Podstawy języka C

część 3

Kontynuujemy omawianie podstaw języka C.

Wskaźniki

Zajmiemy się teraz ważnym, a jednocześnie dla wielu osób trudnym zagadnieniem w języku C, jakim są wskaźniki.

Czym jest wskaźnik? Ogólnie mówiąc, jest to adres zmiennej, czyli liczba wskazująca, w którym miejscu pamięci operacyjnej znajduje się dana zmienna. W pierwszej chwili adres zmiennej może się wydać czymś niezbyt potrzebnym w pisaniu programu, jednak w C jest to bardzo często wykorzystywana dana.

Jednym z zastosowań wskaźników jest dostęp do elementów zmiennych tablicowych. Za pomocą zmiennej wskaźnikowej możemy przemieszczać się po elementach tablicy. Zadeklarujmy więc zmienną tablicową:

```
char napis[] = "Elektronika";
```

a następnie zmienną wskaźnikową:

```
char * w;
```

Nie, nie odbywa się tutaj mnożenie. Gwiazdka oznacza tu, że deklarujemy zmienną wskaźnikową (wskaźnik), która będzie zawierała adres danej typu char. Której zmiennej? Tego jeszcze nie wiadomo, gdyż nowo zadeklarowana zmienna nie została jeszcze zainicjalizowana i jest w tej chwili bezużyteczna.

Zainicjalizujmy więc wskaźnik:

```
w = napis;
```

Tak prosto? Tak, w języku C nazwa zmiennej tablicowej jest wskaźnikiem na jej pierwszy element. Co więcej, w tej chwili możemy używać zamiennie zmiennych w oraz napisu:

```
printf("%s\n", napis);
printf("%s\n", w);
```

Możemy traktować zmienną w jak tablicę:

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

Zostanie wyświetlona litera e. OK, jest to ciekawe, ale nadal nie wiemy, co nam taka zmienna wskaźnikowa daje. Skoro jest to zmienna, to zmodyfikujmy ją, np. zwiększając o 1:

```
w++;
```

Wykonajmy ponownie linijki:

```
printf("%s\n", w);
printf("%c\n", w[2]);
```

Pojawił się napis "lektronika" oraz litera k. Przesunęliśmy bowiem wskaźnik o jeden bajt do przodu.

Wiemy już, że do danych wskazywanych przez wskaźnik możemy odwoływać się za pomocą notacji tablicowej. Zazwyczaj jednak robi się to za pomocą operatora wyłuskania. Tym operatorem znów jest gwiazdka:

```
char z = *w;
printf("%c\n", z);
printf("%c\n", *w);
printf("%c\n", *(w+1));
```

W powyższym przykładzie deklarowana jest nowa zmienna, której przypisywana jest wartość wskazywana przez zmienną w. Kolejne wywołanie funkcji printf wyświetla tę samą daną, ale już bez pośrednictwa zmiennej z. W trzecim wywołaniu wyświetlana jest zawartość bajtu, który znajduje się za bajtem wskazywanym przez w.

Dzięki wskaźnikom można swobodnie „jeździć" po pamięci. Jest to potężne narzędzie w języku C, ale jednocześnie niebezpieczne. Łatwo bowiem strzelić sobie w stopę, odwołując się przez przypadek do niewłaściwego miejsca w pamięci. Jeśli za pomocą wskaźnika odwołujemy się do tablicy, nie ma sprawdzania, czy nie wyszliśmy poza jej obszar. Można więc nie tylko odczytać nie te dane, które się chciało, ale też przypadkowo zmodyfikować inne zmienne. Jeśli odwołamy się do obszaru pamięci poza przestrzenią naszego procesu, system operacyjny zamknie nasz program i zostanie wyświetlony komunikat o wykonaniu nieprawidłowej operacji. Jeśli jednak będzie to obszar wewnątrz naszego procesu, najprawdopodobniej nie będzie żadnego komunikatu, a jedynie nieprawidłowość w działaniu naszego programu, niekoniecznie widoczna od razu. Rozważmy przykład:

```
char napis[] = "Elektronika";
int a = 10;
char * w;
w = napis;
w -= 12;
*w = 100;
printf("%d\n", a);
```

Jeśli kod ten zostanie skompilowany pod Visual C++ 2010, zauważymy, że za pomocą wskaźnika w zmodyfikowaliśmy zmienną a i jej wartość wynosi 100, a nie 10. W przypadku kompilatora GCC zmienna a jest w pamięci za zmienną napis i trzeba do wskaźnika w dodać wartość 12, a nie odjąć. Jeśli to samo zrobimy w Visual C++ (w += 12;), przy wychodzeniu z funkcji włączy się ochrona stosu i program zostanie zamknięty. Zainteresowani mogą wygooglować hasło „stack cookies" oraz opis działania przełącznika /GS w Visual C++.

Przekazywanie argumentów funkcji przez wskaźniki.

W języku C używamy często wskaźników do przekazywania danych do funkcji. Parametry bowiem są przekazywane przez wartość. Jeśli chcemy przekazać do funkcji zmienną tak, aby funkcja mogła tę zmienną zmodyfikować, używamy wkaźnika. Popatrzmy na przykład:

```
#include <stdio.h>
void dodaj(int a);
int main() {
  int a = 3;
  dodaj(a);
  printf("%d\n",a);
  return 0;
}
void dodaj(int a) {
  a += 5;
  printf("%d\n",a);
}
```

W wyniku dostaniemy liczby 8 oraz 3. Wartość 3 została przekazana do funkcji i tam zwiększona, wyświetlone zostało 8. Jednak zmienna w funkcji main nadal miała wartość 3, co zostało potem wyświetlone. Problem możemy rozwiązać przez zwracanie wartości:

```
#include <stdio.h>
int dodaj(int a);
int main() {
  int a = 3;
  a = dodaj(a);
  printf("%d\n",a);
  return 0;
}
int dodaj(int a) {
  a += 5;
  printf("%d\n",a);
  return a;
}
```

Zmieniliśmy zwracany typ z void na int, a zwracana wartość przypisywana jest do zmiennej. Problem jednak w tym, że nie zawsze możemy skorzystać z tego sposobu. Często potrzebujemy zwracać kilka wartości. Tutaj z pomocą przychodzą wskaźniki:

```
#include <stdio.h>
void dodaj(int * a);
int main() {
  int a = 3;
  dodaj(&a);
  printf("%d\n",a);
  return 0;
}
void dodaj(int * a) {
  *a += 5;
  printf("%d\n", *a);
}
```

Deklarujemy funkcję dodaj jako przyjmującą jako argument wskaźnik na zmienną typu int. Aby taki wskaźnik uzyskać, stosujemy operator &.

Wczytywanie danych z klawiatury

Często używaną funkcją pobierającą wskaźnik jest scanf. Jest to funkcja analogiczna do printf, jednak zamiast do wyświetlania danych, służy do ich wprowadzania z klawiatury:

```
#include <stdio.h>

int main() {
  int a;
  printf("Podaj liczbe calkowita: ");
  scanf("%d", &a);
  printf("Wprowadzona liczba to: %d\n", a);
  return 0;
}
```

Tak jak w przypadku printf, do obsługi tekstów służy specyfikator %s. Z powodów wcześniej omówionych, scanf musi znać rozmiar buforu, do którego będzie kopiować znaki. W przypadku liczb całkowitych scanf spodziewa się wskaźnika na zmienną typu

int i zakłada rozmiar 4 bajtów. W przypadku łańcuchów tekstowych rozmiar nie jest określony i konieczne jest podanie rozmiaru bufora po znaku %:

```
#include <stdio.h>

int main() {
  char bufor[30];
  printf("Wpisz tekst: ");
  scanf("%29s", bufor);
  printf("Wprowadzony tekst: %s\n", bufor);
  return 0;
}
```

Podany rozmiar musi być o 1 mniejszy od rozmiaru bufora, aby było jeszcze miejsce na zero kończące łańcuch.

Dynamiczna alokacja pamięci

Wskaźniki używane są też m.in. przy dynamicznej alokacji pamięci. Nie zawsze wiadomo na etapie pisania programu, jak dużo danych będzie przetwarzanych. Ponadto nieraz potrzebny jest duży obszar pamięci tylko przez pewien czas. Nie ma więc sensu, żeby była ona zaalokowana bez przerwy. Dlatego też stosuje się dynamiczną alokację pamięci, program prosi system operacyjny o przydzielenie fragmentu pamięci o danym rozmiarze. Standardową funkcją używaną w tym celu jest malloc z biblioteki stdlib.h:

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

int main() {
  char * bufor = (char *) malloc(30);
  if (!bufor) return 1;
  printf("Wpisz tekst: ");
  scanf("%29s", bufor);
  printf("Wprowadzony tekst: %s\n", bufor);
  free(bufor);
  return 0;
}
```

Funkcja malloc pobiera liczbę bajtów do zaalokowania i zwraca wskaźnik na zaalokowany obszar. Zwracana wartość jest typu void *. My używamy wskaźnika na typ char, w związku z tym potrzebne jest rzutowanie. Operacja rzutowania nie wpływa na zmienne. Informuje kompilator, że chcemy wykonać operację przypisania pomimo różnicy typu danych. Rzutowanie wykonujemy, podając typ docelowy w nawiasach. Alokacja dynamiczna nie musi się powieść. Jeśli system operacyjny ma zbyt mało pamięci operacyjnej, zwrócony zostanie pusty wskaźnik (NULL). W tej sytuacji nasz program kończy działanie poprzez wyjście z funkcji main, zwracając kod błędu 1. To, czy wskaźnik nie jest pusty (nie jest zerem), sprawdzamy za pomocą operatora zaprzeczenia. Jeśli alokacja się powiedzie, program dalej wykonuje się tak jak w wersji z alokacją statyczną. Na koniec zaalokowana pamięć jest zwracana z powrotem do systemu operacyjnego.

Korzystając z dynamicznej alokacji pamięci, należy pamiętać o jej prawidłowym zwalnianiu. Standardową funkcją do zwalniania zaalokowanej pamięci jest free(). Niezwalnianie prowadzi do tzw. wycieków pamięci – program zajmuje więcej pamięci, niż powinien, marnując ją. Niejednokrotnie źle napisany program alokuje coraz więcej pamięci, powodując konieczność coraz częstszego odwoływania się przez system operacyjny do pliku wymiany, a tym samym do coraz wolniejszej pracy. Podobnie jak posługiwanie się wskaźnikami, ręczne zarządzanie pamięcią wymaga w języku C odpowiedniej dyscypliny od programisty.

Wskaźniki podwójne

Funkcja malloc przekazuje wskaźnik na zaalokowaną pamięć, po prostu go zwracając. Czy wskaźnik można przekazać też inaczej? Otóż można – poprzez argument. W języku C jednak nie ma znanego z C++ przekazywania zmiennych przez referencję, konieczne jest skorzystanie z podwójnego wskaźnika. Można to zrobić tak:

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

void myalloc(int size, char ** buf);

int main() {
  char * buffer = NULL;
  myalloc(30, &buffer);
  if (!buffer) return 1;
  printf("Wpisz tekst: ");
  scanf("%29s", buffer);
  printf("Wprowadzony tekst: %s\n", buffer);
  free(buffer);
  return 0;
}

void myalloc(int size, char ** buf){
  *buf = (char *) malloc(size);
}
```

W powyższym przykładzie funkcja myalloc pobiera podwójny wskaźnik, czyli inaczej wskaźnik na wskaźnik. Przekazywany jest do niej adres wskaźnika buffer, czyli wskaźnik na wskaźnik buffer. Wskaźnik ten jest na początku tzw. pustym wskaźnikiem, czyli zawierającym adres 0 (NULL). Nie możemy przekazać po prostu wskaźnika buffer, bo funkcja myalloc otrzymałaby liczbę 0. Przekazując podwójny wskaźnik, przekazujemy nie liczbę 0, ale adres w pamięci, pod którym (obecnie) znajduje się liczba 0. Mając adres miejsca w pamięci, możemy zapisać pod nim inną wartość, np. adres zwrócony przez malloc. Dlatego też nie modyfikujemy argumentu buf, ale miejsce w pamięci wskazywane przez buf.

Dla dociekliwych: z punktu widzenia architektury powyższego programu, funkcja myalloc jest oczywiście zbędna, ponieważ nic nie wnosi względem bibliotecznej funkcji malloc. Jest tylko jej opakowaniem (ang. wrapper). Mogłaby też być typu char * i zwracać wskaźnik za pomocą return. Tutaj jednak służy nam za ilustrację działania podwójnych wskaźników. Ponadto można w różnych bibliotekach spotkać funkcje zwracające tablice w ten właśnie sposób, a za pomocą return zwracające np. kod błędu.

Oprócz wskaźników pojedynczych czy podwójnych można oczywiście stosować także wskaźniki wyższego poziomu. W praktyce jednak rzadko się je spotyka.

Wskaźniki na funkcje

Wskaźnik może służyć także do przechowywania adresu funkcji, czyli miejsca w pamięci, w którym zaczyna się jej kod. Takiego wskaźnika możemy użyć do wywołania funkcji bez znajomości jej nazwy w danej chwili, podobnie jak w przypadku zmiennych. Wystarczy wcześniejsze odpowiednie zainicjowanie. Do czego może nam się przydać wskaźnik na funkcję? Jest dobrym rozwiązaniem właśnie wtedy, gdy nie znamy nazwy funkcji, jaką chcemy wywołać. Ale jak to możliwe? Przecież jeśli chcemy wywołać funkcję, to ją wywołujemy. Zwykle tak jest, jednak czasem sytuacja jest bardziej skomplikowana. Załóżmy, że piszemy bibliotekę, w której znajduje się funkcja wywołująca funkcję użytkownika, tzw. callback. Bez wskaźników na funkcje użytkownik biblioteki musiałby stworzyć funkcję o określonej z góry nazwie. Nie mógłby też w łatwy sposób używać różnych funkcji dla różnych wywołań funkcji bibliotecznej. Dzięki wskaźnikom na funkcje zyskuje się większą elastyczność.

Deklaracja wskaźnika na funkcję wygląda podobnie do deklaracji wskaźnika na zmienną:

```
zwracany_typ (*nazwa)(typy argumentów);
```

Przykładowo, wskaźnik na funkcję pobierającą dwa argumenty typu int i zwracający typ int możemy zadeklarować następująco:

```
int (*func)(int, int);
```

Aby zainicjalizować go, żeby pokazywał na określoną funkcję, podajemy jej nazwę wraz z opcjonalnym operatorem adresu. Nie używamy nawiasów za nazwą funkcji.

```
func = &add;
```

Oczywiście sygnatura przykładowej funkcji add musi się zgadzać z deklaracją wskaźnika, czyli musi zwracać taki sam typ i pobierać takie same argumenty. Intuicja podpowiada, że wywołanie funkcji wskazywanej przez wskaźnik wymaga użycia operatora wyłuskania (gwiazdki):

```
int x = (*func)(2, 3);
```

Podobnie jednak jak przy inicjalizacji wskaźnika, język C pozwala pominąć operator wyłuskania:

```
int x = func(2, 3);
```

Popatrzmy na kod w ramce na następnej stronie. Wskaźnik na funkcję użyty jest w celu przekazania do funkcji wyświetlającej zawartości tablicy warunku, który musi być spełniony, aby dany element tablicy został wyświetlony.

Funkcję printConditionally wywołujemy dwa razy, za każdy razem przekazując tę samą tablicę. Przekazujemy jednak wskaźniki na różne funkcje, jedną sprawdzającą parzystość, a drugą sprawdzającą, czy liczba jest większa od 100. Funkcja printConditionally iteruje po przekazanej tablicy i dla każdego elementu uruchamia funkcję przekazaną przez wskaźnik. W zależności od zwróconej wartości wyświetla dany element lub nie. W programie zdefiniowano typ wskaźnikowy dla zwiększenia czytelności.

Zaprezentowana w przykładzie funkcja printConditionally jest swego rodzaju tzw. iteratorem, czyli mechanizmem pozwalającym wykonać

```
#include <stdio.h>
typedef int (*condFunc)(int);

void printConditionally(int numbers[], int size, condFunc condition) {
  int i;
  for (i = 0; i < size; i++) {
    if ((*condition)(numbers[i])) printf("%d\n", numbers[i]);
  }
}

int even(int n) {
  return (n % 2) == 0;
}
int bigNumber(int n) {
  return n > 100;
}

int main(void) {
  int numbers[] = {3, 14, -6, 432, -93, 17, 809};
  printf("Liczby parzyste:\n");
  printConditionally(numbers, sizeof(numbers)/sizeof(int), &even);
  printf("Liczby wieksze od 100:\n");
  printConditionally(numbers, sizeof(numbers)/sizeof(int), &bigNumber);
  return 0;
}
```

daną akcję dla wszystkich elementów danego zbioru. Co prawda można było użyć bezpośrednio pętli for (lub while) w głównym kodzie, bez uciekania się do tworzenia specjalnej funkcji, jednak dzięki przedstawionemu rozwiązaniu możemy bardziej skupić się na tym, co chcemy osiągnąć (wykonać akcję dla każdego elementu zbioru), niż zajmować się szczegółami implementacyjnymi (tworzeniem zmiennej licznikowej, inicjalizowaniem jej, sprawdzaniem zakresu oraz jej inkrementacją). W wielu językach obsługa iteratorów jest wbudowana, w C można je symulować z wykorzystaniem wskaźników na funkcje.

Typy złożone

Język C ma kilka podstawowych typów danych. Wystarczają one jednak zwykle tylko w bardzo prostych programach. Często operujemy na zestawach powiązanych ze sobą danych i przydałyby się typy złożone. W C mamy do dyspozycji struktury:

```
struct person_s {
  int age;
  char name[30];
};
```

Powyżej zdefiniowana została struktura składająca się z dwóch pól: age o typie int oraz name, będące 30-elementową tablicą typu char. Gdy mamy zdefiniowaną strukturę, możemy zadeklarować zmienną typu strukturalnego:

```
struct person_s person;
```

W przypadku zmiennych strukturalnych konieczne jest używanie słowa kluczowego struct. Aby odwołać się do pola zmiennej strukturalnej, po jej nazwie używamy operatora kropki oraz podajemy nazwę pola:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  struct person_s {
   int age;
   char name[30];
  };
  struct person_s p;
  printf("Podaj imie: ");
  scanf("%s29", p.name);
  printf("Podaj wiek: ");
  scanf("%d", &p.age);
  printf("Imie: %s\n", p.name);
  printf("Wiek: %d\n", p.age);
  return 0;
}
```

W powyższym przykładzie widzimy, jak korzystać ze struktur, ale nadal pola są traktowane jako oddzielne zmienne. Rozbudujmy więc program o funkcje do wczytywania i wyświetlania danych.

```
#include <stdio.h>
struct person_s {
  int age;
  char name[30];
};
void readData(struct person_s *);
void printData(struct person_s *);
int main() {
  struct person_s person;
  person.age = 0;
  person.name[0] = NULL;
  readData(&person);
  printData(&person);
  return 0;
}
void readData(struct person_s * p){
  printf("Podaj imie: ");
  scanf("%s29", p->name);
  printf("Podaj wiek: ");
  scanf("%d", &p->age);
}
void printData(struct person_s * p) {
  printf("Imie: %s\n", p->name);
  printf("Wiek: %d\n", p->age);
}
```

Ponieważ nasza struktura jest używana nie tylko w funkcji main, jej definicja została wyniesiona poza main, przed deklaracje funkcji z niej korzystających. Po deklaracji zmiennej person inicjalizowane są jej pola. Nie jest to konieczne do działania powyższego programu, jednak kompilator może zwrócić błąd przy braku inicjalizacji (VC++ tak robi, gcc nie), a z drugiej inicjalizowanie zmiennych jest dobrą praktyką. Następnie nasza struktura jest przekazywana do funkcji. Tutaj właśnie widać przydatność struktur: możemy kilka wartości zebrać w jedną zmienną i przekazać jako jeden parametr. Jeśli zmodyfikujemy definicję struktury, nie musimy modyfikować deklaracji funkcji ani ich wywołań.

Do funkcji readData() przekazujemy wskaźnik na strukturę. Dzięki temu funkcja może zmodyfikować pola przekazywanej struktury, a nie działa na jej kopii. W przypadku funkcji printData przekazywanie przez wskaźnik nie jest konieczne, jednak bez tego następowałoby niepotrzebne kopiowanie struktury podczas jej przekazywania. Może się wydawać, że kilkadziesiąt bajtów to niewiele, jednak dobrą praktyką jest unikanie niepotrzebnego zajmowania pamięci. Poza tym, jeśli chcemy programować mikrokontrolery, nawet kilkadziesiąt bajtów może być dużym fragmentem pamięci.

Uważni Czytelnicy zapytają: a co się stało z kropkami? Co to za dziwne strzałki? Otóż zmienna wskaźnikowa nie jest zmienną strukturalną, jest tylko adresem, nie ma pól. Aby użyć kropki, trzeba by użyć operatora wyłuskania (gwiazdki), np.:

```
scanf("%s29", (*p).name);
```

Można też użyć specjalnego operatora strzałki, pozwalającego na dostęp do pól zmiennej wskazywanej przez wskaźnik na nią.

Pewną rzadko używaną techniką w języku C są pola bitowe. Są to po prostu struktury z określoną liczbą bitów, jaką może zajmować dane pole, co jest przydatne przy odwzorowywaniu układu bitów rejestrów sprzętowych. Jeśli przykładowo mamy pole typu int (dopuszczalne typy to int, unsigned int oraz _Bool w C99), które będzie przyjmować wartości od 0 do 10, to możemy ograniczyć jego rozmiar do 4 bitów:

```
#include <stdio.h>
int main(void) {
  struct poleBitowe {
    int pole1 : 4;
    int pole2 : 4;
    int pole3 : 4;
    int pole4 : 4;
  } pole;
  printf("%d\n", sizeof(pole));
  return 0;
}
```

Wyświetlona zostanie wartość 4. Warto jednak pamiętać o pewnej rzeczy, dotyczącej struktur ogólnie, nie tylko ich specjalnej postaci, jaką są pola bitowe. Otóż pola struktury są wyrównywane w pamięci tak, aby ich adres dzielił się przez ich rozmiar. Np. adres pola typu int będzie się dzielił przez 4. Załóżmy, że mamy dwa pola: typu char oraz typu int. Jeśli zadeklarujemy je w tej kolejności, to struktura będzie miała nie 5, ale 8 bajtów, ze względu na konieczność wyrównania pola typu int. Trzy bajty zostaną więc zmarnowane. Jeśli zadeklarujemy pola odwrotnie, to pole typu char będzie zaraz za polem typu int i struktura zajmie 5 bajtów. Tak samo jest z polami bitowymi. Ze względu na wyrównywanie, zmniejszanie rozmiaru pól może nie dać żadnych rezultatów albo dużo mniejsze od oczekiwanych.

Co więcej, trzeba też rozpatrywać inne zmienne. Jeśli np. za wspomnianą 5-bajtową strukturą umieścimy deklarację zmiennej typu int, to oszczędność 3 bajtów okaże się iluzoryczna: ze względu na wyrównanie tej zmiennej i tak powstanie niewykorzystana dziura o rozmiarze 3 bajtów.

Ciąg dalszy artykułu w następnym numerze.

Grzegorz Niemirowski
grzegorz@grzegorz.net

Uwaga! Bardzo prosimy przysyłać na adres e-mailowy: edw@elportal.pl wszelkie uwagi, opinie, a także informacje o obawach, problemach i sukcesach związanych z kursem C.