polega jedynie na przesuwaniu w kółko zmiennych `end` i `start`. Aby zapewnić lepszą obsługę bloków, konieczne jest przejście na początek bufora wewnętrznego za każdym razem, gdy nie ma danych.

Testy jednostkowe

Przygotowane testy jednostkowe powinny sprawdzać jak największą liczbę możliwych warunków. Najłatwiejsze podejście polega na przygotowaniu różnych struktur `RingBuffer`, a następnie ręczne sprawdzenie poprawności funkcji i obliczeń matematycznych. Na przykład można sprawdzić, czy zmienna `end` ma prawidłową wartość na końcu bufora, a `start` na początku, a później zobaczyć, jak to wygląda w przypadku niepowodzenia.

Co powinieneś zobaczyć?

Poniżej przedstawiłem zapis sesji po wykonaniu testów jednostkowych.

Sesja dla ćwiczenia 44.:

```
$ ./tests/ringbuffer_tests
DEBUG tests/ringbuffer_tests.c:60: ----- WYKONYWANIE: ./tests/ringbuffer_tests
----
WYKONYWANIE: ./tests/ringbuffer_tests
DEBUG tests/ringbuffer_tests.c:53:
----- test_create
DEBUG tests/ringbuffer_tests.c:54:
----- test_read_write
DEBUG tests/ringbuffer_tests.c:55:
----- test_destroy
WSZYSTKIE TESTY ZOSTAŁY ZALICZONE
Liczba wykonanych testów: 3
$
```

Powinieneś mieć przynajmniej trzy testy potwierdzające poprawność wszystkich podstawowych operacji. Następnie przekonaj się, ile dodatkowych testów potrafisz utworzyć poza przygotowanymi przeze mnie.

Jak można usprawnić kod?

Jak zwykle dla utworzonego w ćwiczeniu kodu powinieneś spróbować zastosować strategie programowania defensywnego. Mam nadzieję, że się tym zajmiesz, ponieważ kod bazowy w większości biblioteki `liblcthw` nie został sprawdzony pod kątem strategii programowania defensywnego, które przedstawiłem wcześniej w książce. Dlatego też to zadanie pozostawiam Ci do wykonania, jako ćwiczenie mające na celu przyzwyczajenie się do poprawiania kodu za pomocą wspomnianych dodatkowych operacji zastosowania strategii programowania defensywnego.

Na przykład w kodzie bufora cyklicznego nie znajduje się zbyt wiele operacji sprawdzenia, czy żądany element faktycznie został umieszczony w buforze.

Jeżeli zapoznałeś się z zamieszczonym w Wikipedii artykułem poświęconym buforowi cyklicznemu, to powinieneś wiedzieć, że „zoptymalizowana implementacja POSIX" używa charakterystycznych dla **POSIX** (ang. *portable operating system interface*) wywołań w celu utworzenia nieskończonej przestrzeni. Przeanalizuj tę kwestię i zobacz, czy potrafisz wykonać przedstawione poniżej zadania dodatkowe.

Zadania dodatkowe

- Utwórz alternatywną implementację struktury `RingBuffer` używającą sztuczki POSIX, a następnie przygotuj dla niej test jednostkowy.
- Do testów jednostkowych dodaj test porównania wydajności. Za jego pomocą będziesz mógł przeprowadzić porównanie obu wersji implementacji przez dostarczenie im losowo wygenerowanych danych oraz wykonywanie losowych operacji odczytu i zapisu. Upewnij się, że dla każdej wersji są przeprowadzane te same operacje, aby móc dokonać wiarygodnego porównania przygotowanych implementacji.

ĆWICZENIE 45

Prosty klient TCP/IP

W tym ćwiczeniu strukturę danych `RingBuffer` wykorzystamy do utworzenia bardzo prostego narzędzia sieciowego o nazwie `netclient`. Wymaga to wprowadzenia zmian w pliku *Makefile* związanych z obsługą niewielkich programów umieszczanych w katalogu *bin*.

Modyfikacja pliku Makefile

Zaczynamy od dodania zmiennych dla programów, podobnie jak w przypadku zmiennych *TEST* i *TEST_SRC* dla testów jednostkowych:

```
PROGRAMS_SRC=$(wildcard bin/*.c)
PROGRAMS=$(patsubst %.c,%,$(PROGRAMS_SRC))
```

Następnie zmienną `PROGRAMS` dodajemy do celu `all`:

```
all: $(TARGET) $(SO_TARGET) tests $(PROGRAMS)
```

Zmienną `PROGRAMS` trzeba jeszcze dodać do polecenia `rm` w celu `clean`:

```
rm -rf build $(OBJECTS) $(TESTS) $(PROGRAMS)
```

Na końcu dodajemy cel przeznaczony do kompilacji całości:

```
$(PROGRAMS): CFLAGS += $(TARGET)
```

Po wprowadzeniu powyższych zmian wystarczy umieścić w katalogu *bin* plik z rozszerzeniem *.c* — zostanie on skompilowany i połączony z biblioteką, podobnie jak ma to miejsce w przypadku testów jednostkowych.

Kod netclient

Poniżej przedstawiłem kod źródłowy dla naszego niewielkiego narzędzia `netclient`.

Plik *netclient.c*:

```
 1 #undef NDEBUG
 2 #include <stdlib.h>
 3 #include <sys/select.h>
 4 #include <stdio.h>
 5 #include <lcthw/ringbuffer.h>
 6 #include <lcthw/dbg.h>
 7 #include <sys/socket.h>
 8 #include <sys/types.h>
 9 #include <sys/uio.h>
10 #include <arpa/inet.h>
```

```
11 #include <netdb.h>
12 #include <unistd.h>
13 #include <fcntl.h>
14
15 struct tagbstring NL = bsStatic("\n");
16 struct tagbstring CRLF = bsStatic("\r\n");
17
18 int nonblock(int fd)
19 {
20     int flags = fcntl(fd, F_GETFL, 0);
21     check(flags >= 0, "Nieprawidłowe opcje w nonblock.");
22
23     int rc = fcntl(fd, F_SETFL, flags | O_NONBLOCK);
24     check(rc == 0, "Nie można ustawić podejścia nieblokującego.");
25
26     return 0;
27 error:
28     return -1;
29 }
30
31 int client_connect(char *host, char *port)
32 {
33     int rc = 0;
34     struct addrinfo *addr = NULL;
35
36     rc = getaddrinfo(host, port, NULL, &addr);
37     check(rc == 0, "Nie udało się odnaleźć %s:%s", host, port);
38
39     int sock = socket(AF_INET, SOCK_STREAM, 0);
40     check(sock >= 0, "Nie udało się utworzyć gniazda.");
41
42     rc = connect(sock, addr->ai_addr, addr->ai_addrlen);
43     check(rc == 0, "Nie udało się nawiązać połączenia.");
44
45     rc = nonblock(sock);
46     check(rc == 0, "Nie udało się ustawić podejścia nieblokującego.");
47
48     freeaddrinfo(addr);
49     return sock;
50
51 error:
52     freeaddrinfo(addr);
53     return -1;
54 }
55
56 int read_some(RingBuffer * buffer, int fd, int is_socket)
57 {
58     int rc = 0;
59
60     if (RingBuffer_available_data(buffer) == 0) {
61         buffer->start = buffer->end = 0;
62     }
63
```

```
64      if (is_socket) {
65          rc = recv(fd, RingBuffer_starts_at(buffer),
66              RingBuffer_available_space(buffer), 0);
67      } else {
68          rc = read(fd, RingBuffer_starts_at(buffer),
69              RingBuffer_available_space(buffer));
70      }
71
72      check(rc >= 0, "Nie udało się odczytać z fd: %d", fd);
73
74      RingBuffer_commit_write(buffer, rc);
75
76      return rc;
77
78  error:
79      return -1;
80  }
81
82  int write_some(RingBuffer * buffer, int fd, int is_socket)
83  {
84      int rc = 0;
85      bstring data = RingBuffer_get_all(buffer);
86
87      check(data != NULL, "Nie udało się pobrać danych z bufora.");
88      check(bfindreplace(data, &NL, &CRLF, 0) == BSTR_OK,
89          "Nie udało się zastąpić NL.");
90
91      if (is_socket) {
92          rc = send(fd, bdata(data), blength(data), 0);
93      } else {
94          rc = write(fd, bdata(data), blength(data));
95      }
96
97      check(rc == blength(data), "Nie udało się zapisać wszystkiego do fd: %d.",
98          fd);
99      bdestroy(data);
100
101     return rc;
102
103 error:
104     return -1;
105 }
106
107 int main(int argc, char *argv[])
108 {
109     fd_set allreads;
110     fd_set readmask;
111
112     int socket = 0;
113     int rc = 0;
114     RingBuffer *in_rb = RingBuffer_create(1024 * 10);
115     RingBuffer *sock_rb = RingBuffer_create(1024 * 10);
116
117     check(argc == 3, "UŻYCIE: netclient host port");
```

```
118
119     socket = client_connect(argv[1], argv[2]);
120     check(socket >= 0, "Próba połączenia z %s:%s zakończyła się
        ↪niepowodzeniem.", argv[1], argv[2]);
121
122     FD_ZERO(&allreads);
123     FD_SET(socket, &allreads);
124     FD_SET(0, &allreads);
125
126     while (1) {
127         readmask = allreads;
128         rc = select(socket + 1, &readmask, NULL, NULL, NULL);
129         check(rc >= 0, "Operacja zakończyła się niepowodzeniem.");
130
131         if (FD_ISSET(0, &readmask)) {
132             rc = read_some(in_rb, 0, 0);
133             check_debug(rc != -1, "Nie udało się odczytać z stdin.");
134         }
135
136         if (FD_ISSET(socket, &readmask)) {
137             rc = read_some(sock_rb, socket, 0);
138             check_debug(rc != -1, "Nie udało się odczytać z socket.");
139         }
140
141         while (!RingBuffer_empty(sock_rb)) {
142             rc = write_some(sock_rb, 1, 0);
143             check_debug(rc != -1, "Nie udało się zapisać do stdout.");
144         }
145
146         while (!RingBuffer_empty(in_rb)) {
147             rc = write_some(in_rb, socket, 1);
148             check_debug(rc != -1, "Nie udało się zapisać do socket.");
149         }
150     }
151
152     return 0;
153
154 error:
155     return -1;
156 }
```

W powyższym kodzie używamy wywołania `select()` do obsługi zdarzeń pochodzących zarówno z `stdin` (deskryptor pliku 0), jak i `socket` (gniazdo), stosowanych podczas komunikacji z serwerem. Implementacja wykorzystuje strukturę danych `RingBuffer` do przechowywania i kopiowania danych. Funkcje `read_some()` i `write_some()` możesz potraktować jako wczesne prototypy podobnych funkcji w strukturze `RingBuffer`.

Ten niewielki fragment kodu zawiera całkiem sporą liczbę funkcji sieciowych, których możesz nawet nie znać. Gdy natkniesz się na tego rodzaju nieznane funkcje, poszukaj w podręczniku systemowym `man` informacji o nich i koniecznie upewnij się, że rozumiesz ich działanie. Ten niewielki plik może zainspirować Cię do poszukania informacji o innych API niezbędnych do utworzenia w języku C małego serwera.

Co powinieneś zobaczyć?

Po przeprowadzeniu kompilacji niezbędnych plików najszybszym sposobem pozwalającym na przetestowanie kodu jest sprawdzenie, czy możesz otrzymać plik specjalny z witryny *http://learncodethehardway.org/*.

Sesja dla ćwiczenia 45.:

```
$
$ ./bin/netclient learncodethehardway.org 80
GET /ex45.txt HTTP/1.1
Host: learncodethehardway.org
HTTP/1.1 200 OK

Date: Fri, 27 Apr 2012 00:41:25 GMT
Content-Type: text/plain
Content-Length: 41
Last-Modified: Fri, 27 Apr 2012 00:42:11 GMT
ETag: 4f99eb63-29
Server: Mongrel2/1.7.5

Learn C The Hard Way, kod z ćwiczenia 45. działa.
^C
$
```

Zapis powyższej sesji pokazuje składnię niezbędną w celu wykonania żądania HTTP do pliku */ex45.txt*, wiersz `Host:` wskazujący adres hosta i pusty wiersz. W dalszej części widzimy otrzymaną odpowiedź wraz z nagłówkami i treścią. Na końcu nacisnąłem klawisze *Ctrl+C*, aby zakończyć działanie programu.

Jak to zepsuć?

Omówiony tutaj kod na pewno zawiera błędy i będę kontynuował prace nad jego usprawnieniem. W międzyczasie spróbuj przeanalizować przedstawiony kod i wykorzystaj go do pracy z innymi serwerami. Istnieje narzędzie o nazwie `netcat` doskonale sprawdzające się podczas przygotowywania wspomnianych serwerów. Innym zadaniem jest użycie języka takiego jak Python lub Ruby w celu utworzenia prostego serwera udostępniającego śmieci i nieprawidłowe dane, losowo zrywającego połączenia i ogólnie wykonującego różne nieprzyjemne rzeczy.

Jeżeli znajdziesz błędy w kodzie, poinformuj mnie o nich, co pozwoli mi na ich usunięcie.

Zadania dodatkowe

- Jak wcześniej wspomniałem, kod zawiera kilka funkcji, których możesz nie znać, więc poszukaj informacji na ich temat. Tak naprawdę powinieneś poszukać informacji o wszystkich funkcjach, nawet tylko o tych, o których sądzisz, że je znasz.

- Uruchom kod za pomocą debugera i poszukaj błędów.
- Względem funkcji zastosuj różne strategie programowania defensywnego i operacje sprawdzenia, aby tym samym je usprawnić.
- Użyj funkcji `getopt()`, aby dać użytkownikowi możliwość *rezygnacji* z przeprowadzania konwersji znaku `\n` na `\r\n`. Tego rodzaju konwersja jest potrzebna jedynie w protokołach wymagających podanego sposobu zakończenia wiersza, na przykład HTTP. Czasami nie chcemy przeprowadzać konwersji, więc należy dać użytkownikowi możliwość wyboru.

ĆWICZENIE 46

Drzewo trójkowe

Ostatnią strukturą danych, którą zamierzam Ci pokazać, jest tak zwane *drzewo trójkowe* (ang. *ternary search tree*), które jest podobne do omówionego wcześniej drzewa binarnego, z wyjątkiem tego, że ma trzy gałęzie: `low`, `equal` i `high`. Podstawowy sposób użycia jest podobny jak sposób użycia drzewa binarnego, struktura `Hashmap` służy do przechowywania danych w postaci klucz-wartość, natomiast samo drzewo działa z poszczególnymi znakami w kluczach. Dzięki temu drzewo trójkowe zyskuje możliwości niedostępne dla drzewa binarnego i struktury `Hashmap`.

W strukturze drzewa trójkowego (`TSTree`) każdy klucz jest ciągiem tekstowym, a jego wstawienie odbywa się przez przejście i zbudowanie drzewa na podstawie wyniku sprawdzenia równości znaków w ciągu tekstowym. Operacja zaczyna się od węzła głównego, gdzie jest szukany znak dla tego węzła. Gdy jest mniejszy, równy lub większy, operacja przechodzi w odpowiednim kierunku. Możesz to zobaczyć w przedstawionym poniżej pliku nagłówkowym.

Plik *tstree.h*:

```
#ifndef _lcthw_TSTree_h
#define _lcthw_TSTree_h

#include <stdlib.h>
#include <lcthw/darray.h>

typedef struct TSTree {
    char splitchar;
    struct TSTree *low;
    struct TSTree *equal;
    struct TSTree *high;
    void *value;
} TSTree;

void *TSTree_search(TSTree * root, const char *key, size_t len);

void *TSTree_search_prefix(TSTree * root, const char *key, size_t len);

typedef void (*TSTree_traverse_cb) (void *value, void *data);

TSTree *TSTree_insert(TSTree * node, const char *key, size_t len,
    void *value);

void TSTree_traverse(TSTree * node, TSTree_traverse_cb cb, void *data);

void TSTree_destroy(TSTree * root);

#endif
```

Struktura TSTree zawiera następujące elementy:

splitchar. Znak w danym miejscu drzewa.

low. Gałąź niższa niż wskazana w splitchar.

equal. Gałąź równa wskazanej w splitchar.

high. Gałąź wyższa niż wskazana w splitchar.

value. Wartość przypisana ciągowi tekstowemu w miejscu wskazywanym przez splitchar.

Można zauważyć, że przedstawiona implementacja zawiera obsługę następujących operacji:

search(). Typowa operacja wyszukiwania wartości dla podanego klucza.

search_index(). Ta operacja powoduje odszukanie pierwszej wartości mającej dany wzorzec jako prefiks klucza. Tego rodzaju operacji nie można w łatwy sposób przeprowadzić w strukturze danych BSTree lub Hashmap.

insert(). Podział klucza na poszczególne znaki i wstawienie ich w drzewie.

traverse(). Przejście przez drzewo, zebranie lub przeanalizowanie wszystkich kluczy oraz znajdujących się w nich wartości.

Jedną brakującą operacją jest tutaj TSTree_delete(), ponieważ jest niezwykle kosztowna, jeszcze bardziej niż w przypadku BSTree_delete(). Kiedy używam struktur TSTree, traktuję je jako stałe dane, po których planuję wielokrotnie się poruszać i w ogóle ich nie usuwać. Operacje poruszania się po danych są bardzo szybkie, natomiast wstawienie i usuwanie danych odbywa się niezwykle wolno. Dlatego też w przypadkach wymagających wstawiania i usuwania danych wybieram strukturę Hashmap, która sprawdza się wtedy znacznie lepiej niż BSTree i TSTree.

Sama implementacja struktury TSTree jest tak naprawdę prosta, choć na początku może okazać się trudna do zrozumienia. Najpierw wprowadź implementację do pliku, a następnie dokładnie ją omówimy.

Plik *tstree.c*:

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <lcthw/dbg.h>
#include <lcthw/tstree.h>

static inline TSTree *TSTree_insert_base(TSTree * root, TSTree * node,
        const char *key, size_t len,
        void *value)
{
    if (node == NULL) {
        node = (TSTree *) calloc(1, sizeof(TSTree));

        if (root == NULL) {
            root = node;
```

```
16          }
17
18          node->splitchar = *key;
19      }
20
21      if (*key < node->splitchar) {
22          node->low = TSTree_insert_base(
23              root, node->low, key, len, value);
24      } else if (*key == node->splitchar) {
25          if (len > 1) {
26              node->equal = TSTree_insert_base(
27                  root, node->equal, key + 1, len - 1, value);
28          } else {
29              assert(node->value == NULL && "Próba wstawienia duplikatu do tst.");
30                  node->value = value;
31          }
32      } else {
33          node->high = TSTree_insert_base(
34              root, node->high, key, len, valueP
35      }
36
37      return node;
38  }
39
40  TSTree *TSTree_insert(TSTree * node, const char *key, size_t len,
41          void *valueP
42  {
43      return TSTree_insert_base(node, node, key, len, value);
44  }
45
46  void *TSTree_search(TSTree * root, const char *key, size_t len)
47  {
48      TSTree *node = root;
49      size_t i = 0;
50
51      while (i < len && node) {
52          if (key[i] < node->splitchar) {
53              node = node->low;
54          } else if (key[i] == node->splitchar) {
55              i++;
56              if (i < len)
57                  node = node->equal;
58          } else {
59              node = node->high;
60          }
61      }
62
63      if (node) {
64          return node->value;
65      } else {
66          return NULL;
67      }
68  }
```

```
 69
 70 void *TSTree_search_prefix(TSTree * root, const char *key, size_t len)
 71 {
 72     if (len == 0)
 73         return NULL;
 74
 75     TSTree *node = root;
 76     TSTree *last = NULL;
 77     size_t i = 0;
 78
 79     while (i < len && node) {
 80         if (key[i] < node->splitchar) {
 81             node = node->low;
 82         } else if (key[i] == node->splitchar) {
 83             i++;
 84             if (i < len) {
 85                 if (node->value)
 86                     last = node;
 87                 node = node->equal;
 88             }
 89          } else {
 90             node = node->high;
 91         }
 92     }
 93
 94     node = node ? node : last;
 95
 96     // Poruszamy się aż do znalezienia pierwszej wartości w łańcuchu,
 97     // która następnie będzie pierwszym węzłem o danym prefiksie.
 98     while (node && !node->value) {
 99         node = node->equal;
100     }
101
102     return node ? node->value : NULL;
103 }
104
105 void TSTree_traverse(TSTree * node, TSTree_traverse_cb cb, void *data)
106 {
107     if (!node)
108         return;
109
110     if (node->lowP
111         TSTree_traverse(node->low, cb, data);
112
113     if (node->equal) {
114         TSTree_traverse(node->equal, cb, data);
115     }
116
117     if (node->high)
118         TSTree_traverse(node->high, cb, data);
119
120     if (node->value)
121         cb(node->value, data);
```

```
122 }
123
124 void TSTree_destroy(TSTree * node)
125 {
126     if (node == NULL)
127         return;
128
129     if (node->low)
130         TSTree_destroy(node->low);
131
132     if (node->equal) {
133          TSTree_destroy(node->equal);
134     }
135
136     if (node->high)
137         TSTree_destroy(node->high);
138
139     free(node);
140 }
```

W przypadku funkcji `TSTree_insert()` używamy takiego samego wzorca dla struktur rekurencyjnych, czyli definiujemy małą funkcję wywołującą rzeczywistą funkcję rekurencyjną. Nie przeprowadzamy w tym miejscu żadnych dodatkowych sprawdzeń, choć powinieneś zastosować strategie programowania defensywnego. Warto pamiętać, że w omawianej strukturze używam nieco innego projektu, nieopierającego się na oddzielnej funkcji `TSTree_create()`. Jednak po przekazaniu wartości `NULL` dla węzła następuje utworzenie i zwrot ostatecznej wartości.

Oznacza to konieczność dokładnego omówienia funkcji `TSTree_insert_base()`, aby pokazać, jak przeprowadzana jest operacja wstawienia danych.

tstree.c:10 – 19. Jak wcześniej wspomniałem, po przekazaniu wartości `NULL` konieczne jest utworzenie węzła i przypisanie mu `*key` (bieżący znak). W ten sposób budujemy drzewo podczas wstawiania kluczy.

tstree.c:21 – 22. Jeżeli klucz `*key` jest mniejszy niż dany, to przystępujemy do rekurencji, ale za pomocą gałęzi `low`.

tstree.c:24. W tym przypadku klucz `*key` jest równy wartości wskazywanej przez `splitchar`. Tego rodzaju sytuacja zdarza się tuż po utworzeniu węzła, czyli na tym etapie budujemy drzewo.

tstree.c:26 – 27. Nadal pozostały znaki do obsłużenia, przystępujemy do rekurencji w dół gałęzi `equal`, ale do następnego znaku `*key`.

tstree.c:29 – 30. To jest ostatni znak, przypisujemy wartość, i na tym koniec. Mamy przygotowane wywołanie `assert()` na wypadek istnienia duplikatu.

tstree.c:33 – 34. Ostatni warunek to klucz `*key` większy niż wskazywana przez `splitchar`, a więc trzeba przeprowadzić rekurencję w dół gałęzi `high`.

Dla omawianej struktury kluczowe znaczenie ma fakt, że inkrementacja znaku następuje tylko wtedy, gdy dopasowano wartość wskazywaną przez `splitchar`. W dwóch pozostałych warunkach po prostu poruszamy się po drzewie aż do napotkania takiego samego znaku pozwala-

jącego na przeprowadzenie rekurencji. Oznacza to bardzo szybkie działanie w przypadku *nieznalezienia* klucza. Po otrzymaniu niewłaściwego klucza można przejść przez kilka gałęzi `high` i `low`, aż do dotarcia do ślepego zaułku i ustalenia, że dany klucz nie istnieje. Nie ma konieczności przetwarzania każdego znaku klucza lub każdego węzła drzewa.

Po zrozumieniu powyższego mechanizmu możemy przejść do analizy sposobu działania funkcji `TSTree_search()`:

tstree.c:51. W strukturze danych `TSTree` nie ma konieczności rekurencyjnego przetwarzania drzewa. Wystarczy po prostu użyć pętli `while` i węzła bieżącego.

tstree.c:52 – 53. Jeżeli bieżący znak jest mniejszy niż wartość `splitchar` węzła, to poruszamy się w dół.

tstree.c:54 – 57. W przypadku równości inkrementujemy wartość `i`, a następnie przechodzimy gałęzią `equal`, o ile nie mamy do czynienia z ostatnim znakiem. Z tego powodu mamy polecenie `if(i < len)`, aby nie trzeba było cofać się zbyt daleko od ostatecznej wartości.

tstree.c:58 – 59. W przeciwnym razie, gdy znak jest większy niż wartość `splitchar` węzła, poruszamy się gałęzią `high`.

tstree.c:63 – 67. Jeżeli po wykonaniu pętli mamy węzeł, to zwracamy jego wartość. W przeciwnym razie wartością zwrotną jest `NULL`.

Przedstawiony powyżej mechanizm nie jest zbyt trudny do zrozumienia, a praktycznie ten sam algorytm zastosowaliśmy w funkcji `TSTree_search_prefix()`. Jedyna różnica polega na tym, że nie staramy się znaleźć dokładnego dopasowania, ale jak najdłuższy prefiks. W tym celu monitorujemy ostatni węzeł określony jako równy, a później po wykonaniu pętli wyszukiwania przechodzimy przez ten węzeł aż do znalezienia wartości.

Analizując funkcję `TSTree_search_prefix()`, zaczniesz dostrzegać kolejną przewagę struktury `TSTree` nad `BSTree` i `Hashmap` w zakresie wyszukiwania ciągów tekstowych. Mając klucz o wielkości *X*, można znaleźć dowolny klucz w *X* ruchach. Ponadto można znaleźć pierwszy prefiks w *X* ruchach plus *N* dodatkowych, w zależności od wielkości dopasowanego klucza. Jeżeli największy klucz drzewa ma dziesięć znaków, to w dziesięciu ruchach można znaleźć dowolny prefiks w tym kluczu. Co ważniejsze, wszystkie tego rodzaju operacje przeprowadzasz przez *jednokrotne* porównanie każdego znaku z kluczem.

Dla porównania ta sama operacja dla struktury `BSTree` wymaga sprawdzenia prefiksów dla każdego znaku w każdym możliwym dopasowanym węźle struktury `BSTree` względem znaków prefiksu. To samo dotyczy również wyszukiwania lub sprawdzania, czy klucz istnieje. Konieczne jest porównanie każdego znaku względem większości znaków w `BSTree`, aby dowiedzieć się, czy istnieje dopasowanie.

Struktura `Hashmap` sprawdza się jeszcze gorzej podczas wyszukiwania prefiksów, ponieważ nie można wygenerować wartości hash dla jedynie prefiksu. W praktyce wyszukiwania prefiksów nie można efektywnie przeprowadzić w strukturze `Hashmap`, o ile dane nie mają postaci możliwej do przetworzenia, na przykład adres URL. Jednak nawet wtedy wymagane jest tworzenie całych drzew struktur `Hashmap`.

Ostatnie dwie funkcje powinny być łatwe do przeanalizowania, ponieważ odpowiadają za typowe operacje poruszania się po węzłach i niszczenia, z którymi miałeś już styczność w innych strukturach danych.

Przechodzimy teraz do prostych testów jednostkowych, dzięki którym możemy sprawdzić, czy całość działa zgodnie z oczekiwaniami.

Plik *tstree_tests.c*:

```
#include "minunit.h"
#include <lcthw/tstree.h>
#include <string.h>
#include <assert.h>
#include <lcthw/bstrlib.h>

TSTree *node = NULL;
char *valueA = "WARTOŚĆA";
char *valueB = "WARTOŚĆB";
char *value2 = "WARTOŚĆ2";
char *value4 = "WARTOŚĆ4";
char *reverse = "WARTOŚĆR";
int traverse_count = 0;

struct tagbstring test1 = bsStatic("TEST");
struct tagbstring test2 = bsStatic("TEST2");
struct tagbstring test3 = bsStatic("TSET");
struct tagbstring test4 = bsStatic("T");

char *test_insert()
{
    node = TSTree_insert(node, bdata(&test1), blength(&test1), valueA);
    mu_assert(node != NULL, "Nie udało się wstawić do tst.");

    node = TSTree_insert(node, bdata(&test2), blength(&test2), value2);
    mu_assert(node != NULL,
        "Nie udało się wstawić do tst drugiej nazwy.");

    node = TSTree_insert(node, bdata(&test3), blength(&test3), reverse);
    mu_assert(node != NULL,
        "Nie udało się wstawić do tst odwróconej nazwy.");

    node = TSTree_insert(node, bdata(&test4), blength(&test4), value4);
    mu_assert(node != NULL,
        "Nie udało się wstawić do tst drugiej nazwy.");

    return NULL;
}

char *test_search_exact()
{
    // tst zwraca ostatni wstawiony węzeł.
    void *res = TSTree_search(node, bdata(&test1), blength(&test1));
```

```
44      mu_assert(res == valueA,
45          "Otrzymano nieprawidłową wartość, powinno być A zamiast B.");
46
47      // tst nie znajdzie węzła, jeśli nie będzie dokładnie taki, jak podano.
48      res = TSTree_search(node, "TESTNO", strlen("TESTNO"));
49      mu_assert(res == NULL, "Nie powinno być żadnej wartości.");
50
51      return NULL;
52  }
53
54  char *test_search_prefix()
55  {
56      void *res = TSTree_search_prefix(
57          node, bdata(&test1), blength(&test1));
58      debug("wynik: %p, oczekiwano: %p", res, valueA);
59      mu_assert(res == valueA, "Otrzymano nieprawidłową wartość valueA według
        ↪prefiksu.");
60
61      res = TSTree_search_prefix(node, bdata(&test1), 1);
62      debug("wynik: %p, oczekiwano: %p", res, valueA);
63      mu_assert(res == value4, "Otrzymano nieprawidłową wartość value4
        ↪dla prefiksu 1.");
64
65      res = TSTree_search_prefix(node, "TE", strlen("TE"));
66      mu_assert(res != NULL, "Nie powinno być nic znalezione dla krótkiego
        ↪prefiksu.");
67
68      res = TSTree_search_prefix(node, "TE--", strlen("TE--"));
69      mu_assert(res != NULL, "Nie powinno być nic znalezione dla częściowego
        ↪prefiksu.");
70
71      return NULL;
72  }
73
74  void TSTree_traverse_test_cb(void *value, void *data)
75  {
76      assert(value != NULL && "Nie powinna być zwrócona wartość NULL.");
77      assert(data == valueA && "Oczekiwano wartości valueA jako danych.");
78      traverse_count++;
79  }
80
81  char *test_traverse()
82  {
83      traverse_count = 0;
84      TSTree_traverse(node, TSTree_traverse_test_cb, valueA);
85      debug("Liczba przejść wynosi: %d", traverse_count);
86      mu_assert(traverse_count == 4, "Nie znaleziono 4 kluczy.");
87
88      return NULL;
89  }
90
91  char *test_destroy()
92  {
```

```
 93     TSTree_destroy(node);
 94
 95     return NULL;
 96 }
 97
 98 char *all_tests()
 99 {
100     mu_suite_start();
101
102     mu_run_test(test_insert);
103     mu_run_test(test_search_exact);
104     mu_run_test(test_search_prefix);
105     mu_run_test(test_traverse);
106     mu_run_test(test_destroy);
107
108     return NULL;
109 }
110
111 RUN_TESTS(all_tests);
```

Wady i zalety

Istnieje wiele innych interesujących i praktycznych rzeczy, które można zrobić za pomocą struktury `TSTree`:

- Poza wyszukiwaniem prefiksów można odwrócić wszystkie wstawiane klucze, a następnie wyszukiwać elementy za pomocą *sufiksów*. Takie rozwiązanie stosuję podczas wyszukiwania nazw hostów, ponieważ chcę znaleźć na przykład `*.learn ↪codethehardway.com`. Jeżeli zastosuję odwrotność, to szybko mogę znaleźć dopasowanie.
- Przeprowadzenie przybliżonego dopasowania przez zebranie wszystkich węzłów zawierających większość takich samych znaków, jakie znajdują się w kluczu. Ewentualnie za pomocą innych algorytmów przybliżonego dopasowania.
- Wyszukanie wszystkich kluczy na podstawie pewnych znaków w środku klucza.

Przedstawiłem już pewne zalety `TSTree`, ale nie powinieneś mieć wątpliwości, że to nie jest najlepsza struktura danych na świecie. Oto niektóre jej wady:

- Jak wcześniej wspomniałem, usuwanie danych z `TSTree` jest mordęgą. Omawiana struktura znacznie lepiej sprawdza się w przypadku danych wymagających ich częstego wyszukiwania i naprawdę rzadkiego usuwania. Jeżeli potrzebujesz możliwość usuwania danych, to po prostu blokuj wartość, a następnie okresowo ponownie utwórz drzewo, gdy stanie się zbyt duże.
- W porównaniu z `BSTree` i `Hashmap` używa ogromnej ilości pamięci dla tej samej przestrzeni kluczy. Pomyśl o tym. Struktura używa pełnego węzła dla każdego znaku we wszystkich kluczach. Może się sprawdzać dobrze dla małych kluczy, ale po umieszczeniu ich wielu w `TSTree`, struktura staje się naprawdę duża.

- Nie działa zbyt dobrze z ogromnymi kluczami, choć trzeba w tym miejscu dodać, że określenie „ogromny" jest bardzo subiektywne. Jak zwykle najlepiej najpierw przeprowadź odpowiednie testy. Jeżeli masz potrzebę przechowywania kluczy o wielkości 10 000 znaków, wybierz strukturę danych `Hashmap`.

Jak można usprawnić kod?

Jak zwykle przeanalizuj kod i spróbuj usprawnić go przez zastosowanie strategii programowania defensywnego, warunków początkowych, asercji i sprawdzeń wszystkich funkcji. Mamy jeszcze inne możliwe usprawnienia, ale naprawdę nie musisz implementować wszystkich wymienionych poniżej:

- Zezwolenie na występowanie duplikatów dzięki użyciu struktury `DArray` zamiast `value`.
- Jak wcześniej wspomniałem, usuwanie jest prawdziwą udręką. Można je jednak zasymulować przez przypisanie wartości `NULL`, co w efekcie oznacza pozbycie się przechowywanej wartości.
- Nie ma sposobu na zebranie wszystkich możliwych dopasowanych wartości. Implementację tej funkcjonalności zlecę Ci w zadaniach dodatkowych.
- Wprawdzie są jeszcze inne znacznie bardziej skomplikowane algorytmy, ale charakteryzują się tylko nieznacznie lepszymi właściwościami. Poszukaj informacji o następujących strukturach: tablica sufiksowa, drzewo sufiksowe i skompresowane drzewo trie (również drzewo Patricia, drzewo pozycyjne).

Zadania dodatkowe

- Zaimplementuj funkcję `TSTree_collect()` zwracającą strukturę `DArray` wraz z wszystkimi kluczami dopasowanymi do danego prefiksu.
- Zaimplementuj funkcje `TSTree_search_suffix()` i `TSTree_insert_suffix()`, aby mieć możliwość wyszukiwania i wstawiania na podstawie sufiksu.
- Użyj debugera w celu sprawdzenia, jak omawiana tutaj struktura danych przechowuje dane w porównaniu ze strukturami `BSTree` i `Hashmap`.

ĆWICZENIE 47

Szybszy router URL

W tym ćwiczeniu pokażę, jak używam struktury danych TSTree do szybkiego routingu URL w utworzonych przeze mnie serwerach WWW. Przedstawione rozwiązanie sprawdza się w przypadku prostego routingu URL, z którego możesz korzystać sporadycznie w aplikacji, ale nie nadaje się do bardziej zaawansowanego (i czasami niepotrzebnego) routingu oferowanego przez wiele frameworków aplikacji sieciowych.

Aby poeksperymentować z routingiem utworzymy niewielkie narzędzie powłoki o nazwie urlor, które będzie odczytywać prosty plik z zapisanymi trasami, a następnie poprosi użytkownika o podanie adresów URL.

Plik *urlot.c*:

```
#include <lcthw/tstree.h>
#include <lcthw/bstrlib.h>

TSTree *add_route_data(TSTree * routes, bstring line)
{
    struct bstrList *data = bsplit(line, ' ');
    check(data->qty == 2, "Wiersz '%s' nie zawiera 2 kolumn.",
        bdata(line));

    routes = TSTree_insert(routes,
        bdata(data->entry[0]),
        blength(data->entry[0]),
        bstrcpy(data->entry[1]));

    bstrListDestroy(data);

    return routes;

error:
    return NULL;
}

TSTree *load_routes(const char *file)
{
    TSTree *routes = NULL;
    bstring line = NULL;
    FILE *routes_map = NULL;

    routes_map = fopen(file, "r");
    check(routes_map != NULL, "Nie udało się otworzyć tras: %s", file);

    while ((line = bgets((bNgetc) fgetc, routes_map, '\n')) != NULL) {
        check(btrimws(line) == BSTR_OK, "Nie udało się skrócić wiersza.");
        routes = add_route_data(routes, line);
        check(routes != NULL, "Nie udało się dodać trasy.");
```

```
36          bdestroy(line);
37      }
38
39      fclose(routes_map);
40      return routes;
41
42  error:
43      if (routes_map) fclose(routes_map);
44      if (line) bdestroy(line);
45
46      return NULL;
47  }
48
49  bstring match_url(TSTree * routes, bstring url)
50  {
51      bstring route = TSTree_search(routes, bdata(url), blength(url));
52
53      if (route == NULL) {
54          printf("Brak dokładnego dopasowania, próbuję prefiks.\n");
55          route = TSTree_search_prefix(routes, bdata(url), blength(url));
56      }
57
58      return route;
59  }
60
61  bstring read_line(const char *prompt)
62  {
63      printf("%s", prompt);
64
65      bstring result = bgets((bNgetc) fgetc, stdin, '\n');
66      check_debug(result != NULL, "Zamknięte standardowe wejście.");
67
68      check(btrimws(result) == BSTR_OK, "Nie udało się skrócić.");
69
70      return result;
71
72  error:
73      return NULL;
74  }
75
76  void bdestroy_cb(void *value, void *ignored)
77  {
78      (void)ignored;
79      bdestroy((bstring) value);
80  }
81
82  void destroy_routes(TSTree * routesP
83  {
84      TSTree_traverse(routes, bdestroy_cb, NULL);
85      TSTree_destroy(routes);
86  }
87
88  int main(int argc, char *argv[])
89  {
```

```
    bstring url = NULL;
    bstring route = NULL;
    TSTree *routes = NULL;

    check(argc == 2, "UŻYCIE: urlor <urlfile>");

    routes = load_routes(argv[1]);
    check(routes != NULL, "Plik tras zawiera błąd.");

    while (1) {
        url = read_line("URL> ");
        check_debug(url != NULL, "do widzenia.");

        route = match_url(routes, url);

        if (route) {
            printf("DOPASOWANIE: %s == %s\n", bdata(url), bdata(route));
        } else {
            printf("NIEPOWODZENIE: %s\n", bdata(url));
        }

        bdestroy(url);
    }

    destroy_routes(routes);
    return 0;

error:
    destroy_routes(routes);
    return 1;
}
```

Kolejnym krokiem jest przygotowanie przykładowego pliku zawierającego trasy do wypróbowania:

```
/ MainApp
/hello Hello
/hello/ Hello
/signup Signup
/logout Logout
/album/ Album
```

Co powinieneś zobaczyć?

Po uruchomieniu narzędzia `urlor` i podaniu pliku z trasami możesz wreszcie wypróbować jego działanie.

Sesja dla ćwiczenia 47.:

```
$ ./bin/urlor ex47_urls.txt
URL> /
DOPASOWANIE: / == IndexHandler
```

```
URL> asdfasf
Brak dokładnego dopasowania, próbuję prefiks.
NIEPOWODZENIE: asdfasf
URL> /test
Brak dokładnego dopasowania, próbuję prefiks.
DOPASOWANIE: /test == TestHandler
URL> /test/
Brak dokładnego dopasowania, próbuję prefiks.
DOPASOWANIE: /test == PageHandler
URL>
$
```

Jak możesz zobaczyć, system routingu najpierw próbuje znaleźć dokładne dopasowanie i jeśli to się nie uda, to próbuje dopasować prefiks. Zastosowałem takie podejście, aby pokazać różnice między tymi dwoma operacjami. W zależności od semantyki adresów URL oczekiwania mogą być różne: zawsze dokładne dopasowanie, zawsze dopasowanie prefiksów lub oba podejścia i wybór najlepszego.

Jak można usprawnić kod?

Adresy URL są dziwne, ponieważ użytkownicy oczekują od nich magicznej obsługi wszystkich szalonych rzeczy, jakie mogą zrobić za pomocą aplikacji sieciowych, nawet jeśli to nie będzie zbyt logiczne. W tej najprostszej prezentacji sposobu użycia struktury `TSTree` do obsługi routingu istnieją błędy, których użytkownicy nie będą w stanie wyrazić. Na przykład `TSTree` dopasuje wyrażenie `/al` do `Album`, co ogólnie rzecz biorąc, jest niepożądane. Użytkownik chce, aby wyrażenie `/album/*` spowodowało dopasowanie `Album`, natomiast `/al` spowodowało wygenerowanie błędu 404.

To na szczęście nie jest trudne do zaimplementowania, ponieważ można zmienić algorytm prefiksu tak, aby dopasowanie odbywało się w oczekiwany sposób. Jeżeli zmienisz algorytm dopasowania w taki sposób, aby wyszukiwał *wszystkie* pasujące prefiksy, a następnie wybierał najlepszy, zadanie modyfikacji aplikacji okaże się łatwe. W takim przypadku wyrażenie `/al` może dopasować `MainApp` lub `Album`. Wykorzystaj otrzymane wyniki, a następnie zmień nieco logikę w celu ustalenia lepszego podejścia.

Kolejną zmianą, którą można wprowadzić, przygotowując rzeczywisty system routingu, jest użycie struktury `TSTree` do wyszukania wszystkich możliwych dopasowań, które będą miały postać małego zbioru wzorców do sprawdzenia. W wielu aplikacjach sieciowych istnieje lista wyrażeń regularnych (określanych również mianem *regex*), do których w trakcie każdego żądania mają być dopasowane adresy URL. Wykonanie tych wszystkich wyrażeń regularnych może być czasochłonne i dlatego możesz użyć struktury `TSTree` do znalezienia wszystkich możliwych dopasowań według ich prefiksów. W ten sposób bardzo zawężasz wyszukiwanie do zaledwie kilku liczbę wzorców koniecznych do wypróbowania.

Za pomocą przedstawionej metody adresy URL będą dopasowywane dokładnie, ponieważ tak naprawdę używamy prawdziwych wzorców wyrażeń regularnych, które znajdują dopasowania znacznie szybciej z powodu wyszukiwania ich przez możliwe prefiksy.

Tego rodzaju algorytm sprawdza się także w innych sytuacjach, gdy potrzebny jest elastyczny mechanizm routingu widocznego dla użytkownika: nazwy domen, adresy IP, rejestry i katalogi, pliki i adresy URL.

Zadania dodatkowe

- Zamiast przechowywać jedynie ciąg tekstowy dla procedury obsługi, utwórz rzeczywisty silnik używający struktury `Handler` do przechowywania aplikacji. Tego rodzaju struktura będzie przechowywała adres URL, do którego jest dołączona, nazwę oraz wszystko inne, co jest potrzebne do utworzenia rzeczywistego systemu routingu.
- Zamiast mapować adresy URL na dowolne nazwy, zastosuj mapowanie na pliki *.so*, a następnie wykorzystaj wywołania `dlopen()` do wczytywania procedur obsługi w locie i wykonywania zdefiniowanych w nich wywołań zwrotnych. Wywołania zwrotne umieść w strukturze `Handler`. Tym samym otrzymujesz utworzony w języku C w pełni dynamiczny system obsługi wywołań zwrotnych.

ĆWICZENIE 48

Prosty serwer sieciowy

Dotarliśmy do tej części książki, w której będziesz pracował nad bardziej zaawansowanymi projektami, wymagającymi poświęcenia większej ilości czasu. Ostatnie pięć ćwiczeń dotyczy problemu utworzenia prostego serwera sieciowego w podobny sposób, jak w przypadku programu `logfind`. Przedstawię opis poszczególnych faz projektu, a Twoim zadaniem będzie wykonanie określonych działań. Zanim przejdziesz do kolejnej fazy, porównaj utworzony przez siebie kod z przygotowanym przeze mnie.

Wspomniane opisy są celowo ogólne, aby pozostawić Ci wolną rękę w zakresie próby rozwiązania danego problemu, choć nadal będę starał się pomagać. Dla każdego ćwiczenia przygotowałem *dwa* klipy wideo. Pierwszy pokazuje sposób działania projektu dla danego ćwiczenia, co pozwoli Ci zobaczyć program w działaniu i podjąć próbę jego emulacji. Z kolei w drugim pokazuję, jak *ja* rozwiązałem dany problem. W ten sposób będziesz mógł porównać opracowane przez siebie rozwiązanie z moim. Na koniec podam adres repozytorium projektu w serwisie GitHub, co pozwoli Ci na zapoznanie się z rzeczywistym kodem.

Najpierw powinieneś spróbować samodzielnie rozwiązać dany problem. Kiedy będziesz miał już działającą wersję programu (lub całkowicie utkniesz i nie potrafisz ruszyć dalej), wtedy obejrzyj drugi klip wideo i spójrz na przygotowany przeze mnie kod. Gdy zakończysz pracę nad danym etapem, możesz przejść do następnego i kontynuować pracę z własnym kodem lub wykorzystać mój dla danego etapu.

Specyfikacja

W pierwszym małym programie przygotujesz fundamenty dla pozostałych faz projektu. Programowi możesz nadać nazwę `statserve`, nawet pomimo tego, że specyfikacja nie zawiera wzmianki o danych statystycznych. Do tego dojdziemy później.

Specyfikacja budowanego projektu jest bardzo prosta:

1. Utwórz prosty serwer sieciowy akceptujący połączenia na porcie 7899 pochodzące od klienta `netclient` lub `nc`, a następnie wyświetlający wszystko to, co zostanie wpisane przez użytkownika.
2. Musisz dowiedzieć się, jak dołączyć do portu, nasłuchiwać na gnieździe i udzielać odpowiedzi na żądania. Wykorzystaj umiejętności w zakresie wyszukiwania informacji do sprawdzenia, jak zostały zbudowane inne rozwiązania tego typu, a następnie podejmij próbę zaimplementowania własnego.
3. Ważnym etapem w tym projekcie jest przygotowanie katalogu na podstawie szkieletu *c-skeleton* i upewnienie się o możliwości kompilacji wszystkich plików oraz prawidłowym działaniu całości.
4. Nie przejmuj się komponentami takimi jak demony itd. Twój serwer ma być uruchamiany z poziomu powłoki i po prostu działać.

Największym wyzwaniem w tym projekcie jest ustalenie, jak utworzyć gniazdo serwera. Jednak dzięki zdobytej dotąd wiedzy, zadanie to jest możliwe do wykonania. Jeżeli uznasz, że samodzielne znalezienie odpowiedniej drogi jest zbyt trudne, obejrzyj najpierw pierwszy klip wideo, w którym przedstawiam nieco informacji na ten temat.

ĆWICZENIE 49

Serwer danych statystycznych

Kolejna faza projektu to implementacja pierwszej funkcjonalności serwera `statserve`. Program przygotowany w ćwiczeniu 48. powinien działać i nie ulegać awarii. Nie zapominaj o podejściu programowania defensywnego i zanim będziesz kontynuować pracę, podejmij wszelkie możliwe próby uszkodzenia programu oraz doprowadzenia go do awarii. Obejrzyj oba klipy wideo przygotowane dla ćwiczenia 48. i zobacz, jak ja sobie z tym poradziłem.

Celem serwera `statserve` jest umożliwienie klientom nawiązania z nim połączenia i wydawanie przez nich poleceń przeznaczonych do modyfikacji danych statystycznych. Jak pamiętasz, w jednym z wcześniejszych ćwiczeń dowiedziałeś się nieco o przyrostowym dodawaniu danych statystycznych. Ponadto powinieneś wiedzieć, jak używać struktur danych, takich jak `Hashmap`, tablice dynamiczne, drzewa binarne i drzewa trójkowe. Wymienione struktury będziemy wykorzystywać w serwerze `statserve` w celu implementacji jego funkcjonalności.

Specyfikacja

Twoim zadaniem jest implementacja protokołu, który będzie mógł być używany przez klienty do przechowywania danych statystycznych. Jak pamiętasz z ćwiczenia 43., mamy cztery proste operacje, jakie można przeprowadzić w API *stats.h*:

`create()`. Utworzenie nowych danych statystycznych.

`mean()`. Pobranie bieżącej średniej dla danych statystycznych.

`sample()`. Dodanie nowej próbki do danych statystycznych.

`dump()`. Pobranie wszystkich elementów z danych statystycznych (suma, suma kwadratów, n, minimum i maksimum).

To dopiero jest początek protokołu, trzeba będzie jeszcze zapewnić obsługę kilku dodatkowych zadań:

1. Należy zaoferować użytkownikom możliwość nadawania nazw tym danym statystycznym, co oznacza użycie jednej ze struktur danych mapowania do przeprowadzenia mapowania nazw na struktury `Stats`.
2. Dla każdej nazwy trzeba dodać obsługę standardowych operacji **CRUD** (ang. *create*, *read*, *update*, *delete* — tworzenie, odczyt, uaktualnianie i usuwanie). Aktualna lista przedstawionych poleceń obejmuje utworzenie danych, obliczenie średniej, pobranie danych w celu ich odczytu oraz dodanie próbki jako formy uaktualnienia danych. Konieczne jest więc zaimplementowanie operacji usunięcia danych.
3. Może istnieć potrzeba utworzenia polecenia `list` pozwalającego na wyświetlanie wszystkich danych statystycznych dostępnych w serwerze.

Mając na uwadze to, że serwer `statserve` powinien obsługiwać protokół pozwalający na przeprowadzenie wymienionych powyżej operacji, przystępujemy do utworzenia danych statystycznych, uaktualnienia ich za pomocą próbki, usunięcia, pobrania wszystkich danych, obliczenia średniej i na koniec wyświetlenia wszystkich danych statystycznych.

Postaraj się opracować prosty (naprawdę *prosty*) protokół, pozwalający na użycie danych w postaci zwykłego tekstu, i zobacz, jaki będzie efekt. Protokół najpierw opracuj na papierze, następnie obejrzyj klip wideo dla tego ćwiczenia, aby dowiedzieć się, jak zaprojektować protokół i otrzymać więcej informacji na temat bieżącej fazy projektu.

Zachęcam Cię również do utworzenia testów jednostkowych w celu sprawdzenia, czy protokół przetwarza dane niezależnie od serwera. Utwórz oddzielne pliki *.c* i *.h* przeznaczone wyłącznie do przetwarzania ciągów tekstowych za pomocą protokołu, a później testuj kod źródłowy, aż zacznie działać zgodnie z oczekiwaniami. Dzięki temu ułatwisz sobie pracę później, gdy dodasz tę funkcjonalność do budowanego serwera.

ĆWICZENIE 50

Routing danych statystycznych

Po rozwiązaniu problemu protokołu i umieszczeniu danych statystycznych w strukturze danych kolejnym krokiem jest przystąpienie do wzbogacenia serwera. W tym ćwiczeniu musisz przeprowadzić przeprojektowanie i refaktoryzację pewnych fragmentów kodu. To ma swój cel, ponieważ przedstawia operacje koniecznie wykonywane podczas tworzenia oprogramowania. Bardzo często będziesz zmuszony wyrzucać stary kod, aby zrobić miejsce dla nowego. Nigdy nie powinieneś się za bardzo przywiązywać do utworzonego przez siebie kodu.

W tym ćwiczeniu wykorzystasz zaprezentowany w ćwiczeniu 47. routing URL do wprowadzenia modyfikacji w protokole w taki sposób, aby umożliwić przechowywanie danych statystycznych pod dowolnym adresem URL.

I to jest cała pomoc, jaką możesz tutaj uzyskać. To proste zadanie i powinieneś spróbować wykonać je samodzielnie, modyfikując protokół, uaktualniając struktury danych i zmieniając kod tak, aby działał.

Obejrzyj klip wideo przeznaczony dla tego ćwiczenia, a zobaczysz, co chcemy osiągnąć. Następnie postaraj się samodzielnie wykonać zadanie przed obejrzeniem drugiego klipu pokazującego moją implementację.

ĆWICZENIE 51

Przechowywanie danych statystycznych

Kolejnym problemem do rozwiązania jest przechowywanie danych statystycznych. Zaletą umieszczenia ich w pamięci jest dużo większa wydajność niż w przypadku przechowywania ich w innym miejscu. Tak naprawdę istnieją systemy przeznaczone do przechowywania ogromnych danych i doskonale sobie z tym radzą. Jednak w przypadku naszego małego serwera pewne dane możemy spokojnie przechowywać na dysku twardym.

Specyfikacja

W tym ćwiczeniu dodamy dwa polecenia odpowiedzialne za umieszczanie danych statystycznych na dysku twardym oraz za ich wczytywanie:

`store`. Jeżeli zostanie podany adres URL, należy go umieścić na dysku twardym.

`load`. Jeżeli zostaną podane dwa adresy URL, należy wczytać z dysku twardego dane statystyczne, opierając się na pierwszym adresie URL, a następnie umieścić je w drugim adresie URL w pamięci.

Wprawdzie przedstawiony problem wydaje się prosty, ale podczas implementacji będziesz musiał stoczyć kilka bitew.

1. Jeżeli adres URL zawiera znaki /, to mamy konflikt z systemem plików używającym takich ukośników. W jaki sposób rozwiążesz ten problem?
2. Jeżeli adres URL zawiera znaki /, to ktoś może wykorzystać serwer do nadpisania plików znajdujących się na dysku twardym przez podanie prowadzących do nich ścieżek dostępu. W jaki sposób rozwiążesz ten problem?
3. Jeżeli zdecydujesz się na użycie głęboko zagnieżdżonych katalogów, to poruszanie się po katalogach w celu znalezienia plików będzie odbywało się niezwykle wolno. W jaki sposób rozwiążesz ten problem?
4. Jeżeli zdecydujesz się na użycie jednego katalogu i wartości hash dla adresu URL (ups, chyba niechcący dałem Ci wskazówkę), to katalogi ze zbyt dużą liczbą plików będą charakteryzowały się małą wydajnością. W jaki sposób rozwiążesz ten problem?
5. Co się stanie, gdy ktoś wczyta do istniejącego adresu URL dane statystyczne z dysku twardego?
6. W jaki sposób użytkownik korzystający z serwera `statserve` może poznać miejsce przechowywania danych?

Alternatywne podejście dla użycia systemów plików do przechowywania danych polega na zastosowaniu bazy danych takiej jak SQLite i ogólnie SQL. Kolejną możliwością jest wykorzystanie systemu takiego jak GNU dbm (GDBM) do przechowywania danych w prostszej bazie danych.

Przeanalizuj wszystkie opcje, obejrzyj klip wideo przeznaczony dla tego ćwiczenia, wybierz najprostszą opcję i spróbuj ją zaimplementować. Podczas fazy analizy rozwiązania nie poświęcaj zbyt wiele czasu na szukanie najlepszego, ponieważ w kolejnym ćwiczeniu będziesz próbował ustalić, jak zniszczyć przygotowany serwer.

ĆWICZENIE 52

Hacking i usprawnianie serwera

Dla ostatniego ćwiczenia przygotowałem trzy klipy wideo. Pierwszy pokazuje, jak zaatakować serwer i spróbować go zniszczyć. Poznasz tutaj wiele doskonałych narzędzi i sztuczek pomagających w łamaniu protokołów, a wady projektu pokażę na przykładzie *mojej* implementacji projektu. Ten klip wideo powinien dostarczyć Ci wiele radości. Jeśli wykonywałeś kolejne fazy projektu, możesz ze mną konkurować pod względem tego, kto zbudował bardziej niezawodnie działający serwer.

Drugi klip wideo pokazuje, jak wprowadzam usprawnienia w serwerze. Przed obejrzeniem tego wideo najpierw powinieneś samodzielnie spróbować usprawnić serwer, a następnie sprawdzić, czy wprowadzone przez Ciebie ulepszenia są podobne do moich.

Trzeci i ostatni klip wideo pokazuje, jak można wprowadzać kolejne usprawnienia programu i jego projektu. Poruszyłem tutaj wszystkie zagadnienia dotyczące ukończonego projektu i jego dalszego usprawniania. W celu zakończenia prac nad projektem zwykle podejmuję następujące kroki:

1. Projekt umieszczam w internecie i udostępniam użytkownikom.
2. Dokumentuję i poprawiam użyteczność, aby mieć pewność, że dokumentacja jest łatwa do czytania.
3. Staram się zapewnić maksymalne pokrycie testami.
4. Usprawniam przypadki skrajne i dodaję zabezpieczenia przed wszelkimi rodzajami ataków, jakie mogę znaleźć.

W drugim klipie wideo znajdziesz prezentację powyższych kroków oraz wyjaśnienie, jak możesz je przeprowadzić samodzielnie.

Zakończenie

Niniejsza książka wydaje się monumentalnym przedsięwzięciem nie tylko dla początkującego programisty, ale również dla innych, którzy nie mają zbyt dużego doświadczenia w wielu omówionych tutaj obszarach. Z powodzeniem poznałeś solidne podstawy języka C, testowania, tworzenia bezpiecznego kodu, opracowywania algorytmów, struktur danych, testów jednostkowych i ogólnie rozwiązywania problemów. Moje gratulacje! Na tym etapie powinieneś być już znacznie lepszym programistą.

Zachęcam Cię do zapoznania się z innymi książkami poświęconymi programowaniu w języku C. Na pewno nie będzie błędem sięgnięcie po pozycję *Język ANSI C. Programowanie. Wydanie II*, napisaną przez twórców języka C, czyli Briana W. Kernighana i Dennisa M. Ritchiego. Materiał przedstawiony w moich książkach ma nauczyć Cię podstaw i stosowania praktycznej wersji języka C, pozwalającej na wykonanie danej pracy. To jednak oznacza, że zajmujesz się głównie innymi zagadnieniami. Z kolei wymieniona książka pokazuje głębszy język C, w postaci zdefiniowanej przez jego twórców oraz sam standard C.

Jeżeli chcesz się nadal rozwijać jako programista, zachęcam Cię do poznania przynajmniej czterech języków programowania. Jeżeli znałeś już jeden, to teraz poznałeś C. Możesz więc spróbować poznać kolejne z wymienionych poniżej:

- Python za pośrednictwem mojej książki *Learn Python The Hard Way, Third Edition* (Addison-Wesley, 2013).
- Ruby za pośrednictwem mojej książki *Learn Ruby The Hard Way, Third Edition* (Addison-Wesley, 2015).
- Go za pośrednictwem opracowanej dla niego dokumentacji dostępnej na stronie *https://golang.org/doc/*. To jest inny język opracowany przez autorów C i według mnie znacznie lepszy.
- Lua to niezwykle zabawny język programowania oferujący doskonałe API dla C, którego znajomością możesz już się cieszyć.
- JavaScript, choć nie jestem pewien, która książka będzie najlepsza.

Istnieje jeszcze wiele innych języków programowania, więc wybierz ten, który najbardziej Cię zainteresuje, i poznaj go. Gorąco zachęcam Cię do tego, ponieważ najłatwiejszym sposobem na osiągnięcie mistrzostwa w programowaniu i nabyciu pewności siebie w tej dziedzinie jest umacnianie się w możliwości poznawania wielu języków programowania. Cztery języki programowania to granica, po której przekroczeniu początkujący staje się kompetentnym programistą. Poza tym nauka programowania dostarcza również wiele radości.

Skorowidz

G

H

I

K

L

M

N

O

P

R

S

Ś

T

U

W

Z