O'REILLY®

Język C dla małych urządzeń

Krótki kod o wielkich możliwościach

Helion

Marc Loy

Tytuł oryginału: Smaller C: Lean Code for Small Machines

Tłumaczenie: Jacek Janusz

ISBN: 978-83-283-8631-0

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

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

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

Printed in Poland.

Spis treści

ROZDZIAŁ 6.

Wskaźniki i referencje

Bezpośredni dostęp do pamięci jest jednym z największych udogodnień języka C oferowanych programistom, którzy zajmują się niskopoziomowymi zagadnieniami związanymi na przykład ze sterownikami urządzeń lub systemami wbudowanymi. Język C udostępnia potężne narzędzia umożliwiające zarządzanie poszczególnymi bajtami pamięci. Mogą się one bardzo przydać, gdy trzeba umiejętnie wykorzystać każdy kawałek wolnej przestrzeni. Z drugiej jednak strony z wielkimi możliwościami zawsze wiąże się olbrzymia odpowiedzialność. Jeśli zajdzie potrzeba, dobrze jednak mieć pod ręką takie narzędzia.

W tym rozdziale zostanie przeanalizowany sposób, w jaki można ustalić położenie elementów w pamięci (ich *adresy*), a także korzystanie z tych lokalizacji za pomocą *wskaźników*, czyli zmiennych, które przechowują adresy innych zmiennych.

Adresowanie w języku C

O wskaźnikach była już mowa w rozdziale, w którym omawialiśmy zastosowanie funkcji `scanf()` do odczytywania zmiennych o typach podstawowych takich jak `int` i `float`, a także łańcuchów będących tablicami znakowymi. W przypadku liczb wymagane było użycie przedrostka `&`. Można go traktować jako prośbę o podanie „adresu" zmiennej. Zwraca on wartość liczbową, która informuje, w jakim miejscu pamięci została zapisana dana zmienna. Informację o tym miejscu można wyświetlić. Przeanalizuj program *r06/address.c*:

```
#include <stdio.h>

int main() {
  int answer = 42;
  double pi = 3.1415926;
  printf("Wartość zmiennej answer: %d\n", answer);
  printf("Adres zmiennej answer: %p\n", &answer);
  printf("Wartość zmiennej pi: %0.4f\n", pi);
  printf("Adres zmiennej pi: %p\n", &pi);
}
```

Za pomocą tego prostego programu tworzymy i inicjalizujemy dwie zmienne. Następnie używamy kilku instrukcji `printf()`, aby wyświetlić zarówno ich wartości, jak i lokalizacje w pamięci. Jeśli skompilujemy i uruchomimy ten przykład, otrzymamy takie wyniki:

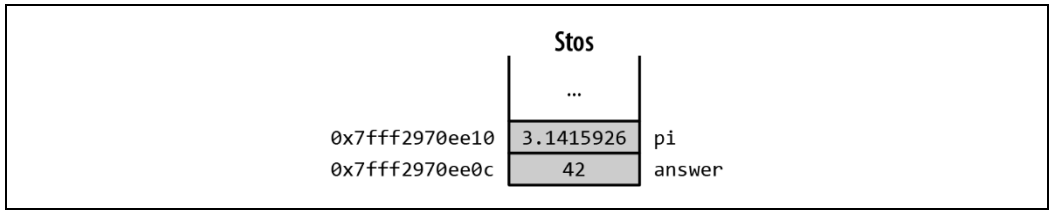
```
r06$ gcc address.c
r06$ ./a.out
Wartość zmiennej answer: 42
Adres zmiennej answer: 0x7fff2970ee0c
Wartość zmiennej pi: 3.1416
Adres zmiennej pi: 0x7fff2970ee10
```

Należałoby raczej napisać, że „otrzymamy *mniej więcej* takie wyniki". Konfiguracja Twojego sprzętu prawdopodobnie będzie się różnić od mojej, więc również adresy będą inne. Nawet kolejne wywołania tego programu mogą również zwracać odmienne wyniki, jeśli chodzi o wartości adresów. Lokalizacja, w której program zostanie umieszczony przed uruchomieniem, zależy od niezliczonych czynników. Jeśli którykolwiek z nich będzie inny, adresy prawdopodobnie również się zmienią.

W przypadku kolejnych przykładów bardziej przydatne będzie zwrócenie uwagi na to, które adresy są zbliżone do innych. Dokładne wartości nie są ważne.

Uzyskanie wartości zapisanej w zmiennej `answer` lub `pi` jest proste i robimy to już od rozdziału 2. Ale eksperymentowanie z adresami zmiennych jest czymś zupełnie nowym. Jak widać, w instrukcji `printf()` musieliśmy nawet użyć nowego łańcucha formatującego `%p`, aby je wyświetlić! Litera `p` tego formatu pochodzi od słowa *pointer* (wskaźnik), które jest ściśle związane ze słowem „adres". Zazwyczaj *wskaźnik* odnosi się do zmiennej, która przechowuje adres, nawet jeśli programiści potocznie mówią o wskaźniku jak o określonej wartości. Można się również natknąć na termin *referencja*, który w języku C jest synonimem wskaźnika. Tego słowa używa się jednak częściej podczas omawiania parametrów funkcji. Na przykład w samouczkach dostępnych w sieci możesz znaleźć takie sformułowania: „gdy przekazujesz referencję do tej funkcji...". Oznacza to, że do funkcji przekazujesz adres jakiejś zmiennej, a nie jej wartość.

Ale wróćmy do przykładu. Te wyświetlone wartości wskaźników są rzeczywiście dużymi liczbami! Nie zawsze tak będzie, ale nie jest to rzadkością w systemach zawierających gigabajty lub nawet terabajty pamięci RAM i używających adresacji logicznej pozwalającej na zarządzanie wieloma procesami. Co oznaczają te wartości? Są to miejsca w pamięci dostępnej dla procesu, w których są przechowywane wartości zmiennych. Na rysunku 6.1 zaprezentowano podstawową konfigurację pamięci odpowiadającą omawianemu przykładowi.



Rysunek 6.1. Wartości i adresy zmiennych

Nawet bez ustalenia dokładnej wartości dziesiętnej widzimy, że wartości adresów są do siebie zbliżone. Dokładniej mówiąc, adres zmiennej `pi` jest o cztery bajty większy niż adres `answer`. Typ `int` w mojej maszynie zajmuje cztery bajty, więc mam nadzieję, że dostrzegasz pewien związek. Typ `float` zajmuje osiem bajtów. Czy zgadniesz, jaki adres mogłaby otrzymać trzecia zmienna dodana do programu?

Sprawdźmy to wspólnie. Program *r06/address2.c* definiuje trzecią zmienną `int`, a następnie dla wszystkich wyświetla wartości i adresy:

```
#include <stdio.h>

int main() {
  int answer = 42;
  double pi = 3.1415926;
  int extra = 1234;
  printf("Wartość zmiennej answer: %d\n", answer);
  printf("Adres zmiennej answer: %p\n", &answer);
  printf("Wartość zmiennej pi: %0.4f\n", pi);
  printf("Adres zmiennej pi: %p\n", &pi);
  printf("Wartość zmiennej extra: %d\n", extra);
  printf("Adres zmiennej extra: %p\n", &extra);
}
```

A oto wynik działania tego programu:

```
r06$ gcc address2.c
r06$ ./a.out
Wartość zmiennej answer: 42
Adres zmiennej answer: 0x7fff9c827498
Wartość zmiennej pi: 3.1416
Adres zmiennej pi: 0x7fff9c8274a0
Wartość zmiennej extra: 1234
Adres zmiennej extra: 0x7fff9c82749c
```

Hmm, okazuje się, że zmienne nie są przechowywane w kolejności, w jakiej zostały zdefiniowane. Jakie to dziwne! Jeśli przyjrzysz się uważnie, zobaczysz, że zmienna `answer` jest nadal przechowywana jako pierwsza (adres `0x...498`), zmienna `extra` cztery bajty dalej (`0x...49c`), a na końcu jest `pi` — również po czterech bajtach (`0x...4A0`). Kompilator często umieszcza zmienne w pamięci w sposób, który uzna za najbardziej skuteczny, a właściwa kolejność nie zawsze jest zgodna z tym, co widać w kodzie źródłowym. Więc nawet jeśli kolejność jest trochę zaskakująca, nadal widać, że wszystkie zmienne są umieszczone jedna po drugiej i zajmują dokładnie tyle miejsca, ile wynika z ich typów.

Wartość NULL i błędy związane ze wskaźnikami

Nagłówek *stdio.h* zawiera bardzo przydatną wartość `NULL`, której możemy użyć, gdy mamy do czynienia ze wskaźnikiem „pustym” lub niezainicjalizowanym. Wartość `NULL` można przypisać do zmiennej wskaźnikowej lub użyć jej w porównaniu, aby sprawdzić, czy dany wskaźnik jest prawidłowy. Jeśli deklarowane zmienne chcesz zawsze inicjalizować, w przypadku wskaźników możesz wykorzystać wartość `NULL`. Na przykład moglibyśmy zadeklarować dwie zmienne: `double` i wskaźnik do `double`. Zainicjalizujemy je wartościami „zerowymi”, które później zmienimy:

```
double pi = 0.0;
double *pi_ptr = NULL;
// ...
pi = 3.14156;
pi_ptr = &pi;
```

Jeśli nie wiesz dokładnie, jaką wartość ma wskaźnik, powinieneś zawsze sprawdzać, czy czasem nie jest równy `NULL`. Taką operację należy wykonać na przykład w funkcji, której został przekazany wskaźnik:

```
double messyAreaCalculator(double radius, double *pi_ptr) {
  if (pi_ptr == NULL) {
    printf("Nie można obliczyć powierzchni za pomocą błędnego wskaźnika!\n");
    return 0.0;
  }
  return radius * radius * (*pi_ptr);
}
```

Oczywiście nie jest to najprostszy sposób na obliczenie powierzchni koła, ale powszechnie stosuje się instrukcję `if`. Taka konstrukcja gwarantuje poprawność danych. Jeśli zapomnisz sprawdzić wskaźnik i mimo tego spróbujesz się do niego odwołać, Twój program (zazwyczaj) przestanie działać i prawdopodobnie wyświetli błąd podobny do poniższego:

```
Segmentation fault (core dumped)
```

Zapewne i tak nie będziesz mógł nic zrobić z pustym wskaźnikiem, ale jeśli przed użyciem go sprawdzisz, wyświetlisz użytkownikowi ładniejszy komunikat o błędzie i unikniesz awarii.

Tablice

A co z tablicami i łańcuchami? Czy są one również umieszczane na stosie jak typy proste? Czy otrzymają adresy z tej samej ogólnej części pamięci? Stwórzmy kilka zmiennych tablicowych i zobaczmy, gdzie zostaną umieszczone i ile miejsca zajmą. Oto odpowiedni program — *r06/address3.c*. Wyświetla on także rozmiar zmiennych, dzięki czemu można łatwo zweryfikować, ile miejsca zostało na nie przydzielone:

```
#include <stdio.h>

int main() {
  char title[30] = "Adresy przykładowych tablic";
  int page_counts[5] = { 14, 78, 49, 18, 50 };
  printf("Wartość zmiennej title: %s\n", title);
  printf("Adres zmiennej title: %p\n", &title);
  printf("Rozmiar zmiennej title: %lu\n", sizeof(title));
  printf("Wartości tablicy page_counts: {");
  for (int p = 0; p < 5; p++) {
    printf(" %d", page_counts[p]);
  }
  printf(" }\n");
  printf("Adres zmiennej page_counts: %p\n", &page_counts);
  printf("Rozmiar zmiennej page_counts: %lu\n", sizeof(page_counts));
}
```

A oto wyniki:

```
Wartość zmiennej title: Adresy przykładowych tablic
Adres zmiennej title: 0x7ffe971a5dc0
Rozmiar zmiennej title: 30
Wartości tablicy page_counts: { 14 78 49 18 50 }
Adres zmiennej page_counts: 0x7ffe971a5da0
Rozmiar zmiennej page_counts: 20
```

Co prawda kompilator ponownie przestawił zmienne w pamięci, ale widzimy, że tablica `page_counts` zajmuje 20 bajtów (5·4 bajty na pojedynczą wartość `int`), a `title` ma adres o 32 bajty większy od adresu `page_counts` (możesz zignorować takie same fragmenty adresów i wykonać pewne obliczenia matematyczne: 0xc0 – 0xa0 == 0x20 == 32). Co się więc mieści w dodatkowych 12 bajtach? Z przydzieleniem

miejsca na tablicę jest związany pewien narzut, a kompilator łaskawie o tym pamiętał. Na szczęście jako programiści lub użytkownicy nie musimy się martwić o ten narzut. Również jako programiści potwierdzamy, że kompilator rezerwuje wystarczająco dużo miejsca na samą tablicę.

Zmienne lokalne i stos

Gdzie dokładnie znajduje się więc to „miejsce", o którym cały czas wspominamy? Ogólnie rzecz ujmując, jest przydzielane w pamięci RAM komputera. W przypadku zmiennych zdefiniowanych w funkcji (jak pamiętasz z podrozdziału „Funkcja main ()" zawartego w rozdziale 5., `main()` również jest funkcją), miejsce jest przydzielane na *stosie*. To nazwa obszaru w pamięci, w którym są tworzone i przechowywane wszystkie zmienne lokalne podczas wykonywania różnych wywołań funkcji. Organizowanie i utrzymywanie tych przestrzeni pamięci jest jednym z podstawowych zadań systemu operacyjnego.

Przeanalizuj kolejny, niewielki program *r06/do_stuff.c*. Jak zwykle mamy funkcję `main()`, a także inną o nazwie `do_stuff()`, która, cóż, wykonuje po prostu różne rzeczy. Niezbyt skomplikowane, ponieważ definiuje zmienną typu `int`, a następnie wyświetla informacje o niej. Jednak nawet nudne funkcje używają stosu, dzięki czemu można się dowiedzieć, w jaki sposób wywołania funkcji wykorzystują pamięć komputera!

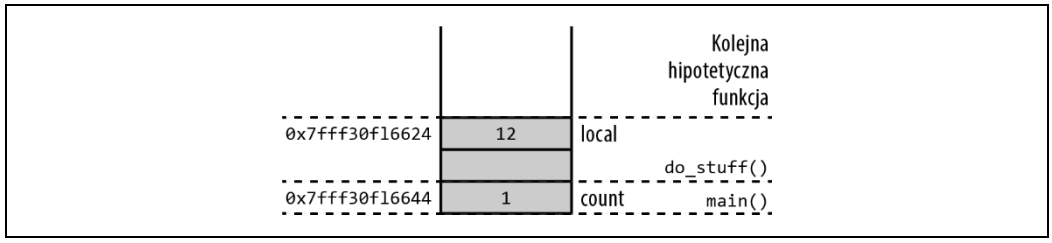
```
#include <stdio.h>

void do_stuff() {
  int local = 12;
  printf("Zmienna local ma wartość %d\n", local);
  printf("Adres zmiennej local: %p\n", &local);
}

int main() {
  int count = 1;
  printf("Wartość początkowa zmiennej count wynosi %d\n", count);
  printf("Adres zmiennej count: %p\n", &count);
  do_stuff();
}
```

A oto uzyskane wyniki:

```
r06$ gcc do_stuff.c
r06$ ./a.out
Wartość początkowa zmiennej count wynosi 1
Adres zmiennej count: 0x7fff30f1b644
Zmienna local ma wartość 12
Adres zmiennej local: 0x7fff30f1b624
```

Widzimy, że adresy zmiennej `count` z `main()` oraz `local` z `do_stuff()` są do siebie podobne. Obie zmienne zostały umieszczone na stosie. Został on zaprezentowany na rysunku 6.2.

Stąd właśnie pochodzi nazwa „stos" — wywołania funkcji i ich zmienne lokalne tworzą coś w rodzaju stosu. Jeśli funkcja `do_stuff()` wywoła jakąś inną, jej zmienne zostaną umieszczone powyżej obszaru pamięci zajmowanego przez zmienną `local`. Gdy działanie funkcji się zakończy, jej zmienne są usuwane ze stosu. Takie układanie danych może trwać dość długo, ale nie w nieskończoność. Jeśli na przykład w funkcji rekurencyjnej (którą przeanalizowaliśmy w rozdziale 5.) nie podasz odpowiedniego przypadku podstawowego, wzrastająca zajętość stosu spowoduje ostatecznie awarię programu.



Rysunek 6.2. Zmienne lokalne na stosie

Być może zauważyłeś, że adresy na rysunku 6.2 maleją. Stos może się zaczynać na początku pamięci przydzielonej programowi (wtedy adresy będą wzrastać) albo na jej końcu (wówczas adresy będą się zmniejszać). Wybór wersji zależy od architektury i systemu operacyjnego. Sama idea stosu i jego wzrastania pozostaje jednak taka sama.

Stos zawiera również wszelkie parametry, które są przekazywane do funkcji, jak również zmienne pętli lub inne, które są deklarowane wewnątrz funkcji. Rozważ poniższy kod:

```
float average(float a, float b) {
  float sum = a + b;
  if (sum < 0) {
    for (int i = 0; i < 5; i++) {
      printf("Ostrzeżenie!\n");
    }
    printf("Wartość średnia jest ujemna. Uważaj!\n");
  }
  return sum / 2;
}
```

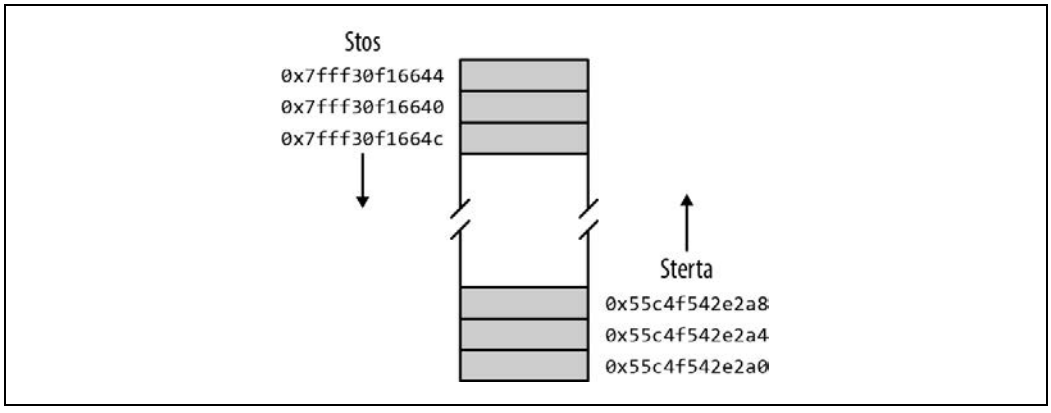
Ten fragment kodu spowoduje zarezerwowanie na stosie miejsca na następujące elementy:

- wartość typu `float` zwracaną przez funkcję `average()`,
- parametr `a` typu `float`,
- parametr `b` typu `float`,
- lokalną zmienną `sum` typu `float`,
- zmienną `i` typu `int` wykorzystywaną w pętli (jedynie w przypadku, gdy `sum<0`).

Jak widać, stos jest naprawdę uniwersalny! Prawie każdy element, który ma związek z daną funkcją, będzie przechowywany na stosie.

Zmienne globalne i sterta

Ale w takim razie co ze zmiennymi globalnymi, które nie są „połączone” z żadną określoną funkcją? Są one umieszczane w oddzielnej części pamięci zwanej *stertą*. Jeśli słowo „sterta” kojarzy Ci się z czymś niechlujnym, masz rację. Każdy kawałek pamięci, którego potrzebuje Twój program, a który nie może być częścią stosu, zostanie przydzielony na stercie. Rysunek 6.3 przedstawia porównanie stosu i sterty.



Rysunek 6.3. Pamięć stosu i sterty

Stos i sterta współdzielą jeden logiczny obszar pamięci przekazany programowi podczas jego uruchamiania. W miarę wywoływania kolejnych funkcji stos będzie rosnąć (w tym przypadku od „góry"). Gdy funkcje zakończą działanie, stos się zmniejszy. Zmienne globalne sprawiają, że sterta rośnie (w górę od „dołu"). Duże tablice lub inne struktury mogą być również przydzielane na stercie (w punkcie „Zarządzanie pamięcią tablic" w tym rozdziale dowiesz się, jak można samodzielnie zarządzać pamięcią na stercie). Pamięć w niektórych obszarach sterty można zwolnić, jednak zmienne globalne pozostaną w niej do momentu, kiedy zakończy się wykonywanie programu.

Współpraca tych dwóch fragmentów pamięci zostanie dokładniej przeanalizowana w rozdziale 9., w podpunkcie „Stos i sterta". Gdy stos i sterta będą się zwiększać, wolna przestrzeń pośrodku będzie się stawała coraz mniejsza. Jeśli te dwie struktury w końcu się ze sobą spotkają, pojawi się problem. Ponieważ stos nie będzie mógł dalej rosnąć, nie będzie można wywołać więcej funkcji. Jeśli mimo tego spróbujesz wtedy wykonać jakąś funkcję, prawdopodobnie program się zawiesi. Sytuacja wygląda podobnie w przypadku sterty — jeśli nie ma na niej wolnego miejsca, ale mimo tego chcesz koniecznie przydzielić trochę pamięci programowi, jedyna odpowiedź komputera polega na zatrzymaniu aplikacji.

Zadaniem programisty jest zadbanie o to, by takie problemy się nie pojawiały. Język C nie powstrzyma Cię przed popełnieniem błędu, ale z drugiej strony udostępni Ci różne możliwości pozwalające na znalezienie właściwego rozwiązania. W rozdziale 10. omówimy kilka problematycznych sytuacji związanych z mikrokontrolerami i sztuczki pozwalające na uniknięcie problemów.

Arytmetyka wskaźników

Niezależnie od miejsca przechowywania zmiennych język C udostępnia potężną (ale także potencjalnie niebezpieczną) opcję pozwalającą na bezpośrednią obsługę adresów. Nie ograniczamy się do wyświetlenia adresu zmiennej w celu jego sprawdzenia. Możemy zapisać go w innym wskaźniku. Następnie możemy go użyć, aby uzyskać dostęp do tego samego obszaru pamięci, który został przydzielony dla pierwotnej zmiennej.

Przeanalizuj program *r06/pointer.c*, aby poznać przykład użycia zmiennej, która wskazuje inną zmienną. W kodzie zostało wyróżnionych kilka ważnych fragmentów:

```
#include <stdio.h>

int main() {
  double total = 500.0;                          ❶
  int count = 34;
  double average = total / count;
  printf("Średnia z %d elementów, których suma wynosi %.1f, jest równa %.2f\n",
     count, total, average);

  // A teraz zróbmy to samo za pomocą wskaźników
  double *total_ptr = &total;                    ❷
  int *count_ptr = &count;
  printf("Zmienna wskaźnikowa total_ptr wskazuje na adres zmiennej total:\n");
  printf("  total_ptr %p == %p &total\n", total_ptr, &total);

  // Do wartości wskazywanej przez wskaźnik można się odwoływać
  // za pomocą przedrostka '*' (operacja wyłuskania)
  printf("Suma wynosi %.1f\n", *total_ptr);
  // Załóżmy, że wcześniej zapomnieliśmy o dwóch elementach, więc skorygujmy wartość zmiennej count:
  *count_ptr += 2;                               ❸
  average = *total_ptr / *count_ptr;
  printf("Poprawna wartość średniej z %d elementów, których suma wynosi %.1f, jest równa %.2f\n",
     count, total, average);                     ❹
}
```

❶ Rozpoczynamy ze zwykłym zestawem zmiennych i wykonujemy proste obliczenie.

❷ Następnie tworzymy dwie nowe zmienne o odpowiednich typach wskaźnikowych — `total_ptr` typu `double*` wskazującą zmienną `total`, a także `count_ptr` typu `int*` wskazującą zmienną `count`.

❸ Wskaźniki wyłuskujemy w celu modyfikacji zmiennych, które są przez nie wskazywane.

❹ Wreszcie udowadniamy, że pierwotne zmienne niebędące wskaźnikami zostały zmodyfikowane poprzez odpowiednie operacje wykonywane na wskaźnikach.

Oto uzyskane wyniki:

```
r06$ gcc pointer.c
r06$ ./a.out
Średnia z 34 elementów, których suma wynosi 500.0, jest równa 14.71
Zmienna wskaźnikowa total_ptr wskazuje na adres zmiennej total:
  total_ptr 0x7ffdfdc079c8 == 0x7ffdfdc079c8 &total
Suma wynosi 500.0
Poprawna wartość średniej z 36 elementów, których suma wynosi 500.0, jest równa 13.89
```

Dane wyjściowe nie są zbyt ekscytujące, ale dowodzą, że byliśmy w stanie zmodyfikować wartość zmiennej `count` za pomocą wskaźnika `count_ptr`. Dostęp do danych za pomocą wskaźników to dość zaawansowana umiejętność. Nie martw się, jeśli nie do końca wszystko zrozumiałeś. Testuj i analizuj kolejne przykłady, a wkrótce przywykniesz do składni, co z kolei pozwoli Ci używać wskaźników w przyszłych projektach.

Wskaźniki do tablic

Używaliśmy już takich wskaźników, chociaż były one bardzo sprytnie ukryte pod postacią tablic. Przypomnij sobie zaawansowane wykorzystanie funkcji `scanf()` w punkcie „Instrukcja scanf() i parsowanie danych wejściowych” z rozdziału 2. Gdy chcieliśmy wczytać liczbę, musieliśmy używać

przedrostka & z nazwą zmiennej numerycznej. Ale odczytywanie łańcuchów nie wymagało takiej składni — po prostu podawaliśmy nazwę tablicy. Wynikało to stąd, że tablice w języku C są już wskaźnikami o oczekiwanej strukturze ułatwiającej odczyt i zapis poszczególnych elementów.

Okazuje się, że można mieć dostęp do zawartości tablicy bez użycia nawiasów kwadratowych. Możesz użyć dokładnie tego samego operatora wyłuskania, który wykorzystywaliśmy w poprzednim przykładzie. Dzięki temu istnieje możliwość dodawania liczb całkowitych do zmiennej tablicowej, a także odejmowania od niej, aby uzyskać dostęp do poszczególnych elementów. Najlepiej będzie, jak przyjrzymy się odpowiedniemu przykładowi kodu. Przeanalizuj program *r06/direct_edit.c*:

```
#include <stdio.h>

int main() {
  char name[] = "j.r. Karpiński";              ❶
  printf("Przed modyfikacją: %s\n", name);
  // Użyjmy arytmetyki wskaźników, aby zmienić wielkość liter w inicjałach
  *name = 'J';                                 ❷
  *(name + 2) = 'R';                           ❸
  printf("Po modyfikacji: %s\n", name);        ❹
}
```

❶ Deklarujemy i standardowo inicjalizujemy łańcuch (tablicę typu `char`).

❷ Zmienną tablicową możemy wyłuskać, aby odczytać lub zmodyfikować pierwszy znak. Równoważną instrukcją jest `name[0] = 'J'`.

❸ Możemy również wyłuskiwać wyrażenie zawierające zmienną tablicową. Można dodawać lub odejmować wartości całkowite, co odpowiada przemieszczeniu się w tablicy do przodu lub tyłu o odpowiednią liczbę elementów. Równoważną instrukcją jest `name[2] = 'R'`.

❹ Widzimy, że pomimo tego, iż zmienna tablicowa nie została „naruszona”, mogliśmy wykonać edycję łańcucha.

Skompiluj program i uruchom go. Oto uzyskane wyniki:

```
r06$ gcc direct_edit.c
r06$ ./a.out
Przed modyfikacją: j.r. Karpiński
Po modyfikacji: J.R. Karpiński
```

Arytmetyka i wyłuskiwanie działa również w przypadku tablic innych typów. Arytmetyka wskaźników może być wykorzystywana w pętlach przetwarzających tablice, na przykład wtedy, gdy zwiększenie wskaźnika oznacza przejście do następnego elementu. Takie używanie wskaźników może być niezwykle efektywne. Pamiętaj jednak, że proste operacje, które widzieliśmy w programie *direct_edit.c*, były kiedyś rzeczywiście szybsze, ale współczesne kompilatory C potrafią bardzo (i to bardzo!) dobrze optymalizować kod.

Zalecam najpierw uzyskać prawidłowo działający program, a dopiero później martwić się o jego wydajność. Rozdział 10. dotyczy pamięci i innych zasobów wykorzystywanych na platformie Arduino, w przypadku której rozważania dotyczące wydajności są nieco bardziej uzasadnione. Jednak nawet w takim środowisku optymalizacja nie powinna być Twoim pierwszym zmartwieniem.

Funkcje i wskaźniki

Wskaźniki naprawdę stają się przyjacielem programisty, gdy zaczyna ich używać jako parametrów lub zwracać je z funkcji. Dzięki temu można zarządzać częścią współdzielonej pamięci bez czynienia jej globalną. Przeanalizuj funkcje z programu *r06/increment.c*:

```
void increment_me(int me, int amount) {
  // zwiększ zmienną "me" o wartość "amount"
  me += amount;
  printf("  Wewnątrz funkcji increment_me: %d\n", me);
}

void increment_me_too(int *me, int amount) {
  // zwiększ zmienną wskazywaną przez "me" o wartość "amount"
  *me += amount;
  printf("  Wewnątrz funkcji increment_me_too: %d\n", *me);
}
```

Pierwsza funkcja, czyli `increment_me()`, powinna wydawać się znajoma. Do tej pory do funkcji przekazywaliśmy wartości. Wewnątrz `increment_me()` możemy dodać zmienną `amount` do `me` i uzyskać poprawną odpowiedź. Jednakże w funkcji `main()` przekazaliśmy tylko *wartość* zmiennej `count`. Powinno to oznaczać, że pozostanie ona nietknięta.

Ale druga funkcja `increment_me_too()` używa wskaźnika. Zamiast prostej wartości możemy teraz przekazać *referencję* do zmiennej `count`. Powinniśmy więc stwierdzić, że po zakończeniu działania funkcji `increment_me_too()` zmienna `count` będzie miała zmodyfikowaną wartość. Sprawdźmy to. Oto najprostsza funkcja `main()`, w której można przetestować obie funkcje:

```
int main() {
  int count = 1;
  printf("Wartość początkowa zmiennej count: %d\n", count);
  increment_me(count, 5);
  printf("Wartość zmiennej count po wywołaniu increment_me: %d\n", count);
  increment_me_too(&count, 5);
  printf("Wartość zmiennej count po wywołaniu increment_me_too: %d\n", count);
}
```

Oto uzyskane wyniki:

```
r06$ gcc increment.c
r06$ ./a.out
Wartość początkowa zmiennej count: 1
  Wewnątrz funkcji increment_me: 6
Wartość zmiennej count po wywołaniu increment_me: 1
  Wewnątrz funkcji increment_me_too: 6
Wartość zmiennej count po wywołaniu increment_me_too: 6
```

Świetnie. Program zadziałał zgodnie z oczekiwaniami. Funkcja `increment_me()` nie wpływa na wartość zmiennej `count` przekazaną z poziomu `main()`, ale już `increment_me_too()` ją modyfikuje. Często będziesz napotykać sformułowania „przekazana przez zmienną” i „przekazana przez referencję”, które oznaczają sposoby, w jakie funkcja obsługuje przekazywane jej argumenty. Zauważ, że w przypadku `increment_me_too()`jeden parametr jest referencją, a drugi wartością. Typy tych parametrów mogą być dowolnie wybierane. Programista musi się tylko upewnić, że poprawnie używa ich w funkcji.

Funkcje mogą również zwracać wskaźnik do czegoś, co utworzyły na stercie. Jak zobaczymy w rozdziałach 9. i 11., jest to popularna sztuczka w przypadku bibliotek zewnętrznych.

Zarządzanie pamięcią tablic

Jeśli wiesz, że będziesz potrzebować dużej ilości pamięci, na przykład do przechowywania obrazu lub dźwięku, możesz ją przydzielić dla tablicy (a także struktury — patrz punkt „Definiowanie struktur" w dalszej części tego rozdziału). W wyniku uzyskasz wskaźnik, który możesz następnie przekazać do określonych funkcji. W ten sposób nie wykonujesz żadnej kopii danych, a oprócz tego możesz się *wcześniej* upewnić, że masz dostęp do zarezerwowanej pamięci. Jest to naprawdę duża zaleta podczas pracy z treściami pochodzącymi z nieznanych źródeł. Jeśli żądana ilość pamięci nie jest dostępna, możesz wyświetlić odpowiedni komunikat o błędzie i poprosić użytkownika, aby spróbował ponownie. Dzięki temu upewnisz się, że program po prostu nie zawiesi się bez żadnego wyjaśnienia.

Przydzielanie pamięci za pomocą malloc()

Do obsługi większych tablic zazwyczaj używamy sterty. Można w niej jednak przydzielić pamięć dla dowolnych struktur. Aby to zrobić, należy wywołać funkcję `malloc()`, przekazawszy jej potrzebną ilość pamięci w bajtach. Ponieważ została ona zdefiniowana w innym pliku nagłówkowym (*stdlib.h*), musimy go dołączyć w taki sposób, jak robimy w przypadku *stdio.h*. W dalszej części książki poznamy więcej funkcji z nagłówka *stdlib.h*. Na razie wystarczy dodać odpowiedni wiersz poniżej standardowej dyrektywy `include`:

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

// ...
```

Teraz możemy utworzyć prosty program, który zilustruje sposób przydzielania pamięci dla zmiennych globalnych i lokalnych, a także niestandardowej struktury na stercie. Przeanalizuj plik *r06/memory.c*:

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

int result_code = 404;
char result_msg[25] = "Nie znaleziono pliku";

int main() {
  char temp[20] = "Wczytywanie ...";
  int success = 200;

  char *buffer = (char *)malloc(20 * sizeof(char));

  // Nie będziemy wykorzystywać tych zmiennych,
  // ale po prostu wyświetlimy ich adresy
  printf("Adres zmiennej result_code:        %p\n", &result_code);
  printf("Adres zmiennej result_msg:         %p\n", &result_msg);
  printf("Adres zmiennej temp:               %p\n", &temp);
  printf("Adres zmiennej success:            %p\n", &success);
  printf("Adres zmiennej buffer (na stosie): %p\n", buffer);
}
```

Globalne deklaracje zmiennych `result_code` i `result_msg` oraz zmiennych lokalnych `temp` i `success` powinny być Ci już znane. Przyjrzyj się jednak, jak zadeklarowaliśmy zmienną `buffer`. Oto użycie funkcji `malloc()` w realnym programie. Poprosiliśmy o przydzielenie miejsca na 20 znaków. Można podać literał liczbowy, ale zwykle bezpieczniejsze (a tak naprawdę często konieczne) jest użycie operatora `sizeof`, jak pokazano w tym przykładzie. Różne systemy wykorzystują odmienne reguły dotyczące wielkości typów i alokacji pamięci, więc zastosowanie operatora `sizeof` chroni przed pojawieniem się niezamierzonych błędów.

Przyjrzyjmy się wyświetlonym adresom zmiennych:

```
r06$ gcc memory.c
r06$ ./a.out
Adres zmiennej result_code:        0x55c4f49c8010
Adres zmiennej result_msg:         0x55c4f49c8020
Adres zmiennej temp:               0x7fffc84f1840
Adres zmiennej success:            0x7fffc84f1834
Adres zmiennej buffer (na stosie): 0x55c4f542e2a0
```

Jak już wspomniano, nie należy się sugerować dokładnymi wartościami adresów. Należy raczej szukać podobnych lokalizacji. Chyba widzisz, że zmienne globalne i wskaźnik `buffer`, dla którego miejsce zarezerwowaliśmy za pomocą funkcji `malloc()`, znajdują się mniej więcej w tym samym obszarze. Podobnie są pogrupowane dwie zmienne lokalne z funkcji `main()`, ale zajmują inne miejsce w pamięci.

Jak widać, funkcja `malloc()` przydziela miejsce dla danych na stercie. Tej przydzielonej przestrzeni użyjemy w punkcie „Wskaźniki do struktur”, ale najpierw musimy się przyjrzeć funkcji `free()`, która jest ściśle związana z działaniem funkcji `malloc()`. Gdy pamięć zostaje przydzielona, koniecznie należy ją zwolnić po zakończeniu wykorzystywania.

Zwalnianie pamięci za pomocą free()

Podczas analizy rysunku 6.3 stwierdziliśmy, że jeśli zużyje się zbyt dużo pamięci na stosie lub stercie (lub na obu z nich), komputerowi zabraknie pamięci i program ulegnie awarii. Jedną z zalet wykorzystywania sterty jest to, że można decydować, kiedy i w jaki sposób pamięć jest przydzielana i zwracana. Oczywiście ma to również wadę — trzeba samodzielnie wykonać operację „zwracania”. W wielu nowszych językach istnieją mechanizmy uwalniające od tego zadania programistę, który zbyt łatwo może zapomnieć o sprzątaniu po sobie. Być może znasz półoficjalne określenie tego problemu — chodzi o wyciek pamięci.

Aby w języku C zwrócić pamięć i uniknąć wycieków pamięci, należy użyć funkcji `free()` (również pochodzącej z nagłówka *stdlib.h*). Działa ona dość prosto — po prostu przekazujesz jej wskaźnik uzyskany po odpowiednim wywołaniu `malloc()`. Oto kod:

```
free(buffer);
```

Proste, nieprawdaż? Problem polega na tym, że należy pamiętać o używaniu funkcji `free()`. Na początku może się to nie wydawać kłopotliwe, ale będzie się stawało coraz trudniejsze, gdy zaczniesz używać funkcji do tworzenia i usuwania bloków danych. Ile razy zostały wywołane funkcje tworzące? Czy wykonałeś odpowiednie funkcje usuwające? Co by się stało, gdybyś spróbował usunąć coś, co nigdy nie zostało przydzielone? Wszystkie te pytania sprawiają, że zarządzanie wykorzystaniem pamięci jest równie kłopotliwe, co niezbędne.

Struktury w języku C

W miarę zajmowania się ciekawszymi problemami Twoje potrzeby przechowywania danych będą się stawać coraz bardziej złożone. Jeśli na przykład chcesz się zajmować wyświetlaczami LCD, musisz obsługiwać piksele, a więc określać ich kolor oraz lokalizację. Lokalizacja zostanie zdefiniowana za pomocą współrzędnych *x* i *y*. Mógłbyś utworzyć trzy oddzielne tablice (jedną dla kolorów, drugą dla współrzędnych *x*, a wreszcie trzecią dla współrzędnych *y*). Taki zbiór nie jest jednak łatwo przekazać do funkcji i otrzymać z niej, a poza tym zwiększa on prawdopodobieństwo pojawienia się błędów takich jak dodanie koloru, ale zapomnienie o jednej ze współrzędnych. Na szczęście język C zawiera słowo kluczowe `struct` umożliwiające tworzenie specjalnych kontenerów w celu zdefiniowania określonych struktur danych.

Zacytujmy Kernighana i Ritchiego: „*Struktura* jest zbiorem zawierającym jedną lub więcej zmiennych, być może różnych typów, zgrupowanych razem pod jedną nazwą w celu uproszczenia ich obsługi”[1]. Autorzy zauważają, że w innych językach takie rozwiązanie nosi nazwę *rekordu*. W internecie można się też natknąć na termin „typ kompozytowy”. Bez względu na to, jakiej nazwy użyjesz, otrzymasz potężne narzędzie. Zobaczmy więc, jak ono działa.

Definiowanie struktur

Aby utworzyć strukturę, użyj słowa kluczowego `struct` i odpowiedniej nazwy, a następnie podaj listę zmiennych wewnątrz nawiasów klamrowych. Dostęp do tych zmiennych można uzyskać na podstawie ich nazw, podobnie jak dostęp do elementów tablicy uzyskuje się za pomocą indeksu. Oto krótki przykład, który można wykorzystać z programem obsługującym konta bankowe:

```
struct transaction {
  double amount;
  int day, month, year;
};
```

Właśnie zdefiniowaliśmy nowy „typ”, którego możemy użyć w programie. Zamiast `int` lub `char[]` mamy `struct transaction`:

```
int main() {
  int count;
  char message[] = "Twoje pieniądze są u nas bezpieczne!";
  struct transaction bill, deposit;
  //...
}
```

Deklaracje zmiennych `count` i `message` powinny wyglądać znajomo. W następnym wierszu deklarujemy dwie kolejne zmienne, `bill` i `deposit`, które są strukturami typu `transaction`. Tego nowego typu można użyć w dowolnym miejscu, w którym wykorzystywane byłyby typy natywne takie jak `int`. Strukturami mogą być zarówno zmienne lokalne, jak i globalne. Można je przekazywać do funkcji lub zwracać z niej. Wykorzystywanie struktur w funkcjach opiera się przede wszystkim na wskaźnikach — to zagadnienie omówimy dokładniej w punkcie „Funkcje i struktury”.

[1] Ta uproszczona obsługa okazuje się bardzo wygodna. Kernighan i Ritchie poświęcają temu zagadnieniu cały rozdział w książce *Język ANSI C. Programowanie*. Oczywiście ich analiza jest bardziej szczegółowa od tej, którą przedstawiono w niniejszej książce, więc tym bardziej powinno Cię to zachęcić do przeczytania tej klasycznej pozycji.

Definicje struktur mogą być dość złożone. Nie istnieje realne ograniczenie związane z liczbą zmiennych, które mogą zawierać. Struktura może nawet mieć zagnieżdżone definicje innych struktur! Oczywiście nie powinieneś przesadzać, ale masz naprawdę dużą swobodę podczas ich tworzenia.

Przypisywanie wartości do składowych struktur i dostęp do nich

Po zdefiniowaniu typu struktury możesz zadeklarować i zainicjalizować odpowiednie zmienne, używając składni podobnej do tej, za pomocą której obsługuje się tablice. Jeśli na przykład wcześniej znałbyś wartości struktury, mógłbyś użyć nawiasów klamrowych, aby ją zainicjalizować:

```
struct transaction deposit = { 200.00, 6, 20, 2021 };
```

Kolejność wartości wewnątrz nawiasów musi być zgodna z kolejnością zmiennych występujących w definicji struktury. Możesz także utworzyć zmienną i dopiero później ją wypełnić. Aby wskazać, które pole chcesz zainicjalizować, używasz operatora kropki. Podajesz nazwę zmiennej struktury (w tym przypadku `bill` lub `deposit`), kropkę, a następnie składową, którą jesteś zainteresowany, taką jak `day` lub `amount`. Dzięki temu możesz wykonywać przypisania w dowolnej kolejności:

```
bill.day = 15;
bill.month = 7;
bill.year = 2021;
bill.amount = 56.75;
```

Bez względu na to, w jaki sposób wypełniłeś strukturę, używasz notacji z kropką, aby uzyskać dostęp do składowych. Aby na przykład wyświetlić szczegóły transakcji, trzeba podać zmienną typu `transaction` (`bill` lub `deposit`), kropkę, a wreszcie nazwę odpowiedniego pola:

```
printf("Wpłata w wysokości %0.2f PLN została zaakceptowana.\n", deposit.amount);
printf("Rachunek musi zostać uregulowany do dnia %d.%02d\n", bill.day, bill.month);
```

Te wewnętrzne elementy możemy wyświetlić na ekranie. Możemy też przypisać im nowe wartości. Możemy ich użyć w obliczeniach. Krótko mówiąc, ze składowymi struktury można zrobić dokładnie to samo, co z innymi zmiennymi. Celem struktury jest po prostu ułatwienie przechowywania danych mających wspólne zastosowania. Ale struktury również zachowują *odrębność* danych. Przeanalizuj instrukcję przypisania dla składowej `amount` zarówno w przypadku struktury `bill`, jak i `deposit`:

```
deposit.amount = 200.00;
bill.amount = 56.75;
```

Mimo że w obu instrukcjach użyto tego samego pola, nigdy nie pojawi się niepewność, które z nich należy wybrać. Jeśli na przykład w przypadku struktury `bill` będziemy chcieli uwzględnić pewną wartość podatku, nie będzie to miało wpływu na ilość pieniędzy w strukturze `deposit`:

```
bill.amount = bill.amount + bill.amount * 0.05;

printf("Rachunek końcowy: %0.2f zł\n", bill.amount); // 59,59 zł
printf("Wpłata: %0.2f zł\n", deposit.amount)         // 200,00 zł
```

Taka odrębność ma sens. Dzięki strukturom możesz traktować rachunki i wpłaty jako oddzielne podmioty, rozumiejąc jednocześnie, że ich szczegóły pozostają unikatowe.

Wskaźniki do struktur

Jeśli utworzysz poprawny typ kompozytowy, który zawiera właściwe dane, prawdopodobnie zaczniesz go wykorzystywać w coraz większej liczbie programów. Takich typów możesz używać do tworzenia zmiennych globalnych i lokalnych, parametrów, a nawet wartości zwracanych z funkcji. Jednak w praktyce częściej będziesz miał do czynienia ze wskaźnikami do struktur, a nie samymi strukturami.

Aby utworzyć (lub usunąć) wskaźnik do struktury, możesz użyć dokładnie tych samych operatorów i funkcji, które są dostępne dla typów prostych. Jeśli masz już strukturę, możesz pobrać jej adres za pomocą operatora `&`. Jeśli utworzyłeś instancję struktury za pomocą funkcji `malloc()`, wywołujesz `free()`, by zwolnić pamięć do sterty. Oto kilka przykładów użycia wskaźników do struktury `transaction`:

```
struct transaction tmp = { 68.91, 8, 1, 2020 };
struct transaction *payment;
struct transaction *withdrawal;

payment = &tmp;
withdrawal = malloc(sizeof(struct transaction));
```

Zmienna `tmp` jest typową strukturą `transaction`. Inicjalizujemy ją za pomocą nawiasów klamrowych. Zarówno `payment`, jak i `withdrawal` są deklarowane jako wskaźniki. Możemy przypisać adres zmiennej `transaction` do wskaźnika `payment` lub, jak w przypadku `withdrawal`, na stercie przydzielić pamięć (która zostanie zainicjalizowana później).

Przed rozpoczęciem inicjalizowania zmiennej `withdrawal` należy jednak wziąć pod uwagę, że jest ona wskaźnikiem, więc wymaga wyłuskania przed użyciem operatora kropki. Co więcej, operator ten ma wyższy priorytet niż operator wyłuskania, więc konieczne jest użycie nawiasów, aby uzyskać poprawny wynik. Może to być trochę skomplikowane, dlatego często stosuje się alternatywną notację. Operator „strzałki” (`->`) pozwala używać wskaźnika do struktury bez wykonywania operacji wyłuskania. Tak jak w przypadku operatora kropki, strzałkę umieszcza się między nazwą zmiennej będącej strukturą a nazwą jej składowej:

```
// Z operatorem wyłuskania:
(*withdrawal).amount = -20.0;

// Z operatorem strzałki:
withdrawal->day = 3;
withdrawal->month = 8;
withdrawal->year = 2021;
```

Na początku taki zapis może być trochę dezorientujący, ale w końcu się do niego przyzwyczaisz. Wskaźniki do struktur pozwalają na współdzielenie istotnych informacji przez różne części programu. Ich największą zaletą jest to, że nie wymagają przenoszenia ani kopiowania składowych struktur. Ta zaleta stanie się widoczna, gdy zaczniesz używać struktur z funkcjami.

Funkcje i struktury

Załóżmy, że chcielibyśmy utworzyć funkcję wyświetlającą ładnie sformatowaną informację o transakcji. Do funkcji moglibyśmy po prostu przekazać całą strukturę. W wierszu definiującym funkcję umieścilibyśmy parametr o typie `struct transaction`, a podczas jej wywołania przekazalibyśmy zwykłą zmienną:

```
void printTransaction1(struct transaction tx) {
  printf("%2d.%02d.%4d: %10.2f\n", tx.day, tx.month, tx.year, tx.amount);
}
// ...
printTransaction1(bill);
printTransaction1(deposit);
```

To dość proste rozwiązanie, jednak przypomnij sobie wyjaśnienie, w jaki sposób funkcje używają stosu. W tym przypadku w momencie wywołania funkcji `printTransaction1()` wszystkie pola struktury `bill` lub `deposit` będą musiały zostać umieszczone na stosie. Wymaga to dodatkowego czasu procesora i pamięci. Okazuje się, że w najwcześniejszych wersjach języka C takie rozwiązanie nie było dozwolone! Oczywiście teraz można już tak robić, ale pamiętaj, że przekazywanie wskaźników do funkcji, a także ich zwracanie jest nadal szybsze. Oto wskaźnikowa wersja funkcji `printTransaction1()`:

```
void printTransaction2(struct transaction *ptr) {
  printf("%2d.%02d.%4d: %10.2f\n",
      ptr->day, ptr->month, ptr->year, ptr->amount);
}
// ...
printTransaction2(&tmp);
printTransaction2(payment)
printTransaction2(withdrawal);
```

Jedynym elementem, który zostanie umieszczony na stosie, będzie adres obiektu `struct transaction`. Czy nie jest to dużo prostsze?

Przekazywanie wskaźników w ten sposób jest związane z interesującym, zaplanowanym efektem ubocznym — w funkcji można zmieniać zawartość struktury. Jak pamiętasz z rozdziału 5., jeśli nie są używane wskaźniki, argumenty do funkcji umieszcza się na stosie. Są one kopiami rzeczywistych zmiennych. Żadna operacja na tych argumentach wewnątrz funkcji nie wpływa na oryginalne wartości zmiennych dostępnych w kodzie wywołującym.

Jeśli jednak przekażemy wskaźnik, możemy go użyć do zmodyfikowania składowych struktury. Te zmiany będą trwałe, ponieważ będziemy je wprowadzać w oryginalnej strukturze, a nie jej kopii. Na przykład moglibyśmy utworzyć funkcję dodawania podatku do każdej transakcji:

```
void addTax(struct transaction *ptr, double rate) {
  double tax = ptr->amount * rate;
  ptr->amount += tax;
}

// ... z powrotem w funkcji main
  printf("Rachunek przed uwzględnieniem podatku: %.2f zł\n", bill.amount);
  addTax(&bill, 0.05);
  printf("Rachunek po uwzględnieniu podatku: %.2f zł\n", bill.amount);
// ...
```

Zauważ, że w funkcji `main()` nie zmieniamy wartości `bill.amount`. Po prostu do funkcji `addTax()` przekazujemy adres struktury wraz ze stawką podatku. Oto uzyskany wynik:

```
Rachunek przed uwzględnieniem podatku: 56.75 zł
Rachunek po uwzględnieniu podatku: 59.59 zł
```

Dokładnie tego oczekiwaliśmy. Przekazywanie struktur przez referencję jest skutecznym rozwiązaniem, dlatego stosuje się je bardzo często. Oczywiście nie każda zmienna musi zostać zorganizowana w postaci

struktury, jak również nie każda struktura musi zostać przekazana poprzez referencję. W przypadku złożonych programów dzięki użyciu struktur osiąga się poprawę organizacji kodu i jego wydajności, co zdecydowanie zachęca do takiego działania.

Możliwość modyfikowania zawartości struktury za pomocą wskaźnika jest zwykle pożądana. Jeśli jednak z jakiegoś powodu podczas używania wskaźnika do struktury *nie chcesz* zmieniać składowej, pamiętaj, aby niczego do niej nie przypisywać. Oczywiście zawsze możesz umieścić kopię wartości składowej w zmiennej tymczasowej, a następnie ją przetwarzać.

Podsumowanie składni wskaźnikowej

W tym rozdziale zaprezentowano na tyle dużo zawiłych elementów składni języka C, że warto je krótko podsumować:

- Za pomocą słowa kluczowego `struct` można definiować nowe typy danych.
- Operator kropki (`.`) pozwala uzyskać dostęp do zawartości struktury.
- Operator strzałki (`->`) umożliwia obsługę elementów struktury za pomocą wskaźnika.
- Miejsce na dane można przydzielać przy użyciu funkcji `malloc()`.
- Dostęp do tak przydzielonego obszaru pamięci uzyskuje się za pomocą operatorów `&` („adres”) i * („wyłuskanie”).
- Pamięć można zwolnić za pomocą funkcji `free()`.

Spójrzmy na te nowe pojęcia i definicje w pewnym kontekście. Oto odpowiedni program: *r06/structure.c*. Szczególnie ciekawe fragmenty kodu nie zostały wyróżnione w książce, lecz zamiast tego są opatrzone komentarzami. W ten sposób możesz szybko przetestować działanie interesujących Cię konstrukcji, jeśli właśnie jesteś w trakcie tworzenia programu:

```
// Dołączamy standardowy nagłówek stdio, a także stdlib, aby uzyskać dostęp
// do funkcji malloc() i free(), a także wartości NULL
#include <stdio.h>
#include <stdlib.h>

// Używamy słowa kluczowego struct, by stworzyć nowe typy kompozytowe
struct transaction {
  double amount;
  int month, day, year;
};

// Nowo utworzony typ może zostać użyty jako parametr funkcji
void printTransaction1(struct transaction tx) {
  printf("%2d.%02d.%4d: %10.2f\n", tx.day, tx.month, tx.year, tx.amount);
}

// Możemy również przekazać wskaźnik do struktury
void printTransaction2(struct transaction *ptr) {
  // Test, czy wskaźnik nie jest pusty
  if (ptr == NULL) {
    printf("Invalid transaction.\n");
```

```
  } else {
    // W porządku! Mamy transakcję, wyświetlmy więc jej szczegóły za pomocą operatora ->
    printf("%2d.%02d.%4d: %10.2f\n", ptr->day, ptr->month, ptr->year, ptr->amount);
  }
}

// Przekazanie wskaźnika do funkcji umożliwia w razie potrzeby
// modyfikowanie zawartości struktury
void addTax(struct transaction *ptr, double rate) {
  double tax = ptr->amount * rate;
  ptr->amount += tax;
}

int main() {
  // Możemy zadeklarować zmienne lokalne (lub globalne) o nowo utworzonym typie
  struct transaction bill;

  // Wartości początkowe umieszczamy wewnątrz nawiasów klamrowych
  struct transaction deposit = { 200.00, 6, 20, 2021 };

  // Możemy także zainicjalizować składowe struktury w dowolnej kolejności za pomocą operatora kropki
  bill.amount = 56.75;
  bill.month = 7;
  bill.day = 15;
  bill.year = 2021;

  // Struktury przekazujemy do funkcji tak, jak inne zmienne
  printTransaction1(deposit);
  printTransaction1(bill);

  // Możemy także utworzyć wskaźniki do struktur i przydzielić pamięć za pomocą funkcji malloc()
  struct transaction tmp = { 68.91, 8, 1, 2020 };
  struct transaction *payment = NULL;
  struct transaction *withdrawal;
  payment = &tmp;
  withdrawal = malloc(sizeof(struct transaction));

  // Dość skomplikowane wyłuskanie w celu uzyskania dostępu do składowej
  (*withdrawal).amount = -20.0;
  // Prostsza metoda: użycie operatora strzałki
  withdrawal->day = 3;
  withdrawal->month = 8;
  withdrawal->year = 2021;

  // Wskaźniki do struktur można przekazywać do funkcji
  printTransaction2(payment);
  printTransaction2(withdrawal);

  // Dodajemy podatek do rachunku za pomocą funkcji i przekazanego do niej wskaźnika
  printf("Rachunek przed uwzględnieniem podatku: %.2f zł\n", bill.amount);
  addTax(&bill, 0.05);
  printf("Rachunek po uwzględnieniu podatku: %.2f zł\n", bill.amount);

  // Na końcu trzeba zawsze zwolnić przydzieloną pamięć
  free(withdrawal);
}
```

Im częściej będziesz używać wskaźników i funkcji `malloc()` we własnych programach, tym szybciej zrozumiesz ich działanie i będziesz się czuć bardziej komfortowo. Tworzenie od podstaw programu, który rozwiązuje interesujący problem, zawsze pomaga ugruntować zrozumienie nowego zagadnienia. Oficjalnie daję Ci pozwolenie na zabawę ze wskaźnikami!

Kolejne kroki

W tym rozdziale omówiliśmy kilka bardzo zaawansowanych elementów języka. Przeanalizowaliśmy, w jaki sposób dane są przechowywane w pamięci oraz adresowane podczas używania operatorów `&`, `*`, `.` oraz `->`, a także funkcji `malloc()` i `free()`. W bardziej specjalistycznych książkach tym zagadnieniom poświęcono wiele miejsca, więc nie zniechęcaj się, jeśli będziesz musiał kilka razy przeczytać niektóre z podrozdziałów. Pamiętaj, że modyfikowanie przykładów kodu jest świetnym sposobem pozwalającym lepiej zrozumieć zaprezentowany materiał.

Dysponujemy już naprawdę imponującym zestawem narzędzi! Możemy więc zgłębiać złożone problemy i uzyskiwać dobre rezultaty. Okazuje się jednak, że w wielu przypadkach zadania, które staramy się rozwiązywać, nie są czymś nowym. Na dobrą sprawę wiele problemów (lub przynajmniej wiele ich składowych, które się pojawią, gdy podzielimy główne zadanie na łatwe do zarządzania fragmenty) zostało już rozwiązanych przez innych programistów. W następnym rozdziale przeanalizujemy, jak można wykorzystać te gotowe rozwiązania.

Programowanie niskopoziomowe? Czysta radość z czystego C!

Wydawałoby się, że język C najlepsze lata ma już za sobą. Opracowano w końcu mnóstwo łatwych w użyciu i efektywnych wysokopoziomowych języków programowania. W rzeczywistości język C wciąż okazuje się niezastąpiony w realizacji tak ważnych celów, jak programowanie sterowników, systemów operacyjnych, kart graficznych, a także niewielkich mikrokontrolerów o ograniczonych zasobach. Właśnie teraz, w czasach burzliwego rozwoju internetu rzeczy, język C udowadnia swoją przydatność. Wystarczy nauczyć się tworzyć w nim czysty kod o niewielkich rozmiarach.

W tym podręczniku do praktycznej nauki programowania w języku C szczególną uwagę zwrócono na pisanie kodu umożliwiającego uzyskanie wysokiej wydajności w środowiskach o bardzo małych zasobach. Znalazło się tu gruntowne omówienie podstaw dobrego programowania w języku C. Opisano struktury kontrolne, operatory, funkcje i inne elementy składni C, a także zasady dobrego programowania i wzorce, dzięki którym można zmniejszać rozmiar skompilowanego programu. Przeanalizowano również środowisko Arduino, które jest świetną platformą docelową dla niewielkich programów napisanych w C. Zawarte tu informacje przydadzą się jednak każdemu, kto chce się zająć programowaniem dla systemów wbudowanych.

W książce między innymi:

- podstawy języka C, w tym typy danych, przepływ sterowania i funkcje
- zarządzanie pamięcią i działanie programów w prostych urządzeniach
- tworzenie czytelnego i łatwego w utrzymaniu kodu w języku C
- optymalizacja kodu pod kątem wydajności
- testowanie istniejących bibliotek i tworzenie własnych

Marc Loy pracował w Sun Microsystems, gdzie prowadził szkolenia dotyczące Javy i Uniksa, aktualnie jest konsultantem i autorem artykułów technicznych. Zajmuje się rozwijaniem systemów wbudowanych i oprogramowania elektroniki osobistej.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
Sprawdź nasze szkolenia!
SZKOLENIA
AKADEMIA IT & BUSINESS
HELIONSZKOLENIA.PL
INFORMATYKA W NAJLEPSZYM WYDANIU
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-8631-0
9 788328 386310
Cena: 67,00 zł