15.13 Efekty uboczne

Definicja 40 *Efekt uboczny*[68] funkcji jest to zmiana w treści tej funkcji wartości zmiennej, która nie jest zmienną lokalną funkcji.

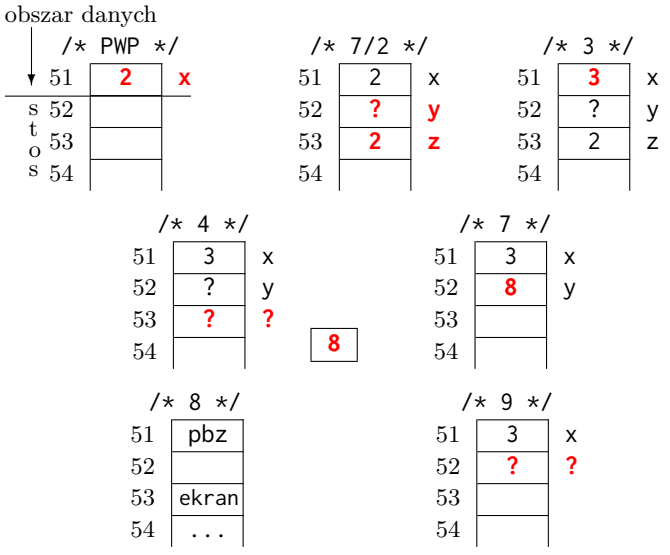
Efekty uboczne zmniejszają czytelność programu i prowadzą do trudno wykrywalnych błędów. Czasami są jednak użyteczne — tak jak w przypadku przekazywania do funkcji adresu tablicy.

Przykład (Zadanie 39) Przeczytaj poniższy program i bez śledzenia stanów pamięci określ, jaki napis będzie wypisany na ekran. Sprawdź rozwiązanie analizując stany pamięci.

```
int x = 2;                                  /*1*/

int do_trzeciej(int z) {                    /*2*/
   x++;                                     /*3*/
   return z * z * z;                        /*4*/
}                                           /*5*/
main() {                                    /*6*/
   int y = do_trzeciej(x);                  /*7*/
   printf("%d do trzeciej to %d", x, y);    /*8*/
}                                           /*9*/
```

Funkcja `do_trzeciej` zwraca podniesiony do trzeciej potęgi argument. Jej efektem ubocznym jest zmiana wartości zmiennej globalnej `x`. Czytający funkcję `main` może spodziewać się, że na ekranie pojawi się napis `2 do trzeciej to 8`, ale jest inaczej.

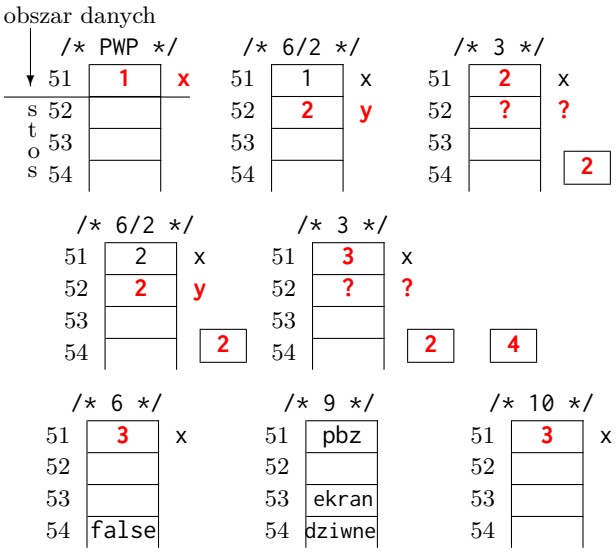


■

Przykład (Zadanie 40) Przeczytaj poniższy program i bez śledzenia stanów pamięci określ, jaki napis zostanie wypisany na ekran. Potem sprawdź rozwiązanie analizując stany pamięci.

```
int x = 1;                      /*1*/

int f(int y) {                  /*2*/
   return y * x++;              /*3*/
}                               /*4*/
main() {                        /*5*/
   if (f(2) == f(2))            /*6*/
      printf("dobrze");         /*7*/
   else                         /*8*/
      printf("dziwne");         /*9*/
}                               /*10*/
```



■

15.14 Funkcje rekurencyjne

Definicja 41 *Funkcja bezpośrednio rekurencja*[69] jest to taka funkcja, w której treści znajduje się wywołanie jej samej.

Przykład (Zadanie 53) Poniższy program przedstawia zastosowanie rekurencji do sprawdzenia, czy słowo jest palindromem, czyli czy czytane wspak będzie tym samym słowem. Zadanie można rozwiązać tak.

- Jeśli długość słowa wynosi 0 lub 1, to słowo jest palindromem (przyjmujemy, że słowo puste jest palindromem).
- Sprawdźmy, czy pierwszy i ostatni znak słowa są równe.
  - Jeśli nie, to dalej nic nie trzeba sprawdzać — słowo nie jest palindromem.
  - Jeśli tak, to „wyrzućmy” pierwszy i ostatni znak i sprawdźmy, czy tak powstałe słowo jest palindromem stosując opisany algorytm.

Dla słowa `kajak` metoda działa tak.

- Porównujemy pierwszy i ostatni znak słowa | k | a | j | a | k |.
- Są takie same. „Wyrzucamy” oba znaki.
- Porównujemy pierwszy i ostatni znak słowa | a | j | a |.
- Są takie same. „Wyrzucamy” oba znaki.
- Pozostało słowo jednoznakowe | j |.
- Jest ono palindromem, więc całe słowo `kajak` jest palindromem.

[68] ang. side effect

[69] ang. direct recursive function

„Wyrzucanie" jest czynnością umowną. W rzeczywistości słowo jest zapisane w tablicy i nigdy nie modyfikujemy tej tablicy. Nie kopiujemy też podsłów do innych tablic. Funkcja `jest_palindrom` dostaje **adres** pierwszego znaku słowa, które sprawdzamy (adres jednego z elementów tablicy) oraz długość słowa. Wie zatem, w której części tablicy znajduje się sprawdzane słowo.

```
/**
 * Sprawdza, czy podane slowo jest palindromem.
 * @param  slowo adres poczatku tablicy, w ktorej
 *               jest zapisane sprawdzane slowo;
 *               ta tablica moze byc 'podtablica'
 *               wiekszej tablicy, wtedy jest to
 *               adres pewnego elementu wiekszej
 *               tablicy
 * @param  dl     dlugosc slowa
 * @return 1   -  gdy slowo jest palindromem
 *         0   -  w p.p.
 */
int jest_palindrom(char *slowo, int dl);      /*1*/

main() {                                      /*2*/
   char *slowo = "ala";                       /*3*/
   int dl = strlen(slowo);                    /*4*/
   if (jest_palindrom(slowo, dl))             /*5*/
      printf("%s jest palindromem", slowo);   /*6*/
   else                                       /*7*/
      printf("%s nie jest palindromem",slowo);/*8*/
}                                             /*9*/

int jest_palindrom(char *slowo, int dl) {     /*10*/
   if (dl <= 1)                               /*11*/
      return 1;                               /*12*/
   else                                       /*13*/
      return (slowo[0] == slowo[dl - 1])
      && jest_palindrom(slowo + 1, dl - 2);   /*14*/
}                                             /*15*/
```

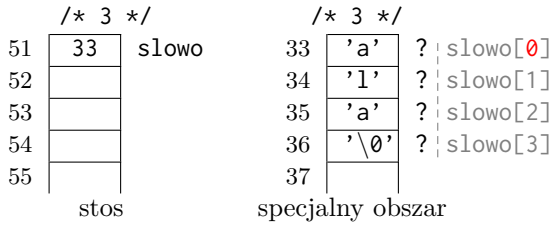
Funkcja `jest_palindrom` wywołuje w treści samą siebie, jest zatem funkcją rekurencyjną. Jej treść zaczyna się od przypadku słowa jednoznakowego lub pustego. Wtedy funkcja zwraca 1, co oznacza, że słowo jest palindromem.

Jeśli długość słowa jest większa, to wynikiem jest wartość koniunkcji z linii 14. Pierwsze zdanie koniunkcji jest porównaniem pierwszego i ostatniego znaku. Jeśli to zdanie jest fałszywe, to wartość całej koniunkcji jest fałszywa i drugie zdanie nie jest obliczane. Jeśli to zdanie jest prawdziwe, to trzeba obliczyć wartość logiczną drugiego zdania, które jest wywołaniem funkcji `jest_palindrom` dla tablicy zaczynającej się o jeden element dalej i długości o 2 mniejszej. Od wartości tego zdania zależy wartość całej koniunkcji.

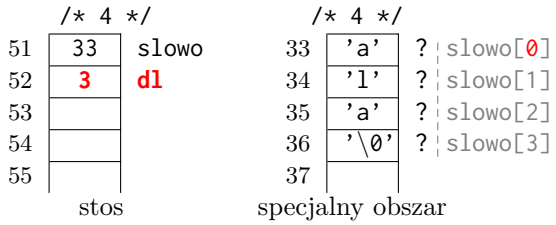
Prześledźmy działanie tego programu. Program zaczyna swe wykonanie od linii 3. Na stosie alokujemy wskaźnik. Przypisujemy jemu wartość początkową w sposób, który jeszcze dotąd nie był omawiany. Instrukcja ta powoduje, że wskaźnik zawiera adres początku tablicy, w której jest zapisane słowo `ala`.

Występujący w programie napis otoczony cudzysłowami nazywa się *literałem napisowym* (np. `"ala"`). Każde wystąpienie literału napisowego powoduje, że kompilator rezerwuje w pamięci tablicę, której elementy są typy `char`. Każdy element zawiera jeden znak. Elementów tej tablicy nie wolno zmieniać, można je tylko odczytywać.

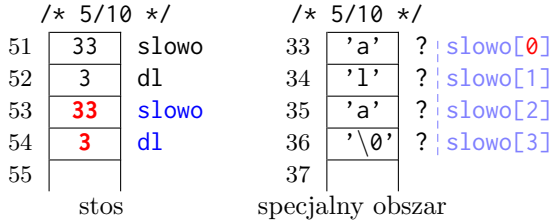
Tablica ma długość o 1 większą niż długość słowa. Ostatni element jest potrzebny na zapisanie znaku końca napisu, kodowanego przez `'\0'`. W pamięci kompilator nie zapisuje, jaką długość ma napis, tylko wstawia znak końca napisu. Na tej podstawie jest możliwe obliczenie długości napisu. Operacje na napisach są zaimplementowane w bibliotece `string.h`. Wszystkie zawarte tam funkcje wymagają, by napis był zakończony znakiem końca napisu.



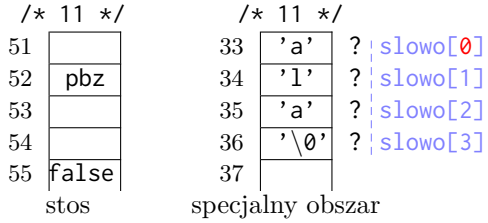
W linii 4 alokowana jest na stosie zmienna `dl` i zapisywana w niej długość napisu. Do obliczenia długości jest użyta funkcja `strlen` z biblioteki `string.h`.



Sprawdzanie warunku z linii 5 zaczyna się od wywołania funkcji `jest_palindrom`. Tak jak zawsze, parametry alokowane są na stosie i nadawane są im wartości początkowe z miejsca wywołania.



Warunek w linii 11 jest fałszywy.



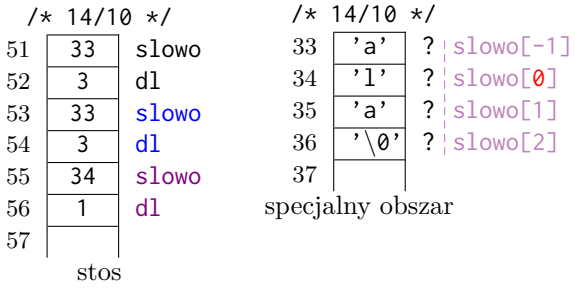
W linii 14 najpierw obliczamy wartość warunku logicznego. Pierwsze zdanie jest prawdziwe. Podczas obliczania drugiego zdania koniunkcji następuje wywołanie funkcji. Tak jak zawsze, parametry alokowane są na stosie i nadawane są im wartości początkowe z miejsca wywołania.

W tym momencie na stosie są

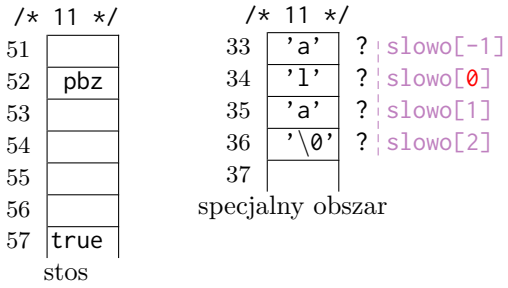
1. zmienne `slowo` i `dl` z funkcji `main`,
2. parametry `slowo` i `dl` powstałe podczas pierwszego wywołania funkcji `jest_palindrom`,

3. parametry `slowo` i `dl` powstałe podczas drugiego wywołania funkcji `jest_palindrom`.

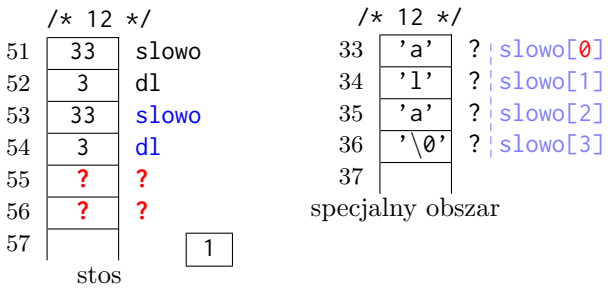
Ponieważ podczas drugiego wywołania parametr `slowo` został zainicjalizowany inną wartością niż przy pierwszym wywołaniu, to doszło do przeindeksowania tablicy (z II konwencji).



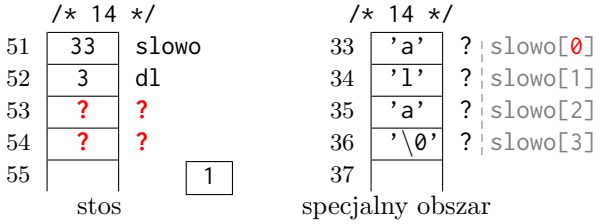
Warunek w linii 11 jest prawdziwy.



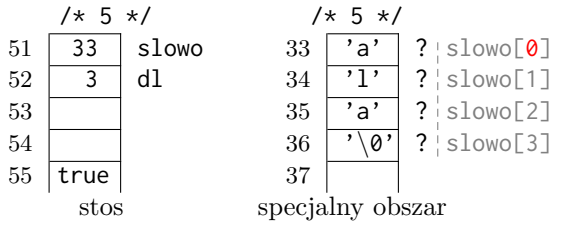
W linii 12 następuje zapisanie wartości zwracanej i dealokacja parametrów z drugiego wywołania.



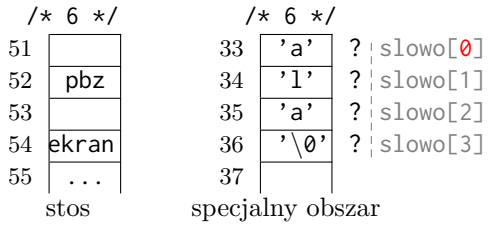
Wracamy do miejsca wywołania, czyli linii 14. Ta linia, przed drugim wywołaniem funkcji, była częściowo obliczona (pierwsze zdanie koniunkcji było prawdziwe). Teraz okazuje się, że drugie też jest (wartość przed chwilą zwrócona wynosi 1). Zatem całe zdanie jest prawdziwe i obliczamy znów wartość zwracaną. Dealokujemy parametry pierwszego wywołania.



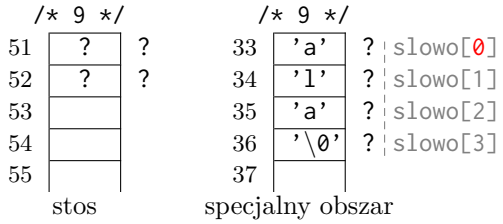
Wracamy do miejsca wywołania, czyli linii 5. Wartość zwrócona stanowi warunek `if`, jest on prawdziwy.



W linii 6 wypisywany jest komunikat na ekran.



W linii 9 następuje dealokacja zmiennych. Literał napisowy będzie dealokowany po wykonaniu programu.



■

Rekurencja może być także *pośrednia*[70]. Dzieje się tak wówczas, gdy funkcja `A` wywołuje w swojej treści `B`, a `B` wywołuje `A`. Łańcuch zależności może być też dłuższy, np. `A` wywołuje `B`, `B` wywołuje `C`, `C` wywołuje `D`, `D` wywołuje `A`.

Programista musi zadbać o zakończenie rekursji, żeby nie dopuścić do sytuacji nieskończonego ciągu wywołań, np.

```
void A(int j) {
   printf("%d \n", j);
   A(j + 1);
}
```

Zmienne lokalne (które nie są statyczne) oraz parametry, w przypadku funkcji rekurencyjnych są tworzone na nowo dla każdego wywołania. Bez tej możliwości rekurencja nie byłaby użyteczna.

Zalety rekurencji:

- ułatwia często zapisanie programu i zwiększa czytelność.

Wady rekurencji:

- programy rekurencyjne są zwykle wolniejsze od ich nierekurencyjnych wersji,
- występuje możliwość przepełnienia stosu — każde wywołanie funkcji zużywa tam miejsce na zmienne lokalne, parametry i dodatkowe informacje.

Z tego powodu niekiedy konieczna jest rezygnacja z rekurencji. Udowodniono, że każdy program zawierający funkcje rekurencyjne można zapisać bez używania rekurencji.

Zachęcam do wykonania zadań 47 – 52

[70] ang. indirect recursion