# Wgłąb języka C

**>>>Spis treści<<<**

**Wydawnictwo Helion**

## PRZEDMOWA

Niniejsza książka jest pomyślana jako książka dla programistów. Nie stanowi ona całościowego, systematycznego opisu języka C i nie jest książką do nauki programowania. Mam natomiast nadzieję, że będzie ona pożyteczna dla każdego, kto zna i używa języka C. Można na tę książkę spojrzeć na dwa sposoby. Z jednej strony zawiera ona bardzo solidne i dogłębne opisy pewnych zagadnień i może służyć jako podręcznik, do którego się często sięga w pracy (takie są rozdziały o funkcjach formatowanego wejścia-wyjścia i preprocesorze). Z drugiej zaś strony książka ta pełna jest perełek, nietypowych konstrukcji i zastosowań języka. Słowem, nawet zaawansowani programiści znajdą tu coś ciekawego.

Rozdziały są właściwie niezależne od siebie i mogą być czytane w dowolnej kolejności. Tym nie mniej, są one ułożone kolejno, według rosnącego stopnia trudności i czytane po kolei będą najłatwiejsze do zrozumienia. Poza tym, przed przeczytaniem rozdziału o programowaniu współbieżnym, w którym bardzo intensywnie jest używany preprocesor, radzę przeczytać rozdział o preprocesorze. Na deser radzę zostawić ostatni rozdział o niewiele mówiącym tytule "Kod wynikowy". Opisuję w nim między innymi jak kompilować programy w C żeby otrzymać kod o długości rzędu 50 bajtów i jak automatycznie zamienić zwykły program w program rezydentny. Po przeczytaniu tego rozdziału trudno będzie się oprzeć przed spędzeniem nocy przy komputerze.

Gliwice 29.05.1993

A.S.

# I. Programowanie niestrukturalne

Tytuł może trochę dziwić: w książkach o programowaniu można raczej spotkać rozdziały o programowaniu strukturalnym. Wielu programistów i autorów ma do sprawy strukturalności stosunek dogmatyczny, nie dopuszczając żadnych odstępstw. Ja chciałbym tutaj wziąć w obronę "wyklęte niestrukturalności" i przy okazji przedstawić sposób ich realizacji w języku C. Rozdział ten przedstawia zarówno zastosowania konstrukcji niestrukturalnych w "solidnym" programowaniu, jak i pewne "sztuczki". W niektórych przykładach pokażę, że czasami algorytm niestrukturalny może dać bardziej czytelny program. Znajdą się w tym rozdziale także konstrukcje nietypowe, prawdopodobnie mniej czytelne, w których niestrukturalność zastosowano w celu otrzymania bardzo zwięzłego kodu.

Przypomnijmy najpierw, na czym polega programowanie strukturalne. Zakłada ono, że sterowanie nie może być przekazywane przy pomocy żadnych instrukcji skoku z wyjątkiem tych "zaszytych" w instrukcjach sterowania pętli (**for**, **while**). Przyjęcie takiej zasady jest na ogół słuszne i zmusza do bardziej porządnego konstruowania programu (piszę "zmusza", bo w wielu przypadkach zapisanie algorytmu w sposób strukturalny jest trudniejsze). Czasami jednak podporządkowanie się tej zasadzie powoduje zbytnie zagnieżdżenie struktur programu i czyni go mniej czytelnym. W takiej sytuacji należy oczywiście zastosować rozwiązanie dające program bardziej przejrzysty, nawet jeśli jest ono niestrukturalne. Innym powodem zastosowania konstrukcji niestrukturalnej jest chęć skrócenia kodu wynikowego (na przykład w programach rezydentnych). W takich sytuacjach mądrze zaplanowany skok pozwala często uniknąć zbędnego powtarzania jakiejś sekwencji instrukcji.

# 1. Instrukcje break i continue

Wbrew dość powszechnemu mniemaniu programowanie niestrukturalne to nie tylko, i nawet nie przede wszystkim instrukcja **goto**. W języku C najczęściej używane instrukcje niestrukturalne to **break** i **continue**. Można powiedzieć, że są to instrukcje bardziej "cywilizowane" od **goto**. Sprowadzają się one do wykonania skoku, ale nie jest to skok do dowolnie umieszczonej etykiety lecz w konkretne miejsce, związane z pętlą, w obrębie której instrukcje te występują.

Instrukcja **break** powoduje wyjście z pętli, czyli zakończenie jej działania z powodu innego niż niespełnienie warunku w instrukcji pętli. Użycie instrukcji **break** jest możliwe w dowolnym miejscu pętli i może wystąpić dowolną ilość razy. Wyjście z pętli za pomocą instrukcji **break** jest bardzo wygodne, gdy warunek wyjścia ze względu na swoją naturę nie powinien być dołączony do głównego warunku pętli (takim warunkiem jest np. sprawdzanie, czy nie wystąpił błąd). Instrukcja **break** powoduje wyjście tylko z tej pętli, w której się bezpośrednio znajduje i nie umożliwia wyjścia na zewnątrz kilku zagnieżdżonych pętli. Strukturalną alternatywą dla instrukcji **break** jest rozbudowanie warunku pętli. Czasami trzeba użyć specjalnej zmiennej sterującej i dodać do warunku pętli sprawdzanie jej wartości. Taki zapis jest zwykle mniej czytelny, szczególnie gdy warunki pętli są i tak rozbudowane, lub trzeba dodawać zmienną sterującą. Dodatkową wadą jest fakt, że przed wyjściem z pętli **for** (przed sprawdzeniem warunku) zostanie wyliczone wyrażenie modyfikujące (trzecie wyrażenie w instrukcji **for**).

Oto kilka sytuacji, w których instrukcja **break** umożliwia napisanie czytelnego i naturalnego kodu:

1. Czytanie pliku aż do znalezienia danego znaku.

```
while((c=getc(file)!=EOF)
 if(c=='♥')break;
```

W wyniku wykonania tej pętli, wskaźnik pliku zostanie ustawiony w pozycji za pierwszym wystąpieniem znaku '♥'.

Użycie rozwiązania wykorzystującego instrukcję **break** odwzorowuje naturalne sformułowanie problemu: czytamy kolejne znaki aż do końca pliku

```
while((c=getc(file)!=EOF)
```

i dla każdego sprawdzamy, czy nie jest to znak '♥'

```
if(c=='♥')break;
```

Połączenie warunków w podobnych wypadkach daje zwykle wyrażenie złożone, trudniejsze do zrozumienia. Zdaję sobie sprawę, że każdy programista bezbłędnie rozumie najbardziej skomplikowane wyrażenia w swoim programie i irytują go argumenty typu: 'za kilka miesięcy nie będziesz wiedział co Twój program robi' albo

'należy pisać tak, żeby program był zrozumiały także dla innych'. Moim zdaniem jednak istnieje ważniejszy argument za porządnym kodowaniem programów. Jeżeli programista ma wyrobiony nawyk zastanawiania się, jak zakodować wymyślony algorytm, a nie pisania szybko, byle działało (niestety, sam tak często robię), to jest szansa, że zanim zrealizuje swój pierwszy pomysł, wymyśli lepszy algorytm.

2. Czasami jakiś warunek opuszczenia pętli powinien znaleźć się wewnątrz ciała pętli i nie może być umieszczony w jej głównym warunku, który znajduje się na początku albo na końcu. Oczywiście nie oznacza to, że nie da się takich algorytmów zapisać z jednym tylko warunkiem pętli. Aby to jednak zrobić, trzeba często stosować zmienne pomocnicze i dodatkowe instrukcje warunkowe. Sytuację taką ilustrują poniższe przykłady:

**a)**
```
do {
     openwind(38,9,66,11,"WCZYTA_");
     if(edline(name,1,1,25,name))
      {
       closewind();
       break;
      }
     closewind();
    }
  while(read());
```

Jest to uproszczony fragment rzeczywistego programu. Funkcje **openwind** oraz closewind służą do otwierania i zamykania okienka tekstowego. Funkcja **edline** służy do wczytania z klawiatury ciągu znaków (w tym wypadku nazwy pliku). Zwraca ona wartość niezerową, gdy użytkownik w czasie wpisywania ciągu nacisnął klawisz ESC. Funkcja **read** odczytuje z dysku plik o nazwie name. Jeżeli pliku nie uda się otworzyć, wypisuje ona komunikat o błędzie i zwraca wartość niezerową. Oczywiście także tę pętlę dałoby się zapisać w pełni strukturalnie, bez użycia instrukcji **break**, ale kod byłby mniej czytelny.

**b)** Następny fragment pochodzi z rezydentnego programu wyróżniającego słowa kluczowe języka C na ekranie. Poniższa pętla (zamieszczona tu w postaci uproszczonej) ma za zadanie sprawdzić, czy w danym miejscu ekranu (wskazywanym przez zmienną scr) zaczyna się słowo kluczowe C. Jeżeli takie słowo zostanie znalezione, jest ono wyróżniane przez zmianę atrybutu. Przed wejściem do pętli zmienna list wskazuje tablicę słów kluczowych C.

```
char *listtab[]={"for","int","if","char","while","case","do",
                 "else","return","void","default","struct",
                 "switch","far","extern","break","goto",
                 "float","const","continue","include","define",
                 "unsigned","static","sizeof",0},**list;
/* ... */
```

```
list=listtab;
while(*list)            /* dla wszystkich słów kluczowych */
 {
   i=0;
   while(scr[2*i]==(*list)[i])i++;      /* porównaj słowa */
   if(!(*list)[i])                      /* jeżeli takie same */
      {
       scr++;
       while(i)scr[2*(--i)]|=0xF;             /* rozjaśnij */
       scr--;
       break;           /* już znalezione, wyjdź z pętli */
      }
   list++;                              /* następne słowo */
 }
```

Instrukcja **break** służy do wyjścia z pętli w przypadku znalezienia w liście słów kluczowych słowa, które zaczyna się we wskazanym miejscu ekranu. Podobnie jak w poprzednim przypadku można zapisać ten algorytm strukturalnie, np. przy użyciu pętli **do**{ ... }**while** z jednym, rozbudowanym warunkiem, ale rozwiązanie takie byłoby nieczytelne.

Instrukcja **break** jest także używana do wyjścia z instrukcji wyboru **switch**. To zastosowanie zostanie opisane podczas omawiania instrukcji **switch** w dalszej części tego rozdziału.

Instrukcja **continue** dotyczy tylko pętli. Jej wywołanie powoduje pominięcie dalszej części ciała pętli i wykonanie następnej iteracji. Jeżeli ciałem pętli jest instrukcja blokowa (zresztą inna konstrukcja, typu `while(x)continue;, jest bez sensu`), to instrukcja **continue** jest równoważna skokowi do etykiety umieszczonej przed nawiasem } zamykającym instrukcję blokową.

Na przykład:

```
do{
   if((c=getch())<'A')continue;
   putchar(c);
   x++;
  }
while(x<10);
```

jest równoważne konstrukcji

```
do{
   if((c=getch())<'A')goto end;
   putchar(c);
   x++;
   end:
  }
while(x<10);
```

Instrukcja **continue** nie jest w tak oczywisty sposób lepsza od swojej strukturalnej alternatywy jak instrukcja **break**. Powyższy przykład lepiej byłoby zapisać strukturalnie:

```
do{
   if((c=getch())>='A')
    {
     putchar(c);
       x++;
    }
  }
while(x<10);
```

Ciało tej malutkiej pętli mieści się na jednym ekranie i na pierwszy rzut oka widać, że instrukcja złożona po instrukcji **if** obejmuje całe ciało pętli.

Instrukcja **continue** będzie jednak lepsza gdy część ciała pętli, która ma być ominięta, jest długa i zagnieżdżona (dodanie kolejnej pary nawiasów klamrowych jeszcze bardziej by ją zagmatwało). Jeżeli pominięcie dalszej części ciała pętli może nastąpić w kilku miejscach i pod różnymi warunkami, należy również stosować instrukcje **continue** zamiast zagnieżdżać kolejne warunki.

## 2. Instrukcja goto

Tu właściwie wszystko jest dozwolone. Można wykonać skok do dowolnego miejsca (oznaczonego etykietą) w obrębie funkcji. Można wskoczyć do pętli i można z niej wyskoczyć. Ta wielka dowolność użycia instrukcji **goto** może powodować przy niefrasobliwym jej używaniu, straszliwe zagmatwanie programu. Nie jest to jednak powód, żeby całkowicie z **goto** rezygnować. Oto sytuacje, w których **goto** wydaje się być jak najbardziej na miejscu.

1. Wyjście z zagnieżdżonych pętli:

```
for(...;...;...)
  for(...;...;...)
    {
      ...
      if(warunek) goto end;
    }
  end: ;
```

Ponieważ instrukcja **break** powoduje wyjście tylko z najbardziej wewnętrznej pętli, trzeba zastosować bezpośredni skok poza wszystkie pętle. Oczywiście, możliwe jest napisanie jakiegoś strukturalnego odpowiednika, ale byłby on koszmarnie nieczytelny (tym bardziej, im więcej pętli). Jako ćwiczenie proponuję strukturalne napisanie wyjścia z kilku zagnieżdżonych pętli **for** z jakimiś konkretnymi warunkami.

2. Organizacja dużych pętli bez warunku zakończenia pętli (nieskończonych). Zamiast pisać:

```
while(1)
 {
  /*...*/
```

i potem widząc kilka ekranów dalej

```
    }
  }
 }
}
```

zastanawiać się, która klamra zamyka pętlę, lepiej napisać:

```
start:
/*...*/
goto start;
```

Język C posiada jeszcze jedną wspaniałą cechę, którą też należy zaliczyć do konstrukcji niestrukturalnych. Myślę tu o możliwości umieszczania instrukcji **return** w kilku różnych miejscach funkcji, z różnymi wartościami do zwrócenia. Wydaje mi się, że ustrukturalnianie funkcji, która w naturalny sposób zwraca wartości w różnych miejscach, tak żeby wyjście znajdowało się tylko w jednym miejscu (na końcu), nigdy nie jest sensowne. Sprowadza się ono do zwracania wartości jakiejś zmiennej pomocniczej, której wartość jest ustawiana w różnych miejscach funkcji. Nigdy nie można wtedy mieć pewności, czy ustawiona wartość nie zostanie z powodu błędu zmieniona zanim sterowanie dotrze do instrukcji **return** na końcu funkcji.

## 3. Instrukcja switch

Jako ostatnią chcę omówić dokładniej królową niestrukturalności w języku C, instrukcję **switch**. Wokół tej potężnej w swoich możliwościach instrukcji panuje wiele niezrozumienia. Pokutują tutaj chyba przyzwyczajenia wyniesione przez wielu programistów z innych języków programowania (szczególnie Pascala).

Składnia instrukcji **switch** jest następująca:

```
switch(wyrażenie)
  {
    case stała1 : /* ciąg_instrukcji1 */
    case stała2 : /* ciąg_instrukcji2 */
    /* default nie musi być na końcu  */
    default     : /* ciąg_instrukcji0 */
    /* ... */
    case stałax : /* ciąg_instrukcjix */
  }
```

gdzie wyrażenie jest dowolnym wyrażeniem całkowitym. Niektóre implementacje dopuszczają także wyrażenia innych typów, których wartość może być poddana automatycznej konwersji do typu całkowitego. W szczególności, w kompilatorze Turbo C następujący fragment kodu:

```
float f;
f=fun();
switch(f)    /*  w MS C   switch((int)f)   */
```

```
{
  case 0  : printf("0≤f<1"); break;
  case 1  : printf("1≤f<2)"); break;
  default : printf("f≥2 lub f<0");break;
}
```

spowoduje wypisanie, do jakiego przedziału należy wartość zmiennej f.

Stałe we frazach **case** muszą być stałymi całkowitymi lub wyrażeniami stałymi (tzn. takimi, których wartość może być wyliczona w czasie kompilacji) całkowitymi. Stała całkowita oznacza liczbę typu **int**[1)] lub znak.

Fraza **default** jest opcjonalna i może być umieszczona w dowolnym miejscu w obrębie instrukcji **switch**.

Dla dobrego zrozumienia działania instrukcji **switch** trzeba przyjąć do wiadomości, że frazy **case** zachowują się jak etykiety skoków. Wykonanie instrukcji **switch** jest równoważne wartościowaniu wyrażenia i wykonaniu skoku do niejawnej etykiety związanej z tą frazą **case**, dla której wartość stałej jest równa wartości wyrażenia, lub do niejawnej etykiety związanej z frazą **default** (jeśli występuje), jeżeli w żadnej z fraz **case** nie ma odpowiedniej wartości. Po wykonaniu takiego niejawnego skoku wszystkie frazy **case** i **default** są ignorowane (podobnie jak ignorowane są normalne etykiety) i program jest wykonywany tak jakby ich nie było. Ponieważ frazy **case** można uważać za etykiety mogą one być umieszczane tam, gdzie można umieszczać etykiety (czyli w dowolnym miejscu między instrukcjami). Jeżeli w obrębie ciała instrukcji **switch** zostanie napotkana instrukcja **break**, nie zagnieżdżona w żadnej pętli, to spowoduje ona wyjście z instrukcji **switch**.

Po tym, trochę może skomplikowanym, opisie kolej teraz na kilka przykładów, które wyjaśnią, mam nadzieję, wszelkie wątpliwości.

Pierwszy przykład ilustruje, jak po skoku do jednej z fraz **case** wykonywane są kolejne instrukcje, niezależnie od tego, czy występują jakieś inne frazy **case** lub fraza **default**.

```
switch(getch())
 {
    case '\r'   : read();
    case '\x1b' : close_file();
                  return;
 }
```

[1] We frazie **case** można umieścić liczbę typu **long**, ale zostanie ona w trakcie kompilacji poddana konwersji do typu **int** (przez odrzucenie dwóch najstarszych bajtów).

Po naciśnięciu klawisza ESC nastąpi wywołanie funkcji **close_file** i wyjście z funkcji, natomiast po naciśnięciu klawisza ENTER nastąpi wywołanie funkcji **read**, potem funkcji **close_file** a potem wyjście z funkcji. Inteligentne wykorzystanie faktu, że instrukcje nie są przypisane do poszczególnych przypadków, a same frazy **case** są faktycznie tylko etykietami, pozwala uniknąć wielokrotnego przepisywania kodu, który ma być wykonany dla więcej niż jednego przypadku.

Jeżeli obsługa przypadków ma być rozdzielona, należy za ostatnią instrukcją obsługi przypadku umieścić instrukcję **break**, tak jak w poniższym przykładzie:

```
switch(menu())
 {
   case 1 : list(); break;
   case 2 : print(); break;
   case 3 : return;
 }
```

Obsługa menu to klasyczny przykład, kiedy każdy przypadek jest obsługiwany osobno: funkcja **menu** zwraca liczbę od 1 do 3 w zależności od tego, którą opcję użytkownik wybrał z menu. Dla każdej opcji istnieje odpowiednia akcja (niezależna od innych).

Jak już wcześniej napisałem, frazy **case**, tak jak etykiety, mogą być umieszczane w dość dowolnych miejscach. Oto przykłady:

**1.**
```
switch(getch())
  {
    case '\x1b' : printf("Zapisać? ");
                  if(getch()=='t')
    case '\r'   :   save();
                  return;
  }
```

W tym przykładzie druga fraza **case** jest umieszczona wewnątrz instrukcji **if**, czego być może na pierwszy rzut oka nie widać. Wywołanie funkcji **save** nastąpi, gdy użytkownik naciśnie klawisz ENTER (oznaczający tu wyjście z zapisem) lub klawisz ESC (wyjście), i na pytanie "Zapisać? " odpowie twierdząco.

**2.** Frazy **case** można także umieścić wewnątrz pętli:

```
switch(x)
 {
  default  : for(x=4;x;x--)
               {
  case  3  :
  case  2  :
  case  1  :         /* ciało pętli */
               }
  case  0  :
  case -1  :
 }
```

Jeżeli zmienna x miała przed instrukcją **switch** wartość 1, 2 lub 3, nastąpi wejście do pętli **for** "od środka". Proszę zauważyć, że w takim wypadku nigdy nie wykona się wyrażenie inicjujące (w tym wypadku x=4). W efekcie, pętla w powyższym przykładzie wykona się x razy lub cztery razy jeżeli x jest większe niż 4. Pętla nie wykona się ani razu dla x równego -1 lub 0. Oczywiście nic nie stoi na przeszkodzie, aby frazy **case** były umieszczone w różnych miejscach ciała pętli, a nie na początku, jak w naszym przykładzie.

W podobny sposób można umieścić frazy **case** w pętli **while** lub **do**. W takim wypadku efektem może być wykonanie tylko fragmentu ciała pętli w pierwszym obiegu (tzn. rozpoczęcie jej "od środka").

Na zakończenie tych teoretycznych rozważań o niestrukturalności chciałbym przedstawić jeden przykład "z życia wzięty". Ciekawego zastosowania algorytmu niestrukturalnego zdarzyło mi się użyć w programie rezydentnym (gdzie liczy się każdy bajt). Trzeba było rozwiązać następujący problem: na ekranie znajduje się niewielka ramka, a użytkownik ma mieć możliwość przesuwania jej przy pomocy klawiszy kursora. Należy mieć na uwadze, że program ma obsługiwać także wiele innych klawiszy. Do dyspozycji mamy:

- funkcję **gettext** (char *mem), zapamiętującą w buforze mem zawartość pamięci ekranu o rozmiarze i kształcie ramki. Położenie ramki jest określane przez globalną zmienną x, która określa przesunięcie lewego górnego wierzchołka ramki względem początku pamięci ekranu,
- funkcję **puttext** (char *mem) działającą odwrotnie do gettext,
- zmienną XX zawierającą przemieszczenie lewego górnego wierzchołka ramki aktualnie wyświetlonej na ekranie względem początku pamięci ekranu,
- dwa bufory mem1 i mem2 przy czym mem1 zawiera zawartość ekranu "spod" aktualnie wyświetlonej ramki.

Algorytm przesunięcia ramki będzie następujący:

1. sprawdź, czy ramkę można przesunąć (czy nie znajduje się na krawędzi ekranu). Jeśli nie można, idź do (7)
2. zapamiętaj zawartość ramki w mem2
3. odtwórz zawartość ekranu z mem1
4. zmień wartość x tak, żeby wskazywała na nowe położenie ramki
5. zapamiętaj zawartość ekranu w mem1
6. odtwórz zawartość ramki z mem2
7. koniec

Kod realizujący ten algorytm dla odpowiednich klawiszy mógłby wyglądać następująco:

```
          switch(getch())
           {
            case UP  : if(XX>=160)                          /*
   w górę */
                       {x=XX; gettext(mem2); puttext(mem1);
                        x=XX-=160;
                        gettext(mem1); puttext(mem2);
                       } break;
            case DN  : if(XX<3520)                           /*
   w dół */
                       {x=XX; gettext(mem2); puttext(mem1);
                        x=XX+=160;
                        gettext(mem1); puttext(mem2);
                       } break;
            case LF  : if((XX%160)>=0)                      /*
   w lewo */
                       {x=XX; gettext(mem2); puttext(mem1);
                        x=XX-=2;
                        gettext(mem1); puttext(mem2);
                       } break;
            case RT  : if((XX%160)<132)                    /*
   w prawo */
                       {x=XX; gettext(mem2); puttext(mem1);
                        x=XX+=2;
                        gettext(mem1); puttext(mem2);
                       } break;
            case ESC : /* kolejne frazy case obsługujące */
                       /* inne klawisze */
           }
```

Każdy od razu zauważy, jak olbrzymi fragment kodu jest tu cztery razy powtarzany. Ruchy w różnych kierunkach różnią się tylko sprawdzanym warunkiem i wartością dodawaną do zmiennych x i XX. Spróbujmy usunąć powtarzający się kod i umieścić go tylko w jednym miejscu, za ostatnią frazą **case** (nie można go umieścić poza instrukcją **switch**, bo obsługuje ona także inne klawisze). Kolejna robocza wersja będzie wyglądała następująco:

```
          switch(getch()) /* wersja robocza, nie działająca
   poprawnie */
           {
            case UP : if(XX>=160)dx=-160;
            case RT : if((XX%160)<132)dx=2;
            case DN : if(XX<3520)dx=160;
            case LF : if((XX%160)>0)dx=-2;
                      x=XX;     gettext(mem2); puttext(mem1);
                      x=XX+=dx; gettext(mem1); puttext(mem2);
                      dx=0;
                      break;
            case ESC: /* kolejne frazy case  */
           }
```

Problem nasz jest następujący: chcemy, żeby sterowanie z każdego przypadku **case** doszło w końcu do wywołań funkcji **gettext** i puttext przesuwających ramkę, a jednocześnie nie chcemy, żeby były wykonywane wszystkie napotkane po drodze instrukcje modyfikujące zmienną dx. Na pierwszy rzut oka wymagania wydają się całkowicie sprzeczne. Może pomogą pytania pomocnicze...

Co zrobić, żeby po wykonaniu jednego przypisania zmiennej dx pozostałe były ignorowane?

A kiedy następuje przypisanie zmiennej dx?

Gdy spełniony zostanie jeden z warunków.

A więc co zrobić, żeby kolejne instrukcje były ignorowane gdy spełniony zostanie jeden z warunków?

Oczywiście trzeba uzupełnić go o frazę **else**.

Oto pełne rozwiązanie:

```
switch(getch())
 {
  case UP : if(XX>=160)dx=-160; else
  case RT : if((XX%160)<132)dx=2; else
  case DN : if(XX<3520)dx=160; else
  case LF : if((XX%160)>0)dx=-2;
            x=XX;     gettext(mem2); puttext(mem1);
            x=XX+=dx; gettext(mem1); puttext(mem2);
            dx=0;
            break;
  case ESC: /* kolejne frazy case */
 }
```

Takie rozwiązanie nie jest dokładnym odpowiednikiem wersji wyjściowej. Jeżeli ramka znajduje się np. przy dolnej krawędzi ekranu, a użytkownik naciśnie klawisz ↓, to ramka będzie "ślizgać się" w lewo wzdłuż krawędzi. Takie działanie uboczne nie jest jednak szkodliwe (a nawet może być uznane za efekt zamierzony).

Można oczywiście modyfikację wersji wyjściowej poprowadzić w innym kierunku i umieścić kod przesuwający ramkę w osobnej funkcji, która byłaby wywoływana z odpowiednim argumentem określającym kierunek przesunięcia. Kod wynikowy byłby niewiele dłuższy.

## 4. Optymalizacje

Dotychczasowe rozważania o niestrukturalności dotyczyły składni języka C. Chciałbym teraz napisać kilka słów na temat kompilatora Turbo C firmy Borland.

Kompilator ten posiada m.in. opcję *jump optimization*. Działanie tej optymalizacji polega na umieszczeniu przez kompilator niejawnych skoków w celu uniknięcia powtarzania jakiegoś fragmentu kodu. Tak więc kompilator tłumaczy strukturalny algorytm tak, jakby był on zapisany w bardziej zwartej, ale niestrukturalnej postaci.

Przeanalizujmy poniższy fragment kodu, wczytujący kolejne rekordy z pliku i zapamiętujący początek każdego z nich. Długość pliku i rozmiar rekordów nie są znane.

```
fgetpos(file,&pos);
read_record(&rec);
while(!feof(file))
 {
  tab[x].pos=pos;
  strcpy(tab[x++].dat,rec.dat);
  fgetpos(file,&pos);
  read_record(&rec);
 }
```

Jeżeli plik jest pusty, pętla nie powinna być wykonana ani razu. Dlatego też należy użyć pętli **while** sprawdzającej warunek !feof(file) przed wykonaniem pętli. Funkcja **feof** zwraca wartość różną od zera dopiero wtedy, gdy zostanie dokonana próba czytania poza końcem pliku (a nie wtedy, gdy w wyniku przeczytania ostatnich znaków wskaźnik plikowy znalazł się w pozycji końcowej). Ta ostatnia cecha funkcji **feof** zmusza nas do umieszczenia dodatkowego wywołania funkcji **read_record**, a co za tym idzie - funkcji **fgetpos**, przed całą pętlą.

Zamiast powtarzania tego fragmentu można by wykonać przed pętlą odpowiedni skok. Powyższy fragment wyglądałby wtedy następująco:

```
goto label;
while(!feof(file))
 {
  tab[x].pos=pos;
  strcpy(tab[x++].dat,rec.dat);
label:
  fgetpos(file,&pos);
  read_record(&rec);
 }
```

Okazuje się jednak, że zastosowanie opcji *jump optimization* ma dokładnie taki sam skutek jak jawne dopisanie skoku. Niejawny skok dodany przez kompilator można zauważyć śledząc krokowo pierwszy fragment kodu skompilowany z ustawioną opcją *jump optimization*. Debugger wyraźnie pokaże, że dla dwóch wierszy przed pętlą **while** w ogóle nie jest generowany kod wynikowy; zamiast tego wykonywany jest skok w odpowiednie miejsce pętli **while**.

Dzięki zastosowaniu optymalizacji wykonywanej przez kompilator możemy mieć więc jednocześnie przejrzysty, strukturalny kod źródłowy i efektywny kod wynikowy odpowiadający konstrukcji z bezpośrednim skokiem.

Mam nadzieję, że przykład ten skłoni Czytelnika do zbadania opcji optymalizacji, które wykonuje używany przez niego kompilator. Jednocześnie należy pamiętać o wyłączeniu optymalizacji w fazie uruchamiania programu, kiedy używa się debuggera i kiedy pożądane jest by kod wynikowy dokładnie odpowiadał kodowi źródłowemu.

# II. Preprocesor

Preprocesor języka C jest narzędziem tyleż potężnym co niedocenianym. Co więcej, mimo swojej prostoty i wewnętrznej spójności jest także często nie rozumiany. Funkcje i sposób działania poszczególnych dyrektyw są powszechnie znane, ale nie jest rozumiana filozofia, duch preprocesora. Postaram się by ten rozdział wyjaśnił ewentualne niejasności. Znajdą się w nim także przykłady zastosowań preprocesora odbiegające nieco od klasycznego definiowania "stałych".

Preprocesor jest ściśle związany z językiem C; jest on wręcz zawarty w stylu programowania w C. Tak samo jak składnia języka, preprocesor jest przedmiotem definicji języka i nie jest w żaden sposób zależny od implementacji. Nie należy preprocesora porównywać do dyrektyw sterujących kompilacją, występujących np. w kompilatorach Pascala.

Czym właściwie jest preprocesor i jak działa? Kiedyś preprocesor był osobnym programem, teraz zwykle jest zintegrowany z kompilatorem. Zasada działania pozostała jednak ta sama: preprocesor przetwarza tekst źródłowy programu. Podkreślam tu słowo *tekst*, ponieważ czasami dyrektywom preprocesora przypisywane jest wręcz magiczne działanie (np. dyrektywie **#include** przypisywana jest rola przy konsolidacji programu). Preprocesor manipuluje napisami nie mając pojęcia co one oznaczają, nie ma także bezpośredniego wpływu na przebieg kompilacji. Częste niezrozumienie sposobu pracy preprocesora wynika prawdopodobnie z tego, że jest on w nowoczesnych kompilatorach niewidoczny. Programista nie widzi efektu jego działania, podczas gdy rzeczywiście kompilowany jest właśnie tekst, który pojawia się na wyjściu preprocesora.

## 1. Dyrektywy preprocesora

Dyrektywy preprocesora muszą być umieszczone w osobnym wierszu, zaczynającym się znakiem # (poprzedzonym co najwyżej odstępami).

### 1.1. Dyrektywa #pragma

Dyrektywa **#pragma** została wprowadzona w standardzie ANSI aby umożliwić rozszerzenia w poszczególnych implementacjach. Pomiędzy implementacjami gwarantuje się jedynie, że nieznane dyrektywy **#pragma** zostaną zignorowane.

### 1.2. Dyrektywa #include

Dyrektywa **#include** służy do włączania zawartości pliku do tekstu źródłowego. Używa się jej głównie do włączania tzw. plików nagłówkowych. Pliki nagłówkowe zawierają zwykle deklaracje obiektów zewnętrznych: funkcji i zmiennych, definicje typów oraz definicje identyfikatorów wykorzystujące dy-

rektywę **#define**. Nie należy do dobrego stylu programowania włącznie przy pomocy dyrektywy **#include** kodu źródłowego zawierającego definicje czy to

zmiennych czy funkcji. Definicje obiektów zadeklarowanych w pliku nagłówkowym znajdują się zwykle w osobnych, niezależnie kompilowanych plikach; są one często - w postaci skompilowanej - przechowywane w bibliotekach. Przyjmuje się, że pliki nagłówkowe mają rozszerzenie ".h" .

Wykorzystanie możliwości włączania do pliku źródłowego zawartości innego pliku jest tylko kwestią stylu programowania i nic nie stoi na przeszkodzie, żeby przy pomocy dyrektywy **#include** włączać do tekstu źródłowego dowolny fragment, który z jakichś powodów lepiej jest przechowywać w osobnym pliku. Może to być na przykład jakiś dłuższy tekst, który ma być dołączony do programu w postaci zaszyfrowanej. W takim przypadku najlepiej umieścić ten tekst w osobnym pliku, zaszyfrowanym przy pomocy odpowiedniego programu, wersję zaszyfrowaną wpisać do innego pliku (np. o nazwie *haslo*) i włączyć w odpowiednie miejsce pliku źródłowego dyrektywą **#include**:

```
        char *passwd=   /* zakładamy, że wiersze pliku haslo
uję te */
                        /* są w cudzysłowy i nie zawierają
żadnych */                          /* znaków sterujących np. \r
, \t          */
        #include "haslo"
        ;
```

W razie konieczności zmiany wartości przypisywanej zmiennej passwd wystarczy zmienić odpowiedni plik i zaszyfrować go.

Nazwa pliku włączanego dyrektywą **#include** może być oczywiście dowolna (pliki *.h są wyróżniane tylko przez ludzi, preprocesorowi jest wszystko jedno). Jako ciekawostkę podaję, że plikiem takim może być także konsola operatorska!

```
        #include "con"
```

Powyższa dyrektywa spowoduje włączenie do kompilowanego pliku tekstu wpisanego przez operatora i zakończonego znakiem końca pliku ^Z. Tekst ten będzie można wpisać w czasie kompilacji.

Nazwa pliku w dyrektywie **#include** może być umieszczona albo w cudzysłowach:

```
        "nazwa"
```

albo w nawiasach kątowych:

```
        <nazwa>
```

W pierwszym przypadku plik do włączenia jest poszukiwany w katalogu bieżącym. Jeżeli nazwa pliku umieszczona jest w nawiasach kątowych, to plik jest po-

szukiwany w katalogu, określonym przy pomocy opcji kompilatora jako katalog zawierający pliki nagłówkowe. Drugiej możliwości używa się głównie do włączania plików nagłówkowych bibliotek standardowych.

Przy okazji pisania o dyrektywie **#include** i o włączaniu plików nagłówkowych chciałbym krótko wspomnieć o deklaracjach funkcji, które jako takie nie mają nic wspólnego z preprocesorem, stanowią natomiast główną zawartość plików nagłówkowych.

Deklaracja funkcji jest napisem określającym typ funkcji oraz typy argumentów funkcji. Typy podane w deklaracji muszą być takie same jak określono w definicji funkcji. Jeżeli w jakimś pliku źródłowym umieszczone jest wywołanie funkcji skompilowanej w innym pliku (np. funkcji bibliotecznej), kompilator bez deklaracji tej funkcji nie jest w stanie odgadnąć jakie są typy argumentów i jakiego typu wartość zwraca funkcji. Informacje te mają oczywiście dla kompilatora kapitalne znaczenie. Od typu argumentu zależy liczba bajtów, które odczyta ze stosu funkcja czytając ten argument. Taką samą liczbę bajtów należy na stos zapisać. Od typu funkcji zależy sposób zwracania wyniku. Deklaracje funkcji służą właśnie do przekazania kompilatorowi tych informacji. Jeżeli nie zna on typów argumentów i wyniku jakiejś funkcji (nie ma jej deklaracji) wówczas przyjmuje, że wszystkie argumenty i wynik są typu **int**. Deklaracje funkcji wpływają tylko na postać generowanego kodu (położenie na stosie dwóch czy pięciu bajtów), nie powodują natomiast wygenerowania dodatkowego kodu. Z powyższego wynika, że nie mają one wpływu na to jakie moduły zostaną dołączone z bibliotek w procesie konsolidacji. Dołączenie kodu funkcji z biblioteki następuje wtedy, gdy jej identyfikator jest używany (np. do wywołania funkcji) w danym programie, a nie wtedy, gdy jest zadeklarowana.

## 1.3. Dyrektywy #if, #ifdef, #ifndef, #else, #endif

Dyrektywy te służą do warunkowego przekazania fragmentu tekstu do kompilacji. Po dyrektywie **#if** wymagane jest wyrażenie stałe. Jeżeli w wyrażeniu tym występują identyfikatory zdefiniowane wcześniej przy pomocy dyrektywy **#define** to są one zastępowane odpowiednimi napisami (patrz opis dyrektywa **#define** w dalszej części rozdziału). Wyrażenia nie mogą być zakończone średnikami. Po dyrektywach **#else** i **#endif** nie mogą występować żadne napisy (jeżeli występują, są ignorowane).

Ogólny schemat użycia dyrektyw włączania warunkowego jest następujący:

```
#if wyrażenie
/* tekst1 */
#else
/* tekst2 */
#endif
```

Dyrektywa **#else** jest w tej konstrukcji opcjonalna, natomiast dyrektywa **#endif** jest obowiązkowa i każdej dyrektywie **#if** musi odpowiadać dokładnie jedna dyrektywa **#endif**.

Jeżeli *wyrażenie* ma wartość różną od zera, wiersze tekstu oznaczone na powyższym schemacie jako *tekst1*, są włączane do dalszego przetwarzania, natomiast wiersze oznaczone jako *tekst2*, nie są włączane. Jeżeli *wyrażenie* ma wartość zero do dalszego przetwarzania włączane są wiersze oznaczone jako *tekst2* (jeżeli występują, tzn. jeżeli występuje odpowiednia dyrektywa **#else**). Fragment tekstu po włączeniu, jest dalej przetwarzany przez preprocesor. Z powyższego wynika, że konstrukcje dyrektyw włączania warunkowego mogą być zagnieżdżone. Konstrukcja z użyciem dyrektyw **#ifdef** i **#ifndef** jest taka sama jak ta wykorzystująca dyrektywę **#if**. Taką samą rolę pełnią też dyrektywy **#else** i **#endif**. Preprocesor opracowując dyrektywę **#ifdef** lub **#ifndef** sprawdza, czy pierwsza występująca po niej w wierszu jednostka leksykalna (dalsza część wiersza jest ignorowana) jest identyfikatorem zdefiniowanym przy pomocy dyrektywy **#define**. W przypadku dyrektywy **#ifdef** spełnienie powyższego warunku powoduje akcję odpowiadającą warunkowi o wartości niezerowej w dyrektywie **#if,** a niespełnienie, akcję odpowiadającą warunkowi o wartości zero. Dyrektywa **#ifndef** jest interpretowana odwrotnie.

Typowym zastosowaniem dyrektyw włączania warunkowego jest pisanie programu w taki sposób by można go było łatwo skompilować w kilku wersjach (np. wersja pełna i wersja demonstracyjna). Żeby to osiągnąć, fragmenty, które mają znaleźć się tylko w wersji demonstracyjnej umieszcza się w konstrukcji typu

```
#ifdef DEMO
/* kod tylko dla wersji demonstracyjnej */
#endif
```

a fragmenty, które mają się znaleźć tylko w wersji normalnej, w konstrukcji typu

```
#ifndef DEMO
/* kod tylko dla wersji normalnej */
#endif
```

Jak widać do sterowania włączaniem tekstu do kompilacji służy identyfikator DEMO. Jeżeli chcemy otrzymać program w wersji demonstracyjnej należy na początku kody źródłowego umieścić dyrektywę

```
#define DEMO
```

Jeżeli natomiast chcemy otrzymać program w wersji normalnej, należy tę dyrektywę usunąć lub ująć w znaki komentarza.

Innym powszechnym zastosowaniem dyrektyw włączania warunkowego jest uniemożliwianie wielokrotnego przetwarzania tego samego kodu źródłowego. Sytuacja taka może wystąpić w przypadku kilkukrotnego włączenia dyrektywą

**#include** pliku nagłówkowego. Żeby się przed tym zabezpieczyć, plik nagłówkowy można skonstruować w następujący sposób:

```
/* początek pliku nagłówkowego okienka.h */
#ifndef _OKIENKA
#define _OKIENKA

/* treść pliku */

#endif
/* koniec pliku okienka.h */
```

Dzięki takiej konstrukcji przy pierwszym włączeniu takiego pliku zostanie zdefiniowany identyfikator _OKIENKA, a przy następnych włączeniach powtórne przetwarzanie zawartości tego pliku zostanie uniemożliwione dyrektywą `#ifndef _OKIENKA`. Nazwę identyfikatora sterującego (w tym przypadku _OKIENKA) należy tak dobrać, żeby zminimalizować ryzyko zdefiniowania go przypadkowo w innym celu. Ogólnie przyjęte jest, że wszystkie niejawne identyfikatory globalne (także nazwy niejawnych zmiennych i funkcji) rozpoczynają się jednym lub dwoma znakami podkreślenia, natomiast identyfikatorów jawnych nie rozpoczyna się tym znakiem.

## 1.4. Dyrektywy #define i #undef

Dyrektywa **#define** służy do definiowania identyfikatorów i przypisywania im ciągów znaków. Składnia dyrektywy **#define** jest następująca:

```
#define ident napis
```

lub

```
#define ident(par1,par2,...,parn) napis
```

W pierwszej postaci dyrektywa **#define** przypisuje identyfikatorowi *ident* ciąg znaków *napis*. Jeżeli preprocesor przetwarzając plik źródłowy natrafi na jednostkę leksykalną *ident*, zastąpi ją ciągiem znaków *napis*. Zastąpienie nie nastąpi, jeżeli jednostka znajduje się w wierszu zawierającym dyrektywę **#define**, **#undef**, **#ifdef**, **#ifndef**, lub wewnątrz literału znakowego (pomiędzy parą cudzysłowów). Tekst powstały po zastąpieniu identyfikatora przypisanym mu napisem jest dalej przetwarzany przez preprocesor. Wynika z tego, że w nim ewentualnie znalezione inne identyfikatory mogą także zostać zastąpione odpowiednimi napisami. Ciąg znaków *napis* może zawierać dowolne znaki (znak # ma specjalne znaczenie, opisane poniżej). Ciąg ten jest ograniczony końcem wiersza. W celu uzyskania czytelniejszego zapisu można zakończyć wiersz znakiem "\", który oznacza, że następny wiersz jest kontynuacją aktualnego, na przykład:

```
/* drukuj wartość sformatowaną na ekranie i drukarce */
        #define print(format,wart)    printf(format,wart);
\

fprintf(stdprn,format,wart);
```

W drugiej z uprzednio przytoczonych postaci dyrektywa **#define** powoduje utworzenie identyfikatora z *parametrami*. Jeżeli preprocesor napotka taki identyfikator, po którym będzie występować odpowiednia liczba argumentów w nawiasach (), to zastąpi znaleziony identyfikator odpowiednim łańcuchem. W czasie zastąpienia każde wystąpienie paramertu formalnego w ciągu przypisanym identyfikatorowi jest zastępowane odpowiednim argumentem aktualnym. Czasami wymóg zgodności liczby argumentów nie jest przestrzegany i nieodpowiednia ilość argumentów powoduje jedynie wygenerowanie ostrzeżenia. Argumenty powinny być rozdzielone przecinkami[2]. Argumenty mogą być dowolnymi napisami (także pustymi).

Jeżeli w napisie przypisanym identyfikatorowi występuje ciąg ## (para znaków #) to po rozwinięciu identyfikatora, ciąg ten jest usuwany wraz z otaczającymi go odstępami. Ta cecha preprocesora jest używana do tworzenia identyfikatorów (patrz przykład 4 pod koniec rozdziału).

Jeżeli natomiast w definicji identyfikatora z parametrami nazwa parametru jest poprzedzona znakiem # to przy zastępowaniu odpowiedni argument zostanie umieszczony w cudzysłowie (patrz j.w.).

Przypisania dokonane za pomocą dyrektyw **#define** obowiązują od wiersza, w którym znajduje się dyrektywa, do końca pliku. Nigdy nie obowiązują one w innych plikach[3]. Przypisania takie mogą być anulowane w dowolnym miejscu za pomocą dyrektywy:

```
#undef ident.
```

Oto kilka przykładów ilustrujących najważniejsze cechy omawianych dyrektyw:

1. W ciągu przypisanym identyfikatorowi można użyć innego, wcześniej zdefiniowanego identyfikatora.

```
#define max(x,y) ((x)>(y)?(x):(y))
#define max3(x,y,z) (max(max(x,y),z))
```

Jeżeli w tekście źródłowym wystąpi napis

```
max3(a,b,c);
```

to zostanie on zastąpiony napisem

```
((((a)>(b)?(a):(b)))>(c)?(((a)>(b)?(a):(b))):(c));
```

---

[2] Przecinki umieszczone pomiędzy znakami cudzysłowu, apostrofami lub nawiasami nie oddzielają argumentów

[3] Chodzi o pliki osobno kompilowane. Jeżeli plik jest włączony do innego dyrektywą **#include** to nie jest to już inny plik.

Przykład ten pokazuje, że preprocesor po zastąpieniu jakiegoś identyfikatora odpowiednim napisem, dalej przetwarza ten napis i dokonuje ewentualnych dalszych zastąpień. W tym przypadku, w tekście, którym został zastąpiony identyfikator max3, dwukrotnie występował identyfikator max, który został z kolei zastąpiony przypisanym mu tekstem

```
((x>y)?(x):(y))
```

2. Nazwy identyfikatorów preprocesora trzeba tak dobierać żeby nie kolidowały z identyfikatorami funkcji i zmiennych. Dobrą zasadą jest używanie w identyfikatorach preprocesora tylko dużych liter.

```
fun(int x)
{
/* ... */
}

#define fun(a,b) a+b

main()
{
 fun(10);             /* zła ilość argumentów */
}
```

Powyższego programu nie uda się skompilować, ponieważ preprocesor stwierdzi, że identyfikator *fun* posiada niewłaściwą liczbę argumentów.

Jeszcze bardziej niebezpieczna sytuacja powstanie, gdy zarówno funkcja **fun** jak i identyfikator zdefiniowany przy pomocy dyrektywy **#define** będą miały tę samą liczbę parametrów. Program wtedy zostanie skompilowany bezbłędnie, lecz w "tajemniczy" sposób nie będzie wywoływał funkcji **fun**. W powyższym przykładzie definicja funkcji została umieszczona przed dyrektywą **#define**. W przypadku kolejności odwrotnej "nie grozi" nam skompilowanie programu, gdyż już w definicji funkcji jej nazwa zostanie uznana za identyfikator i będzie zastąpiona odpowiednim tekstem.

3. W zasięgu definicji

```
#define PETLA(a,b) a{ b; }
```

napis

```
PETLA(for(x=1;x<5;x++),y++);
/* ... */
PETLA(while(!kbhit()),);
```

zostanie zastąpiony napisem

```
for(x=1;x<5;x++){ y++; };
/* ... */
while(!kbhit()){ ; };
```

Ten trochę bezsensowny z praktycznego punktu widzenia przykład dosadnie ilustruje fakt, że argumenty identyfikatora mogą być dowolnymi napisami (nie są

w niczym podobne do argumentów funkcji). Wynika to bezpośrednio z faktu, że preprocesor (co ciągle podkreślam) jest programem manipulującym napisami.

**4.** Poniższy przykład ilustruje wykorzystanie specjalnego znaczenia znaku #.

Przetwarzając poniższy fragment kodu:

```
#define PRINT(x,y) fprintf(std ## x,#y)
PRINT(out,standardowe wyjście);
PRINT(prn,drukarka);
PRINT(err,strumień błędów);
```

preprocesor wygeneruje następujące wiersze:

```
fprintf(stdout,"standardowe wyjście");
fprintf(stdprn,"drukarka");
fprintf(stderr,"strumień błędów");
```

# 2. Zastosowania dyrektywy #define

## 2.1. Definiowanie "stałych"

Klasycznym zastosowanie dyrektywy **#define** jest definiowanie "stałych". Używam tej nazwy w cudzysłowach gdyż, jak wiemy, w rzeczywistości nie jest to definiowanie stałej konkretnego typu lecz przypisanie identyfikatorowi ciągu znaków. W programach w języku C występuje zwykle wiele stałych używanych do operacji niskiego poziomu, których wartości niewiele mówią. Przykładem mogą tu być numery funkcji DOS-a lub parametry tych funkcji. W takich przypadkach do dobrego stylu programowania należy zdefiniowanie dla tych stałych identyfikatorów o nazwach wyjaśniających do czego służą i co znaczą. Na przykład:

```
#define Ctrl_L 12
#define KURSYWA "\x1b4"
#define NO_CURSOR       0x3C0D
#define NORMAL_CURSOR   0x0C0D
#define SOLID_CURSOR    0x050B
```

Identyfikator Ctrl_L można wykorzystać do sprawdzenia czy z klawiatury został wprowadzony znak CTRL+L, KURSYWA wysłane do drukarki spowoduje włączenie kursywy, natomiast trzy ostatnie identyfikatory są parametrami funkcji BIOS-u zmieniającej rozmiar kursora. Jest chyba oczywiste, że program napisany z użyciem tych identyfikatorów będzie o wiele bardziej czytelny niż program, w którym byłyby bezpośrednio użyte odpowiednie wartości.

Czytelność programu jest wiec pierwszym argumentem za użyciem definicji "stałych". Inną sytuacją, w której należy używać identyfikatora a nie bezpośrednio stałej, jest program, zawierający parametry definiowane przed kompilacją, np. dopuszczalny rozmiar danych. Parametr taki jest zwykle wykorzystany

w kilku miejscach programu (np. w definicji tablicy, w warunku pętli, itp.). W tym przypadku zdefiniowanie identyfikatora nie tylko czyni program bardziej czytelnym ale, co ważniejsze, umożliwia prostą zmianę parametru. Wystarczy zmienić tylko wartość przypisaną identyfikatorowi w definicji, a preprocesor podstawi nową wartość we wszystkich właściwych miejscach. Oprócz wygody, zabezpiecza to nas przed sytuacją gdy w niektóre miejsca wpiszemy nową stałą a w niektórych zostawimy starą.

Parę słów chciałbym poświęcić identyfikatorom, których wartość ma być częścią jakiegoś łańcucha w programie. Problem polega tu na tym, że preprocesor nie przetwarza tekstu umieszczonego w cudzysłowach a więc nie wstawi ciągu przypisanego identyfikatorowi do łańcucha. Na przykład w zasięgu definicji

```
#define KURSYWA "\x1b4"
```

następujący tekst źródłowy

```
fputs("To jest druk normalny KURSUWA a to  kursywa",stdprn);
```

powoduje wygenerowanie kodu drukującego na drukarce dosłownie tekst

To jest druk normalny KURSYWA a to kursywa a nie, jak byśmy chcieli

To jest druk normalny *a to kursywa*

Z tej konkretnej sytuacji możemy się wykpić wykorzystując możliwości funkcji **printf**:

```
fprintf(stdprn,"To jest druk normalny%s a to kursywa",KURSYWA);
```

Co jednak zrobić, jeżeli napis przypisany identyfikatorowi musi być podstawiony do jakiegoś łańcucha i nie można tego obejść? Żeby nie być gołosłownym załóżmy, że w pisanym programie nie jest z góry ustalone rozszerzenie jego plików roboczych, powinno ono być przypisane odpowiedniemu identyfikatorowi aby można je było w przyszłości łatwo zmienić.

Żeby to zrobić, wykorzystamy pewną cechę języka C. Otóż sąsiadujące ze sobą literały łańcuchowe, np.

```
"ala ma" "kota"
```

są przez kompilator C traktowane jak jeden łańcuch:

```
"ala ma kota"
```

Wykorzystując tę cechę możemy zdefiniować identyfikator EXT:

```
#define EXT ".$$$"
```

i używać go w następujący sposób:

```
fopen("baza"EXT,"w");
```

Także poprzedni przykład z drukarką możemy zapisać następująco:

```
fputs("To jest druk normalny "KURSYWA"a to kursywa",stdprn);
```

## 2.2. Makrodefinicje

Klasycznym zastosowaniem identyfikatorów z parametrami jest tworzenie swego rodzaju makrodefinicji. Są to zwykle definicje zastępujące jakieś skomplikowane wyrażenia. Dobrymi przykładami są tu standardowe makrodefinicje umożliwiające dostęp do argumentów funkcji ze zmienną liczbą argumentów umieszczone w pliku nagłówkowym "**stdarg.h**". Operują one na dość niskim poziomie, mogą więc różnić się sposobem implementacji. Polecam przestudiowanie ich jako pouczających przykładów zastosowania makrodefinicji.

Wcześniej pokazałem już przykład makrodefinicji wyznaczającej maksimum dwóch wartości:

```
#define max(x,y) ((x)>(y)?(x):(y))
```

Na przykładzie tej prostej makrodefinicji omówię zasady, których należy przestrzegać przy konstrukcji takich definicji oraz niebezpieczeństwa, jakie czyhają na programistę przy ich używaniu. Wszystkie te zagrożenia wynikają bezpośrednio z faktu, że zinterpretowanie makrodefinicji nie polega na obliczeniu czegoś lecz na zastąpieniu jej odpowiednim tekstem, który następnie jest kompilowany.

Po pierwsze, zarówno całe wyrażenie przypisane identyfikatorowi max, jak i poszczególne identyfikatory parametrów są ujęte w nawiasy. Nawiasy otaczające całe wyrażenie stanowią zabezpieczenie na wypadek, gdyby w wyniku podstawienia wyrażenie to znalazło się w sąsiedztwie operatorów o wyższym priorytecie. Gdyby definicja wyglądała następująco

```
#define max(x,y)  x>y?x:y
```

wówczas instrukcja

```
if(x==max(a,b)){ /* ... */ }
```

nie porównałaby zmiennej x z większą z liczb a i b. Po zastąpieniu przez preprocesor identyfikatora max odpowiadający mu napisem wiersz wyglądałby następująco:

```
if(x==a>b?a:b){ /* ... */ }
```

Ze względu na priorytety operatorów najpierw zostanie wykonane porównanie zmiennych a i b, potem wynik porównania (0 lub 1) zostanie przyrównany do

zmiennej x i, w zależności od wyniku tej operacji, wartością wyrażenia w instrukcji **if** będzie wartość zmiennej a lub b.

Nawiasy wokół identyfikatorów parametrów zabezpieczają natomiast przed sytuacją gdy sam argument jest wyrażeniem. Przykładowo, instrukcja

```
maximum=max(a&15,b);
```

po zastąpieniu identyfikatora max odpowiednim napisem przyjmie postać

```
maximum=a&15>b?a&15:b;
```

Tym razem źródłem problemu jest priorytet operatora >, wyższy od priorytetu operatora iloczynu logicznego &: najpierw zostanie wykonane porównanie 15>b.

Jak widać, ewentualne błędy powstałe z powodu braku nawiasów mogą być bardzo trudne do wykrycia. Dla bezpieczeństwa warto przyjąć zasadę, że lepiej użyć o pięć nawiasów za dużo niż o jeden za mało.

Kolejna ważna uwaga dotyczy argumentów makrodefinicji. Ponieważ po rozwinięciu makrodefinicji argument może zostać wstawiony do tekstu kilkakrotnie (w makrodefinicji max każdy argument jest wstawiany dwukrotnie) należy unikać używania jako argumentów wyrażeń mających efekty uboczne, w tym także wywołań funkcji. Jeżeli jest to konieczne, trzeba to robić z dużą ostrożnością. Oto kilka tragicznych w skutkach przykładów:

```
#define max(x,y) ((x)>(y)?(x):(y))

maximum=max(++a,10);
                /*wynikiem będzie 10 albo a+2   */
                /*a będzie zwiększone o 1 lub 2 */

maximum=max(fgetc(file),maximum)
                /*tak można znaleźć sąsiada     */
                /*największego znaku w pliku    */
maximum=max(rand(),maximum)
                /* !!!!!!                       */
```

Dla programującego w języku C odruchem staje się kończenie każdej instrukcji średnikiem. Ponieważ dyrektywy preprocesora nie są instrukcjami języka C użycie średnika nie jest wymagane. Umieszczenie średnika na końcu wiersza zaczynającego się dyrektywą **#define** nie spowoduje błędu kompilacji i samo w sobie nie jest błędem, trzeba jednak zdawać sobie sprawę, jakie jest znaczenie średnika w tym miejscu. Będzie on, jak każdy inny znak po prostu częścią tekstu, którym zostanie zastąpione każde wystąpienie identyfikatora. Jeżeli nasza przykładowa makrodefinicja byłaby zdefiniowana następująco:

```
#define max(x,y) ((x)>(y)?(x):(y));
```

to użycie jej w kontekście typu:

```
maximum=max(a,maximum);
```

będzie jak najbardziej poprawne. Proszę jednak zauważyć, że nie moglibyśmy jej użyć jako wyrażenia, np.:

```
printf("%d",max(a,b));
```

Powyższy wiersz spowodowałby błąd kompilacji (najprawdopodobniej brak nawiasu zamykającego), gdyż przed nawisem zamykającym ')' w wywołaniu funkcji znalazłby się średnik. Żeby uniknąć takich niespodzianek dobrze jest przyjąć, że *nie kończy się średnikiem* makrodefinicji "zwracającej" wartość. Będzie jej wtedy można użyć jako wyrażenia, a jeżeli zostanie ona użyta jako instrukcja, to i tak "odruchowo" zostanie zakończona średnikiem.

## 2.3. Inne zastosowania definicji identyfikatorów

Przy bardziej wyszukanych zastosowaniach definicji preprocesora wykorzystuje się strukturę blokową programu w języku C. Po pierwsze, jeżeli w tekście przypisanym identyfikatorowi znajduje się więcej niż jedna instrukcja, to cały ten tekst umieszcza się wewnątrz nawiasów klamrowych, tak żeby identyfikator mógł być wykorzystywany jak pojedyncza instrukcja. Po drugie, wykorzystuje się możliwość definiowania zmiennych lokalnych w dowolnej instrukcji blokowej, co pozwala na używanie w definicji zmiennych pomocniczych.

Na przykład:

```
#define DOPISZ(name,f,thing)  {
\
                               FILE *_out;
\
                               _out=fopen("a:\\kat\\"#name,"a");
\
                               fprintf(_out,f,thing);
\
                               fclose(_out);
\
                              }
DOPISZ(tmp.$$$,"%d",x);    /* dopisanie na koniec pliku  */
                           /* a:\kat\tmp.$$$ wartości x  */
                           /* jako liczby całkowitej     */
```

Znak \ na końcu wiersza w definicji identyfikatora oznacza, że definicja jest kontynuowana w następnym wierszu. Dla zwiększenia czytelności można ten znak poprzedzić odpowiednią ilością odstępów, tak żeby w edytorze znalazł się w ostatniej kolumnie ekranu.

Po prześledzeniu poprzedniego przykładu łatwo już będzie napisać makrodefinicję zamieniającą wartości zmiennych podanych jako jej argumenty. Potrafimy

zdefiniować potrzebną zmienną pomocniczą jako lokalną zmienną instrukcji blokowej: pozostaje się jeszcze zastanowić, jakiego typu ma być ta zmienna. Można oczywiście ustalić, że makrodefinicja będzie działać tylko dla zmiennych typu **int** i takiego typu zmiennej pomocniczej używać. Większą uniwersalność zapewnia jednak wykorzystanie możliwości języka polegającej na automatycznej konwersji między wszystkimi typami numerycznymi. Możemy więc napisać uniwersalną makrodefinicję dla zmiennych dowolnego typu numerycznego:

```
#define SWAP(x,y)      {                          \
                         double _tmp;             \
                         _tmp=x;                  \
                         x=y;                     \
                         y=_tmp;                  \
                       }
float f=3.14,g=1.1;
char  c='A';

/* ... */

SWAP(f,g);             /* f=1.1  g=3.14 */
SWAP(c,g);             /* c='♥'  g=65.0 */
```

Czasami w celu utworzenia skomplikowanych konstrukcji strukturalnych definiuje się dwa lub więcej identyfikatorów, które nie mogą wystąpić samodzielnie, lecz są składowymi jakiejś większej całości. Prostą ilustracją tego typu definicji są identyfikatory repeat i until, które mogą zastępować pętlę **do while**:

```
#define repeat do{
#define until(x) }while(!(x));

/* ... */

repeat
  c=fgetc(file);
until(c=='♥')
```

Innym przykładem (bardziej praktycznym) będzie zbudowanie makrodefinicji ułatwiających korzystanie z funkcji ze zmienną liczbą argumentów (patrz rozdział "Formatowane wejście/wyjście"). Argumenty takiej funkcji pobiera się przy pomocy makrodefinicji zawartych w standardowym pliku nagłówkowym "**stdarg.h**". Schemat zastosowania tych definicji w funkcji jest następujący:

```
     fun(char *control, ... ) /* przykładowy nagłówek, może być
inny */
     {
      va_list arguments;
      va_start (arguments,control);
      /* ... */
      x=va_arg(arguments,int);       /* pobranie argumentu typu
int */
      /* ... */
      va_end(arguments);
     }
```

Standardowe makrodefinicje te mają następujące niedogodności:

- należy jawnie zadeklarować zmienną typu va_list, której się jednak zwykle nie używa jawnie,
- tę zmienną trzeba zainicjować makrem va_start,
- w każdej makrodefinicji va_arg udostępniającej kolejny argument trzeba umieścić identyfikator wyżej wspomnianej zmiennej.

Żeby nie utracić cechy przenośności programu będziemy oczywiście używać standardowych definicji z pliku "**stdarg.h**". Ukryjemy jednak niepotrzebne szczegóły: definicję zmiennej typu va_list i jej inicjację.

```
/* Plik moje_arg.h */
#include <stdarg.h>

#define start_va(a)      {
\
                           va_list _arguments;
\
                           va_start(_arguments,a);
#define end_va             va_end(_arguments);
\
                         }

#define arg_va(type)       va_arg(_arguments,type)
```

W zasięgu powyższych definicji, definicja funkcji ze zmienną liczbą argumentów może wyglądać następująco:

```
void fun(char *control,...)
{
 start_va(control);
 /* ... */
 arg_va(int);              /* pobranie argumentu typu
int */           /* ... */
 end_va;
}
```

Prawda, że jest to czytelniejsze?

Ktoś mógłby się zastanawiać, dlaczego "mózgi z ANSI" nie zdefiniowały tego od razu w ten sposób. Otóż mózgi nie pomyliły się. Pisząc o "niepotrzebnych szczegółach" nieco przesadziłem; w pewnych sytuacjach zmienna typu va_list, której definicję "ukryliśmy", musi być użyta w sposób jawny i jest niezbędna znajomość jej nazwy. Taka sytuacja występuje przykładowo, gdy funkcja o zmiennej liczbie argumentów chce przekazać swoje argumenty innej funkcji (patrz rozdział "Formatowane wejście/wyjście"). Jedynym sposobem jest wtedy przekazanie wartości właśnie rzeczonej zmiennej typu va_list.

Spróbujmy teraz wykorzystać definicje preprocesora do zaimplementowania odpowiednika instrukcji **switch** operującego na łańcuchach. Wyrażenie, na pod-

stawie którego będzie podejmowana decyzja będzie typu tekstowego (*char); stałe w przypadkach CASE także będą tekstami.

Wydaje się koniecznym, aby zarówno cała konstrukcja SWITCH, jak każdy przypadek CASE składały się z par identyfikatorów: otwierającego i zamykającego. Dla całej instrukcji będą to identyfikatory SWITCH i END_SWITCH a dla przypadków CASE i BREAK. Identyfikator SWITCH będzie posiadał jeden parametr, tj. wyrażenie, na podstawie którego będzie podejmowana decyzja. Parametr identyfikatora CASE będzie natomiast określał, dla jakiej wartości wyrażenia w identyfikatorze SWITCH mają być wykonane instrukcje dla danego przypadku.

Warunek sprawdzający równość wyrażenia i stałej musi znajdować się w tekście przypisanym identyfikatorowi CASE. Żeby przekazać każdej frazie CASE wartość wyrażenia podanego w identyfikatorze SWITCH skorzystamy ze zmiennej lokalnej

```
#define SWITCH(x)  do{ char *_ptr=x;
#define CASE(y)    if(!strcmp(_ptr,y)){
```

Tekst przypisany identyfikatorowi SWITCH rozpoczyna pętlę **do while**. Umieszczenie całości w pętli służy tylko temu, żeby móc używać instrukcji **break** do natychmiastowego wyjścia z konstrukcji SWITCH.

```
#define END_SWITCH }while(0);
#define BREAK      break; }
```

Pozostaje jeszcze tylko zdefiniować identyfikator DEFAULT

```
#define DEFAULT   {
```

Nawias { służący tutaj wyłącznie sparowaniu nawiasu powstałego z rozwinięcia identyfikatora BREAK kończącego przypadek **default.**

Komplet definicji znajduje się w pliku **switch.h**.

Jak widać, definicje są proste i tworzą całkiem przyzwoity odpowiednik instrukcji **switch**. Jako przykład ich zastosowania przedstawiam poniżej fragment kodu rozpoznający argumenty wywołania programu.

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

/* definicje zmiennych */

main(int argc,char *argv[])        /* uproszczona UNIX-owa funkcja
find */
{
 if(argc<4)                /* co najmniej 3 argumenty + nazwa
programu = 4 */
   {
    fprintf(stderr,"find pathname expression\n");
    return;
   }
```

```
 path=argv[1];                              /* pierwszy argument to
ścieżka */
 for(x=2;x<argc-1;x++)     /* pozostałe określają jakich plików
szukać */
  SWITCH(argv[x])
   {
    CASE("-name")
                file=argv[++x];    /* po -name ma być nazwa pliku */
                  BREAK;
    CASE("-type")
              filetype=*argv[++x];  /* typ pliku: jedna litera */
                  BREAK;
    CASE("-group")
               gname=argv[++x];    /* nazwa grupy użytkowników */
                  BREAK;
    CASE("-user")
               uname=argv[++x];    /* nazwa właściciela pliku  */
                  BREAK;
    CASE("-size")
                size=atoi(argv[++x]);             /* rozmiar pliku */
                  BREAK;
    CASE("-atime")
                  atime=atoi(argv[++x]);   /* kiedy był uruchomiany */
                  BREAK;

    /* ... */                              /* i tak dalej ...       */

    DEFAULT
                fprintf(stderr,"find pathname expression\n");
                return;
                BREAK;             /* inna sekwencja oznacza błąd */
   }
  END_SWITCH;

/* ... */

}
```

Wiele przykładów praktycznego zastosowania definicji preprocesora zawiera rozdział tej książki pod tytułem "Programowanie współbieżne"

# III. Formatowane wejście/wyjście

Język C nie posiada żadnych instrukcji wejścia/wyjścia. Przenośnym sposobem realizacji komunikacji programu ze światem zewnętrznym jest korzystanie ze standardowych funkcji bibliotecznych ANSI, których sposób działania jest ściśle zdefiniowany. Do kompilatora zgodnego ze standardem ANSI dołączone są biblioteki zawierające definicje tych funkcji i zawsze, niezależnie od kompilatora czy komputera, działają one tak samo. Ta pełna przenośność powoduje czasami, iż zapomina się o tym, że są to jednak funkcje, które w żaden sposób nie różnią się od innych funkcji napisanych w języku C[4)].

Pisząc o standardowych funkcjach formatowanego wyjścia mam na myśli funkcję printf i jej pochodne (zarówno te zdefiniowane w bibliotekach standardu ANSI, jak i te, które stanowią rozszerzenie tego standardu i są dostępne w niektórych kompilatorach). Funkcje formatowanego wejścia to funkcja **scanf** oraz rodzina jej pochodnych. Funkcje formatowanego we/wy są wśród standardowych funkcji wejścia/wyjścia funkcjami najwyższego poziomu. W rozdziale tym będę używał nazw "printf" i "scanf" zarówno do określenia konkretnych funkcji **printf** i **scanf** jak i całej rodziny funkcji pochodnych.

Funkcje **printf** i **scanf** są funkcjami o zmiennej liczbie argumentów. Ponieważ w dalszej części opisu będę często odwoływał się do pojęć związanych z takimi funkcjami, omówię teraz krótko mechanizm ich działania i sposób definiowania.

## 1. Funkcje ze zmienną liczbą argumentów

Mimo, że w języku C można tworzyć funkcje pobierające zmienną liczbę argumentów, to składnia języka nie zawiera pełnych mechanizmów umożliwiających ich użycie. Jedyną rzeczą, jaką zapewnia język C, jest możliwość zdefiniowania funkcji, dla której nie będzie sprawdzane, czy liczba argumentów w wywołaniu jest równa liczbie zadeklarowanych parametrów. Natomiast pobieranie argumentów funkcji o zmiennej ich liczbie musi być rozwiązane przez programistę, na poziomie bezpośrednich odwołań do pamięci. Mimo, że istnieje standard takich odwołań i w każdym kompilatorze dostępne są odpowiednie makrodefinicje, uważam (i postaram się to poniżej udowodnić), że używając tych makrodefinicji trzeba zdawać sobie sprawę jak one działają i co robią. Dlatego też poniższy opis będzie zawierał wiele szczegółów dotyczących sposobu realizacji przez kompilator pewnych rzeczy, których normalnie programista nie musi znać.

Ogólnie przyjętym rozwiązaniem, stosowanym nie tylko w języku C, jest przekazywanie argumentów przez stos. Przed wywołaniem funkcji, na stosie umiesz-

[4] Zresztą większość funkcji bibliotecznych jast napisana w C

czane są wartości kolejnych argumentów. Klasyczna funkcja, wiedząc ile i jakiego typu argumentów posiada, może odpowiednio zinterpretować zawartość stosu. Proszę zwrócić uwagę, jak ważna jest znajomość liczby i typów argumentów. Bez tej informacji funkcja nie jest w stanie w żaden sposób stwierdzić, które wartości na stosie są jej argumentami i co one oznaczają. Przykładowo niemożliwe jest stwierdzenie, czy dwa kolejne bajty o wartościach 65 i 0 reprezentują znak 'A', czy też może są fragmentem jakiejś liczby zmiennoprzecinkowej. Dlatego też dla "normalnej" funkcji należy zadeklarować liczbę i typy jej parametrów. Kompilator sprawdza, czy liczba argumentów w wywołaniu funkcji jest zgodna z definicją funkcji a także przeprowadza odpowiednie konwersje argumentów na podstawie typów parametrów formalnych. Rozważmy następujący przykład:

```
void  fun(float f)
{ /*...*/ }

/*...*/

int I=1;

fun(I);
```

W wywołaniu funkcji **fun** jako argument podana jest zmienna typu **int**, która jest zapisana w dwóch bajtach pamięci (dla uproszczenia zakładam, że typ **int** jest reprezentowany dwoma bajtami, choć może oczywiście być inaczej). Kompilator na podstawie definicji funkcji **fun** "wie" jednak, że "spodziewa" się ona wartości typu **float**. Gdyby wygenerowany kod umieścił na stosie tylko dwa bajty (1 i 0), to funkcja **fun**, odczytując liczbę zmiennoprzecinkową, otrzymałaby wartość przypadkową (po pierwsze odczytałaby więcej bajtów niż zostało umieszczone na stosie, a po wtóre liczby typu float mają inną reprezentację wewnętrzną niż liczby typu int). W rzeczywistości kompilator, wiedząc o tym, że argument funkcji ma być typu **float**, wygeneruje kod dokonujący konwersji wartości zmiennej I na postać zmiennoprzecinkową i umieszczający na stosie odpowiednią ilość bajtów. Jest to właśnie automatyczna konwersja argumentów do typu odpowiedniego parametru formalnego.

Kompilator języka C wykonuje jeszcze jedną automatyczną konwersję. Zmienne typu **char**, które są jednobajtowe i tyle miejsca zajmują normalnie w pamięci, są automatycznie konwertowane na wartości typu **int**. Konwersja ta wykonywana jest także wtedy, gdy funkcja ma zadeklarowany parametr formalny typu **char**.

Efekt ten ilustruje następujący program:

```
#include <stdio.h>
void  fun(char a,char b)
{
 printf("%d\n",&b-&a);
}
char x,y;
void  main()
```

```
{
 printf("%d\n",&y-&x);
 fun(x,y);
}
```

W wyniku działania tego programu wypisane zostaną różnice adresów dwóch sąsiednich zmiennych globalnych typu **char** i dwóch argumentów typu **char**. Ponieważ globalne zmienne znakowe zajmują jeden bajt, w pierwszym wywołaniu funkcji **printf** wypisana zostanie wartość 1. W drugim wywołaniu funkcji **printf**, z wnętrza funkcji **fun**, wypisana zostanie jednak wartość 2, bo argumenty typu **char** zajmują dwa bajty - tyle, ile wartości typu **int**.

Fakt, że argumenty typu **char** zajmują dwa bajty a nie jeden, jak zmienne typu **char**, nie ma zwykle dla programisty znaczenia. Należy jednak o nim pamiętać, ponieważ czasami może być powodem bardzo trudnych do wykrycia błędów. Jako ostrzeżenie chciałbym przytoczyć błąd jaki popełnili programiści firmy Microsoft. Błąd ten znalazł się w standardowym pliku "**stdarg.h**" dołączonym do kompilatora Microsoft C 5.1. Plik "**stdarg.h**", opisany dokładniej w dalszej części tego rozdziału, zawiera makrodefinicje udostępniające argumenty funkcji o zmiennej liczbie argumentów. W kompilatorze Microsoft makrodefinicje te działają niepoprawnie, jeżeli ostatni obowiązkowy argument jest typu **char**. Błąd ten spowodowany jest właśnie nieuwzględnieniem faktu, że argumenty typu **char** zajmują dwa bajty (podczas gdy operator **sizeof** dla takiego argumentu da wartość 1). Zachęcam Czytelników mających dostęp do tego kompilatora, do przestudiowania zawartości pliku "**stdarg.h**" i przetestowania błędnego działania makrodefinicji.

W niektórych kompilatorach, np. w wymienionym wcześniej Microsoft C 5.1, nie tylko argumenty znakowe ale także lokalne zmienne typu char zajmują dwa bajty.

Podobnie jak konwersje typu **char** na **int**, niektóre kompilatory mogą przeprowadzać konwersję argumentu typu **float** na **double**. Konwersja ta nie jest bezwzględnie przestrzegana, tak więc różne kompilatory mogą się różnie zachowywać. Obowiązkowo jest ona jednak wykonywana na argumentach funkcji o zmiennej liczbie argumentów. Gdyby nie była ona przeprowadzana, nie mogłaby poprawnie działać na przykład funkcja **printf**.

Dotychczasowe rozważania dotyczyły "klasycznych" funkcji z określoną liczbą i typami argumentów. Jak widać, kompilator w dużej mierze uniemożliwia popełnienie błędów przy przekazywaniu argumentów takim funkcjom.

Funkcje ze zmienną liczbą argumentów są natomiast ucieleśnieniem zasady "programista ma zawsze rację". Jak wynika z samej nazwy, funkcja taka nie wie ani ile argumentów otrzyma na stosie, ani jakiego będą one typu (ile bajtów będzie je reprezentować).

W tym miejscu należy wspomnieć o czymś, co nas nie interesowało gdy mowa była o klasycznych funkcjach. Jak wcześniej zostało powiedziane, argumenty funkcji są przekazywane przez stos. Stos ma tę właściwość, że jeżeli coś na nim umieścimy, to należy to bezwzględnie zdjąć. W przypadku argumentów funkcji możliwe są dwa rozwiązania: albo argumenty ze stosu zdejmuje ta funkcja, która je tam przed wywołaniem umieściła (wywołująca), albo ta, której te argumenty zostały przez stos przekazane (wywoływana). W języku C za zdejmowanie argumentów ze stosu odpowiedzialna jest funkcja wywołująca. W przypadku funkcji ze zmienną liczbą argumentów ma to kapitalne znaczenie: funkcja wywołana nie mogłaby przecież zdjąć ze stosu swoich argumentów ponieważ z definicji nie wie ile ich tam jest! Mechanizm przekazywania argumentów jest więc następujący: przed wywołaniem funkcji (każdej, nie tylko funkcji ze zmienną liczbą argumentów) na stosie umieszczane są jej argumenty, do których funkcja odwołuje się jak do zmiennych lokalnych, (adresując po prostu odpowiednie miejsce w pamięci). Po powrocie z funkcji, ze stosu jest usuwana odpowiednia liczba bajtów (modyfikowany jest odpowiednio wskaźnik stosu). W ten sposób kompilator zabezpiecza stos przed błędami programisty, także w przypadku funkcji ze zmienną liczbą argumentów.

Jak w praktyce wygląda odwołanie się do argumentów w funkcji ze zmienną liczbą argumentów? Oczywiście argumenty te nie posiadają identyfikatorów. Funkcja wie tylko, że znajdują się one na stosie, oraz że występują kolejno po ostatnim obowiązkowym argumencie (funkcja musi mieć co najmniej jeden obowiązkowy argument, który wystąpi w każdym wywołaniu). Do pobierania tych argumentów służą makrodefinicje zdefiniowane w standardowym pliku nagłówkowym "**stdarg.h**". Szczegółowa ich realizacja może być zależna od kompilatora i komputera (gorąco polecam przestudiowanie zawartości pliku "stdarg.h" dołączonego do używanego kompilatora), ale sposób ich działania jest zdefiniowany w standardzie ANSI i mogą one być używane do pisania przenośnych programów.

Znaczenie poszczególnych makrodefinicji z pliku **stdarg.h** omówię na przykładzie:

```
void modul(char *types,...)
{
 double a,b;
 va_list arguments;
 va_start(arguments,types);
 while(*types)
  switch(*(types++))
   {
    case 'R':                                  /* liczba
rzeczywista */
    case 'r': a=va_arg(arguments,double);
              printf("|%f|=%f\n",a,a<0?-a:a);
              break;
    case 'c':                                    /* liczba
zespolona */
```

```
    case 'C': a=va_arg(arguments,double);
              b=va_arg(arguments,double);
              printf("|(%f,%f)|=%f\n",a,b,sqrt(a*a+b*b)); break;
    default : fprintf(stderr,"Błąd\n"); break;
   }
 va_end(arguments);
}
/*... przykładowe wywołanie funkcji modul */

modul("ccr",1.0,1.0,0.0,2.0,-4.2);
```

Funkcja **modul** wypisuje moduły liczb podanych jako argumenty. Liczby mogą być zarówno liczbami rzeczywistymi jak i zespolonymi. Liczby zespolone podaje się jako dwie kolejne liczby zmiennoprzecinkowe odpowiadające części rzeczywistej i urojonej. Obowiązkowy parametr types służy do poinformowania funkcji, jak ma zinterpretować argumenty na stosie. Litera 'c' w łańcuchu wskazywanym przez argument types oznacza, że dwie kolejne liczby zmiennoprzecinkowe należy potraktować jako część rzeczywistą i urojoną liczby zespolonej. Litera 'r' oznacza, że kolejna liczba zmiennoprzecinkowa jest liczbą rzeczywistą.

Proszę teraz przyjrzeć się definicji tej funkcji.

W nagłówku funkcji ze zmienną liczbą argumentów umieszcza się najpierw wszystkie obowiązkowe parametry (w naszym przykładzie jeden: `char *types)` a po nich parametr `... (trzykropek).` W funkcji należy zadeklarować lokalną zmienną typu va_list. Przed pierwszym odwołaniem do argumentów makrem va_arg, należy umieścić makro va_start, którego pierwszym argumentem jest zmienna typu va_list, a drugim nazwa ostatniego obowiązkowego argumentu funkcji. Makro va_start służy do zainicjowania zmiennej typu va_list i umieszcza się je zwykle bezpośrednio za definicjami zmiennych lokalnych. Wartości kolejnych argumentów można otrzymać przy pomocy makra va_arg, którego pierwszym argumentem jest zmienna typu va_list, a drugim nazwa typu argumentu. Wynika z tego, że programista jest całkowicie odpowiedzialny za to jak zostaną zinterpretowane wartości znajdujące się na stosie. W przykładowej funkcji każdy argument traktowaliśmy jako liczbę zmiennoprzecinkową. Proszę zwrócić uwagę, że w przypadku funkcji ze zmienną liczbą argumentów kompilator nie ma żadnych podstaw do wykonania automatycznych konwersji argumentów wywołania, dlatego też jeżeli chcemy przekazać np. liczbę 1 jako liczbę zmiennoprzecinkową, musimy jawnie użyć zapisu 1.0. Gdybyśmy chcieli policzyć przy pomocy naszej funkcji wartości bezwzględne jakiejś zmiennej typu **int**, musielibyśmy użyć jawnej konwersji do typu **float** lub **double**:

```
int x=1;
modul("r",(float)x);
```

Definicja funkcji ze zmienną liczbą argumentów powinna być zakończona makrem va_end z zadeklarowaną wcześniej zmienną typu va_list jako argumentem.

Używając funkcji o zmiennej liczbie argumentów programista musi zdawać sobie sprawę, że jest w nich odpowiedzialny za to, co normalnie robi kompilator. Dlatego też trzeba "myśleć na niższym poziomie", uwzględniając np. rozmiar argumentów na stosie. Jeżeli chcemy pobrać argument znakowy (typu **char**), jako drugi argument makra va_arg podać należy typ **int**. Wynika to bezpośrednio z tego, że argumenty typu **char** zajmują na stosie dwa bajty (są zawsze poddawane automatycznej konwersji do typu **int**). Analogicznie, chcąc pobrać wartość zmiennoprzecinkową, należy w makrodefinicji va_arg używać typu **double**, nigdy zaś **float**.

Po tym ogólnym wstępie, dotyczącym funkcji ze zmienną liczbą argumentów, możemy zająć się właściwym tematem tego rozdziału, czyli funkcjami formatowanego wejścia i wyjścia.

# 2. Funkcje formatowanego wyjścia

## 2.1. Funkcja printf

Deklaracja:

```
int printf(const char *format, ... );
```

Funkcja **printf** wyprowadza swoje argumenty na standardowe wyjście. Normalnie jest to konsola użytkownika, zwykle jednak system operacyjny umożliwia podmianę konsoli dowolnym plikiem a także, przy użyciu mechanizmu potoków, skierowanie wyjścia na wejście innego programu.

Funkcja **printf** ma jeden obowiązkowy parametr: const char *format. Argument skojarzony z tym parametrem wskazuje na ciąg znaków informujący funkcję printf, jakie i ile argumentów zostało jej przekazanych, a także w jaki sposób mają one być wypisane. Funkcja **printf** czyta kolejne znaki ciągu format, i do momentu napotkania znaku % (procent) przesyła przeczytane znaki do standardowego wyjścia. Znak % rozpoczyna sekwencję sterującą, informującą funkcję **printf**, że na stosie znajduje się argument, określającą typ tego argumentu a także sposób jego wyprowadzenia. W ciągu wskazanym przez argument format może wystąpić dowolna ilość takich sekwencji sterujących. Liczba argumentów w wywołaniu funkcji **printf** (oczywiście z wyjątkiem argumentu format) powinna być dokładnie taka, jaka wynika z sekwencji sterujących w ciągu wskazanym przez argument format. Jeżeli liczba argumentów w wywołaniu jest większa, to nadmiarowe (ostatnie) argumenty zostaną zignorowane. Jeżeli argumentów jest mniej niż wynikałoby to z ciągu format, to funkcja **printf** w dobrej wierze wypisze "śmieci" pobrane ze stosu (oczywiście sformatowane zgodnie z odpowiednimi sekwencjami sterującymi w ciągu format).

Sekwencja sterująca rozpoczyna się znakiem % i kończy specyfikatorem określającym typ argumentu i sposób, w jaki ma on być wyprowadzony. Specyfikatorem może być jedna z następujących liter:

d Argument typu **int** zostanie wypisany jako liczba dziesiętna

o Argument typu **int** zostanie wypisany jako liczba ósemkowa bez znaku

x Argument typu **int** zostanie wypisany jako liczba szesnastkowa

u Argument typu **int** zostanie wypisany jako liczba dziesiętna bez znaku

c Argument typu **char** zostanie wypisany jako znak

s Wypisany zostanie ciąg znaków wskazany przez argument typu **char** *

e Argument typu **float** lub **double** będzie wypisany w postaci *[-]m.nnnnnnE[±]xx*

f jak wyżej ale w postaci *[-]mmm.nnnnnn*

g Argument typu **float** lub **double** zostanie wypisany w formacie opowiadającym specyfikacji `%e` lub specyfikacji `%f` w zależności od tego, która da krótszy tekst.

Mimo, że w powyższym opisie używam słowa "typ" w stosunku do argumentów, należy pamiętać, że printf jest funkcją o zmiennej liczbie argumentów, nie rozróżnia ona więc typu argumentów i w pełni zdaje się na informacje otrzymywane w ciągu sterującym format. Powyższy opis należy rozumieć w ten sposób, że sekwencji %d powinien odpowiadać argument typu **int**, sekwencji **%f** powinien odpowiadać argument typu **float** lub **double** itd. Często zdarza się jednak inne niż standardowe skojarzenie argumentu z sekwencją sterującą. Oto kilka przykładów.

1. Jako liczbę całkowitą można polecić wydrukować kod znaku. Kompilator zadba o konwersję wartości typu char do int.

```
char c;

printf("%c-%d",c,c);
/* wypisanie znaku i jego kodu dziesiętnie       */
```

2. Przekazanie dalekiego wskaźnika jako argumentu funkcji **printf** jest równoważne przekazaniu dwóch liczb całkowitych.

```
char far *ptr;
printf("%04X:%04X",ptr);
/* wypisanie adresu znaku wskazywanego przez ptr */
/* w postaci Offset:Segment  (w systemie MS-DOS) */
```

3. Poniższy przykład jest podobny do poprzedniego. W modelu Large nazwa funkcji przekazana jako argument jest traktowana jak dalekie wskazanie do tej funkcji.

```
printf("%04X:%04X",main);
/* adres funkcji main j.w. ( w modelu Large )    */
```

Wszystkie powyższe zastosowania funkcji **printf** są poprawne, chociaż wyprowadzane argumenty nie są takiego typu, dla jakiego zaprojektowana jest dana specyfikacja.

W celu dokładniejszego formatowania wyjścia można polecić umieszczenie argumentu w tzw. polu wyjściowym. Do określenia rozmiaru tego pola oraz sposobu umieszczania w nim argumentu służą opcjonalne elementy, które mogą pojawić się między znakiem % a specyfikatorem:

### znak - (minus)

Poleca dosunięcie argumentu do lewej strony pola. Standardowo (gdy nie jest użyty znak minus) argument jest dosuwany do prawej strony pola.

### liczba

Określa minimalny rozmiar pola. Jeżeli długość wypisanego argumentu będzie mniejsza niż szerokość pola, to zostanie on uzupełniony znakami spacji. Standardowo uzupełnianie następuję z lewej strony, ale w przypadku użycia znaku minus spacje zostaną dodane z prawej strony. Jeżeli liczba określająca rozmiar pola zaczyna się znakiem 0 (zero) to do uzupełniania zamiast spacji używana jest cyfra 0 (uzupełnianie zerem jest możliwe tylko po lewej stronie, tzn. gdy nie użyto znaku -).

Użycie zamiast *liczby* znaku * (gwiazdka) oznacza, że szerokość pola jest przekazana jako kolejny argument (przed argumentem, którego dotyczy analizowana sekwencja sterująca).

### .liczba (liczba poprzedzona kropką)

W sekwencji wyprowadzającej ciąg znaków liczba ta oznacza maksymalną długość wyprowadzanego łańcucha znaków.

W sekwencji wyprowadzającej liczbę zmiennoprzecinkową określa liczbę cyfr wyprowadzanych po kropce dziesiętnej.

Użycie zamiast *liczby* znaku * (gwiazdka) oznacza, że odpowiednia wartość jest przekazana jako kolejny argument (przed argumentem, którego dotyczy analizowana sekwencja sterująca).

**l (litera el)**

Może ona wystąpić w sekwencji wyprowadzającej liczbę całkowitą i mówi, że na stosie znajduje się liczba typu **long** a nie **int**, jak przyjmowane jest domyślnie.

Wystąpienie w ciągu sterującym format napisu %% spowoduje wyprowadzenie znaku % (procent). Ciągi, w których po znaku % nie występuje żaden z opisanych wyżej specyfikatorów ani znak %, są traktowane różnie w różnych kompilatorach.

Oto kilka przykładów ilustrujących zastosowanie tych elementów specyfikacji, które mogą budzić wątpliwości (w komentarzach podano wynik działania; znaki | oznaczają granice pola:

```
printf("%*s",12,"ciąg próbny");  /* | ciąg próbny| */
printf("%*.*f",10,2,3.141);      /* |      3.14|   */
printf("%.4s","ciąg próbny");    /* |ciąg|         */
printf("%-7.4s","ciąg próbny");  /* |ciąg   |      */
printf("%2s","ciąg próbny");     /* |ciąg próbny|  */
printf("%07d",13);               /* |0000013|      */
printf("%-07d",13);              /* |13     |      */
printf("%lx",(long)-1);          /* |ffffffff|     */
printf("%.1d",10000);            /* |10000|        */
```

Funkcja **printf** zwraca wartość całkowitą równą liczbie wyprowadzonych znaków. Jeżeli w czasie wyprowadzania wystąpił jakiś błąd, funkcja ta zwraca wartość EOF.

## 2.2. Funkcja fprintf

Deklaracja:

```
int fprintf(FILE *plik,const char *format, ... );
```

Funkcja **fprintf** służy do wyprowadzania danych do pliku identyfikowanego przez obowiązkowy argument `FILE *plik.` Jej działanie i interpretacja ciągu sterującego format są identyczne jak dla funkcji **printf**. Wynikiem funkcji fprintf jest liczba wyprowadzonych znaków albo wartość EOF w przypadku wystąpienia błędu.

Istnieją trzy wyróżnione, predefiniowane pliki, do których funkcja **fprintf** może wyprowadzać informację:

- stdout – standardowe wyjście. Wywołanie funkcji **fprintf** z plikiem stdout jako argumentem jest równoważne wywołaniu funkcji **printf**.
- stderr – standardowe wyjście dla błędów. Wyjście to, zawsze gdy to jest możliwe, skojarzone jest z konsolą operatorską. Jak wskazuje nazwa, głównym przeznaczeniem pliku stderr jest wypisywanie wszelkich ko-

munikatów dla użytkownika (szczególnie informacji o błędach). Dzięki zastosowaniu pliku stderr komunikaty te będą pojawiać się na ekranie nawet wtedy, gdy wyjście programu zostanie skierowane do pliku lub na wejście innego programu.

- stdprn – standardowa drukarka.

## 2.3. Funkcja sprintf

Deklaracja:

```
int sprintf(char *ptr,const char *format, ... );
```

Funkcja **sprintf** działa tak jak printf ale wyprowadza swoje argumenty do obszaru w pamięci, wskazywanego przez obowiązkowy argument ptr. Rezultatem, tak jak w przypadku poprzednich funkcji, jest liczba wyprowadzonych znaków albo wartość EOF w przypadku wystąpienia błędu. Przykłady zastosowań funkcji **sprintf** przedstawiono przy okazji omawiania funkcji **sscanf**.

## 2.4. Funkcje vprintf, vfprintf, vsprintf

Deklaracje:

```
int vprintf(const char *format, va_list arg );
int vfprintf(FILE *plik,const char *format, va_list arg
);
int vsprintf(char *ptr,const char *format, va_list arg
);
```

Jak widać z prototypów trzy powyższe funkcje *nie są* funkcjami o zmiennej liczbie argumentów. Zamiast parametru ... posiadają one parametr typu va_list.

Żeby zrozumieć działanie i przeznaczenie tych funkcji, zastanówmy się najpierw nad kwestią bardziej ogólną: jak funkcja o zmiennej liczbie argumentów może przekazać swoje argumenty innej funkcji. Nie można po prostu umieścić ich kolejno w wywołaniu tej funkcji po pierwsze dlatego, że nie mają identyfikatorów, a po drugie (co ważniejsze) dlatego, że nie wiadomo ile ich jest. Jedyną sensowną metodą przekazania argumentów jest przekazanie wartości lokalnej zmiennej typu va_list, deklarowanej lokalnie w funkcji o zmiennej liczbie parametrów i wskazującej na listę argumentów na stosie. Opisany sposób przekazywania argumentów najlepiej zilustruje przykład:

```
float srednia(int jaka, ... )          /* w MS C   char
jaka */
 {                                /* działałoby zle     */
  va_list arg;
  va_start(arg,jaka);
  switch(jaka)
   {
     case 'w' :
     case 'W' : return _harmoniczna(arg);
```

```
            case 'a' :
            case 'A' : return _arytmetyczna(arg);
            case 'k' :
            case 'K' : return _geometryczna(arg);
           }
          va_end(arg);
         }
        float  _arytmetyczna(va_list arg)   /* średnia
arytmetyczna */
         {
          int il=va_arg(arg,int),i=il;    /* pierwszy arg. =
ilość */
          float suma=0;
          while(i--)suma+=va_arg(arg,int);           /* sumuj
... */
          return il?suma/il:0;        /* ... i podziel przez
ilość */
         }
```

Funkcja **srednia**, oblicza średnią określoną przy pomocy obowiązkowego argumentu jaka. Średnia obliczana jest na podstawie ciągu liczb całkowitych przekazanych jako kolejne argumenty, przy czym pierwsza liczba określa długość ciągu tj. ile liczb zostało przekazanych jako argumenty (nie licząc tej właśnie liczby). Funkcja **srednia** korzysta z usług trzech funkcji, którym przekazuje wartość zmiennej arg, wskazującej obszar pamięci zawierający jej argumenty. Funkcje te pobierają argumenty i obliczają ich średnią używając odpowiedniej metody. Czynność pobrania argumentów jest tutaj rozłożona na dwie funkcje. Funkcja **srednia** posiada lokalną zmienną arg typu va_list, którą inicjuje makrem va_start tak, żeby wskazywała na argumenty na stosie. Funkcja ta nie pobiera jednak swoich argumentów makrem va_arg. Natomiast funkcja **_arytmetyczna**, będąca normalną funkcją o jednym argumencie, pobiera argumenty od funkcji **srednia** w sposób typowy dla funkcji ze zmienną liczbą argumentów, używając makra va_arg.

Funkcje **vprintf**, **vsprintf** i **vfprintf** podobne są do funkcji **_arytmetyczna** opisanej w powyższym przykładzie. Mogą one być wywoływane z funkcji o zmiennej liczbie argumentów. Przykładowo:

```
void drukuj(const char *format, ... )
 {
  va_list arg;
  va_start(arg,format);
  vfprintf(stdprn,format,arg);
  va_end(arg);
 }
```

Opisywane funkcje stosowane są do pisania specyficznych funkcji formatowanego wyjścia, wykorzystujących mechanizmy funkcji standardowych.

Funkcje te oczekują, jako swojego argumentu, wartości arg zainicjowanej za pomocą makra va_start w wywołującej je funkcji o zmiennej liczbie argumentów.

Można jednak wykorzystać ich możliwości przekazując im jako argument coś innego.

Oto przykład takiego "oszukania" funkcji **vprintf**:

```
#include <stdio.h>
typedef int tab[3][3];
void macierz(tab x)
{
 vprintf("┌                  ┐\n"
         "│%5d%5d%5d  │\n"
         "│%5d%5d%5d  │\n"
         "│%5d%5d%5d  │\n"
         "└                  ┘\n",(va_list)x);
}

tab x={{1,2,192},{32,63,1092},{29,0,0}};

void  main()
{
 macierz(x);
}
```

Funkcja macierz wypisuje tablicę `int[3][3]` w eleganckiej postaci:

```
┌          ┐
│    1    2  192  │
│   32   63 1092  │
│   29    0    0  │
└          ┘
```

Funkcji **vprintf** "wydaje się", że otrzymuje wskazanie do ciągu argumentów na stosie. W rzeczywistości przekazujemy jej dwuwymiarową tablicę typu **int**, poddaną wcześniej konwersji do typu va_list. Sztuczka udaje się dzięki temu, że tablica liczb całkowitych jest w pamięci reprezentowana tak samo jak odpowiednia ilość argumentów typu **int** na stosie.

## 2.5. Funkcja cprintf

Deklaracja:

```
int cprintf(const char *format, ... );
```

Wiele kompilatorów oferuje rozszerzenia standardowych funkcji wyjściowych zdefiniowanych przez ANSI. Do takich rozszerzeń należy funkcja **cprintf**, będąca funkcją wyjścia na konsolę. Przejmuje ona ogólną "filozofię" funkcji **printf** i akceptuje takie same sekwencje sterujące.

Czym różni się funkcja **cprintf** od funkcji **printf**? Otóż funkcja **printf** wyprowadza informacje do standardowego wyjścia, nie zaś na ekran. Za to, że wyprowadzone przy pomocy funkcji **printf** napisy pojawiają się na ekranie (jeżeli standardowe wyjście jest związane z ekranem) odpowiedzialny jest system operacyjny. System operacyjny ma także pewien wpływ na sposób wypisania danych na

ekranie. Na przykład w systemie MS-DOS inny wynik może dawać program uruchomiony na komputerze z zainstalowanym sterownikiem **ANSI.SYS**, a inny na komputerze bez tego sterownika. Dzieje się tak dlatego, że pomiędzy wyjściem funkcji **printf** a ekranem konsoli "leży" system operacyjny, który przekształca strumień wysłany przez program do standardowego wyjścia na odpowiedni efekt na ekranie.

Jak natomiast działa funkcja **cprintf**? Przede wszystkim należy pamiętać, że nie należy ona do standardu ANSI C. W różnych kompilatorach może ona działać w różny sposób (niektóre kompilatory mogą w ogóle nie zawierać jej implementacji). Generalnie funkcja **cprintf** wyprowadza swoje argumenty na ekran. Wynika stąd, że zwykle z funkcją **cprintf** wiążą się takie pojęcia jak atrybuty tekstu, współrzędne na ekranie, czy rozmiar okna, do którego jest wyprowadzany tekst. Drugą różnicą jest to, że funkcja **cprintf** może niektóre znaki, będące dla systemu znakami sterującymi (np. tabulacja), wypisywać dosłownie, jako odpowiednie znaki z rozszerzonego zestawu ASCII. Wynik działania funkcji **cprintf** powinien się pojawiać zawsze na ekranie - także wtedy, gdy wyjście standardowe zostało przekierowane do pliku.

Jak napisałem, w różnych kompilatorach różnie to wygląda. Firma Borland bardzo konsekwentnie realizuje tę funkcję jako funkcję wyjścia na ekran. Funkcje ustawiające atrybuty tekstu wpływają na kolor napisów wypisywanych przez funkcję **cprintf**. Można zdefiniować rozmiar i położenie na ekranie okna, do którego funkcja **cprintf** będzie wyprowadzała napisy, zaś kody sterujące są w większości wypisywane jako odpowiednie znaki (np. Esc jako ←).

Inaczej jest w kompilatorach Microsoftu, w których funkcja **cprintf** właściwie niewiele różni się od funkcji **printf**. Jeżeli wyjście programu zostanie przekierowane do pliku, to wynik działania funkcji **cprintf** nie pojawi się na ekranie lecz zostanie również skierowany do tego pliku. Ponadto funkcje ustawiające atrybuty tekstu nie wpływają na kolor tekstu wypisywanego przez funkcję **cprintf**.

Różnice pomiędzy konsolą a standardowym wejściem i wyjściem (analogicznie do stdout jest także zdefiniowany plik stdin) są jeszcze lepiej widoczne w przypadku funkcji formatowanego wejścia: scanf (wejście z stdin) i cscanf (wejście z konsoli). Funkcje te zostaną opisane w dalszej części tego rozdziału.

# 3. Funkcje formatowanego wejścia

## 3.1. Funkcja scanf

Deklaracja:

```
int scanf(const char *format, ... );
```

Funkcja ta czyta dane ze standardowego wejścia i po odpowiednim przetworzeniu przypisuje je zmiennym, do których wskaźniki przekazywane są jako argumenty. Normalnie standardowe wejście jest związane z konsolą użytkownika. System operacyjny umożliwia zwykle zastąpienie konsoli dowolnym plikiem. Przy użyciu mechanizmu potoków, można spowodować także pobieranie danych z wyjścia innego programu.

Funkcja **scanf** ma jeden obowiązkowy parametr: `const char *format`. Argument skojarzony z tym parametrem wskazuje na ciąg znaków informujący funkcję **scanf**, jak należy zinterpretować ciąg wejściowy. Kolejne argumenty są wskaźnikami do zmiennych, którym należy przypisać zinterpretowane dane ze strumienia wejściowego. Funkcja analizuje kolejne znaki ciągu format. Jeżeli znak nie należy do sekwencji sterującej (sekwencja sterująca, podobnie jak w przypadku funkcji **printf**, rozpoczyna się znakiem %), to jest on porównywany z kolejnym znakiem z wejścia. Jeżeli znaki są różne, to funkcja **scanf** kończy swoje działanie, w przeciwnym wypadku cała procedura jest powtarzana dla następnego znaku w ciągu format i następnego znaku na wejściu. W specjalny sposób traktowane są w ciągu format białe znaki ('\n',' ','\r' oraz '\t'). Zinterpretowanie w nim nawet jednego białego znaku powoduje pominięcie w strumieniu wejściowym całego ciągu białych znaków. Jeżeli białemu znakowi lub ciągowi białych znaków w ciągu format nie odpowiada co najmniej taka sama liczba białych znaków w strumieniu wejściowym, to funkcja **scanf** kończy swoje działanie. Białe znaki nie są rozróżniane, (wszystkie są traktowane jak odstępy), dlatego też wystąpienie w ciągu sterującym format np. znaku '\n' może spowodować pominięcie w strumieniu wejściowym ciągu spacji.

Sekwencja sterująca w funkcji **scanf** ma prostszą postać niż w przypadku funkcji **printf**. Zaczyna się ona znakiem % (procent), po którym mogą występować opcjonalnie znak * (omówiony dalej) i liczba określająca maksymalną szerokość pola wejściowego. Sekwencja kończy się specyfikatorem, którym może być jedna z wymienionych poniżej liter:

**d** ciąg znaków ze strumienia wejściowego, po pominięciu poprzedzających białych znaków, jest interpretowany jako liczba całkowita w zapisie dziesiętnym. Początkiem pola wejściowego jest więc pierwszy czarny znak w strumieniu wejściowym a końcem pierwszy znak nie będący cyfrą dziesiętną. Jeżeli bezpośrednio po znaku % w sekwencji sterującej występuje liczba, to określa ona maksymalną szerokość pola wyjściowego w znakach.

Opracowanie tej sekwencji kończy się sukcesem (tzn. przypisaniem odczytanej danej zmiennej wskazywanej przez odpowiedni argument i kontynuacją działania funkcji **scanf**), jeżeli została odczytana przynajmniej jedna cyfra dziesiętna.

Oto przykłady (w komentarzach podano strumień wejściowy ograniczony znakami | i wyniki działania funkcji **scanf**).

```
scanf("%d",&i);     /* |  100|  */
                    /* i=100    */

scanf("%2d",&i);    /* |  100|  */
                    /* i=10     */

scanf("%d",&i);     /* |.10|    */
                    /* Nie wystąpi przypisanie.      */
                    /* Kolejnym znakiem wczytanym    */
                    /* przez jakąś funkcję wejściową */
                    /* będzie . (kropka)             */
```

**o** wczytanie liczby w reprezentacji ósemkowej (zawierającej cyfry od 0 do 7 w sposób opisany powyżej).

**x** wczytanie w analogiczny sposób liczby w reprezentacji szesnastkowej (zawierającej cyfry od 0 do f).

Cyfry od a do f mogą być także zapisane dużymi literami.

```
scanf("%o%d%x",&o,&d,&x);  /* |   278fF| */
                           /* wartości dziesiętnie:  */
                           /* o=23   d=8   x=255     */
```

Standardowo dla trzech powyższych specyfikacji przyjmuje się, że odpowiedni argument jest wskaźnikiem do zmiennej typu **int**. Odpowiednie specyfikatory mogą jednak być poprzedzone modyfikatorami rozmiaru h (dla zmiennych **short**) i l (dla zmiennych **long** ). Wczytana liczba przed przypisaniem do wskazywanej zmiennej jest poddawana konwersji do odpowiedniego typu: do typu **int** jeżeli nie wystąpił modyfikator, **short** dla modyfikator h oraz do typu **long** jeżeli wystąpił modyfikator l. Jeżeli ciąg cyfr na wejściu reprezentuje liczbę przekraczającą zakres typu **long**, to efekt działania funkcji **scanf** jest różny dla różnych kompilatorów. W Turbo C odpowiedniej zmiennej zostanie przypisana dana o wartości 0xffff (typ **int**) lub 0x7fffffff (typ **long**), a w buforze zostaną nie wczytane cyfry. W kompilatorze Microsoft C odpowiedniej zmiennej zostanie przypisana wartość równa wartości liczby na wejściu modulo 0xffff (dla typu **int**) albo modulo 0xffffffff (dla typu **long**). Na przykład:

```
scanf("%x",&x);     /* |  829fa90|    */
                    /* x=0xFA90       */

scanf("%lx",&x);    /* |   81111119f9f98| */
                    /* TC   x=0x7fffffff  */
                    /* MS C x=0x119f9f98  */
```

**c** zmiennej wskazywanej przez odpowiedni argument jest przypisywany następny znak w strumieniu wejściowym. Dotyczy to także białych znaków, które wyjątkowo dla specyfikatora `c` nie są pomijane.

```
scanf("%c",&c);      /* | a|     c=' '    */
```

**s** wczytanie najbliższego ciągu czarnych znaków do tablicy znakowej wskazywanej przez odpowiedni argument i uzupełnienie go znakiem '\x0', na który także trzeba zarezerwować miejsce w tablicy.

Jeżeli bezpośrednio po znaku % występuje liczba, to określa ona maxymalną szerokość pola wejściowego w znakach. Np.:

```
char tab[10],c;
scanf("%s",tab);      /* |    Adam Sapek  | */
                      /* tab="Adam"         */

scanf("%3s",tab);     /* |    ADAM Sapek  | */
                      /* tab="ADA"          */

scanf("%1s",&c);      /* |    Adam Sapek  | */
                      /* c='A'              */

scanf("%c",&c);       /* |    Adam Sapek  | */
                      /* c=' '              */
```

**[]** dla specyfikatora s przyjęte jest, że pole wejściowe ograniczone jest przez białe znaki. Z tego powodu nie można użyć tego specyfikatora do wczytania np. łańcucha zawierającego odstępy. Do tego służy specyfikator w postaci ciągu znaków ujętych w nawiasy kwadratowe.

Przy użyciu tego specyfikatora można również wczytać ciąg znaków do tablicy wskazywanej przez odpowiedni argument, ale ogranicznikiem pola jest dowolny znak nie wymieniony w nawiasach. Na przykład instrukcja:

```
scanf("%[1234567890]",tab);
```

spowoduje wczytanie do tablicy tab ciągu cyfr.

Jeżeli ciąg znaków w nawiasach kwadratowych rozpoczyna się znakiem ^ to ogranicznikiem pola jest dowolny ze znaków wymienionych w nawiasach. Np.:

```
scanf("%[^\n]",tab);
```

spowoduje wczytanie do tablicy tab ciągu znaków aż do znaku końca wiersza.

Przy użyciu specyfikacji `[]` w przeciwieństwie do użycia specyfikacji `%s` nie są pomijane początkowe białe znaki. Przykładowo:

```
scanf("%[1234567890]",tab);  /* | 2892 | */
                             /* tab=""   */
```

Sytuacja jak w powyższym przykładzie, tzn. niewczytanie żadnego znaku do tablicy, nie może zdarzyć się w przypadku specyfikacji `%s`, która zawsze powoduje wczytanie co najmniej jednego czarnego znaku.

Specyfikacja `[]` może być uzupełniona o liczbę określającą maksymalny rozmiar pola wejściowego. Instrukcja

```
scanf("%10[^\n]",tab);
```

spowoduje wczytanie ciągu do znaku końca wiersza jednak nie dłuższego niż 10 znaków.

Przy użyciu specyfikacji `[]` bardzo ważne jest pamiętanie o fakcie, że nie pomija ona początkowych białych znaków. Często spotykanym błędem jest próba wczytania kilku kolejnych ciągów bez zadbania o pominięcie białych znaków, jak np. w poniższej konstrukcji:

```
scanf("%[^\n]%[^\n]",adres,miejscowość);
```

Powyższe wywołanie funkcji **scanf** ma służyć wczytaniu do odpowiednich tablic adresu i nazwy miejscowości. Ponieważ zarówno w adresie jak i w nazwie miejscowości mogą wystąpić spacje, użyta została specyfikacja `%[^\n]` umożliwiająca wczytanie dowolnego ciągu znaków aż do końca wiersza. Błąd tutaj polega na tym, że po wczytaniu adresu następnym znakiem w ciągu wejściowym będzie znak końca wiersza `'\n',` który jako ogranicznik pola nie zostanie wczytany. Z tego powodu nie zostanie wczytana nazwa miejscowości, gdyż funkcja **scanf** od razu natrafi na ogranicznik pola `'\n'`. Poprawne rozwiązanie może przybrać jedną z poniższych postaci.

```
scanf("%[^\n] %[^\n] ",adres,miejscowość);
```

lub

```
scanf("%[^\n]%*c%[^\n]%*c",adres,miejscowość);
```

Różnica między nimi polega na tym, że spacje w pierwszym wywołaniu funkcji **scanf** spowodują pominięcie nie tylko znaku końca wiersza `'\n',` ale także wszystkich innych białych znaków, w tym ewentualnych spacji poprzedzających nazwę miejscowości. W drugim rozwiązaniu użyto specyfikacji `%*c` powodującej pominięcie następnego znaku w strumieniu wejściowym, a więc w tym przypadku tylko znaku `'\n'.`

Dla porządku należy wspomnieć, że ciąg wewnątrz nawiasów **[]** może zawierać sekwencje typu: x-y. Sekwencja taka jest rozumiana tak jakby zamiast niej występował ciąg znaków od x do y. Powodem, dla którego wspominam o tej użytecznej sekwencji na marginesie, jest to, że w kompilatorach firmy Microsoft nie jest ona poprawnie implementowana. W wersji MS C 5.1 ciąg x-y jest rozumiany dosłownie jako ciąg znaków x, - (minus) i y. Co prawda wersji 6.0 kompilatora sekwencja ta jest rozpoznawana, nie jest ona jednak interpretowana w pełni. Dla przykładu nie jest możliwe utworzenie specyfikacji:

```
%[a-zA-Z]
```

która powinna spowodować wczytanie ciągu małych i dużych liter.

**f** najbliższy ciąg czarnych znaków ze strumienia wejściowego jest interpretowany jako liczba zmiennoprzecinkowa. Format tej liczby może być następujący: znak,

ciąg cyfr z ewentualną kropką dziesiętną, litera `e` lub `E` poprzedzająca wykładnik potęgi (może być ze znakiem).

Interpretacja sekwencji `%f` zakończy się sukcesem, jeżeli ciąg wejściowy będzie składał się z co najmniej jednej cyfry poprzedzonej co najwyżej znakiem.

Specyfikator `f` może być poprzedzony znakiem l, który oznacza, że odpowiedni argument wskazuje na zmienną typu **double,** a nie **float**, jak się domyślnie przyjmuje.

Normalnie ogranicznikiem pola wejściowego jest pierwszy biały znak lub znak, który nie może wystąpić w zapisie liczby zmiennoprzecinkowej. Można jednak zawęzić pole wejściowe, podając jego maksymalną szerokość po znaku % (procent).

```
scanf("%f",&f);      /* |   -1.01e+5  |  */
                     /* f=-101000       */
scanf("%3f",&f);     /* |103.511      |  */
                     /* f=103.0         */
```

Dowolna sekwencja sterująca może zawierać znak * (gwiazdka). Oznacza on, że odpowiednia dana zostanie odczytana ze strumienia wejściowego, ale nie zostanie przypisana żadnej zmiennej. Przykładowo:

```
scanf("%*3d%c",&c);   /* |1234| */
                      /* c='4'  */
scanf("%*[^1234567890]%c",&c);
/* wczytanie pierwszej cyfry ze strumienia  wejściowego*/
```

Funkcja **scanf** zwraca wartość całkowitą równą liczbie wczytanych pól wejściowych lub wartość EOF w przypadku napotkania znaku końca pliku. Wczytanie pól związanych ze specyfikacjami `%s` `%c` i `%[]` zawsze kończy się sukcesem (jeżeli nie zostanie napotkany znak końca pliku). Inne specyfikacje, jak `%d` czy `%f`, wymagają, by w odpowiednim polu wejściowym znajdowały się właściwe znaki. Na przykład, jeżeli pierwszym znakiem w polu wejściowym odpowiadającym specyfikacji `%d` nie będzie cyfra dziesiętna ani żaden ze znaków `+` (plus) lub `-` (minus), to odczytywanie takiego pola zakończy się niepowodzeniem i funkcja **scanf** zakończy działanie, zwracając liczbę wczytanych uprzednio pól.

## 3.2. Standardowe wejście

W poniższym opisie zajmę się standardowym wejściem w sytuacji, gdy jest ono związane z konsolą użytkownika. W przeciwnym przypadku, tzn. gdy standardowe wejście zostało przekierowane, zachowuje się ono dokładnie tak samo jak wejście z pliku.

Gdy standardowe wejście jest związane z konsolą użytkownika, dane dla programu czytającego zeń wpisuje się z klawiatury. W efekcie plik stdin jest często utożsamiany właśnie z klawiaturą. Nie jest to jednak słuszne.

Informacja wprowadzana z klawiatury jest przetwarzana przez system operacyjny. Na czym polega to przetwarzanie? Przede wszystkim, dokonywane jest buforowanie całych wierszy; dlatego też nie można wczytać ze standardowego wejścia mniej niż jednego wiersza. Nie wynika to z konstrukcji funkcji **scanf** (czy innej funkcji czytającej ze standardowego wejścia) - po prostu system udostępnia programowi dane w postaci całych wierszy. Poza tym, w zależności od systemu operacyjnego, użytkownik ma mniejsze lub większe możliwości edycji wiersza zanim zostanie on przekazany programowi przez standardowe wejście. Z tego powodu program nie może wczytać ze standardowego wejścia znaków uznawanych przez system za znaki sterujące (odpowiadających np. klawiszom kursora, HOME, END, INSERT), gdyż nie znajdą się one nigdy w strumieniu wejściowym.

Najważniejszą konsekwencją jest jednak to, że program wykonywalny czytający ze standardowego wejścia może w różnych systemach zachowywać się nieco odmiennie. Można to zaobserwować kompilując poniższy programik i uruchamiając program wykonywalny pod nadzorem systemu MS-DOS i DR DOS 6.0.

```
#include <stdio.h>

void main()
 {
  printf("Naciśnij Ctrl+C aby skończyć\n");
  while(1)
   scanf("%*[^\n]%*c");
 }
```

Program ten wczytuje kolejne wiersze wpisywane przez użytkownika. W systemie DR DOS, który daje dużo większe możliwości edycji wiersza wejściowego, można robić przy tym takie "sztuki" jak przeglądanie poprzednio wpisanych wierszy klawiszami ↑↓, przesuwanie się na początek i koniec wiersza klawiszami HOME i END, przełączanie między trybami wstawiania (insert) a zamazywania (overwrite) przy pomocy klawisza INS (ze zmianą kształtu kursora), a nawet wyszukanie ostatniego wiersza zaczynającego się tak jak wiersz właśnie wpisywany (klawisz CTRL+R). Oczywiście nie robi tego wszystkiego nasz programik, a właśnie system operacyjny. Funkcja **scanf** zaczyna działać dopiero po naciśnięciu przez użytkownika klawisza ENTER. W strumieniu wejściowym otrzymuje ona ciąg znaków, który powstał w wyniku edycji i jest widoczny na ekranie, uzupełniony znakiem końca wiersza `'\n'`.

## 3.3. Funkcja fscanf

Deklaracja:

```
int fscanf(FILE *file,const char *format, ... );
```

Funkcja **fscanf** jest funkcją czytającą dane z pliku. Plik jest identyfikowany przez pierwszy obowiązkowy parametr typu FILE* (typ FILE jest zdefiniowany w pliku

"**stdio.h**"). Działanie funkcji i interpretacja ciągu sterującego format są identyczne jak w funkcji **scanf**.

Do skojarzenia pliku dyskowego ze zmienną służy w języku C standardowa funkcja **fopen**. Funkcja ta umożliwia otwarcie pliku w dwóch różnych trybach: tekstowym i binarnym. Jeżeli plik jest otwarty w trybie tekstowym, to wprowadzenie do niego znaku końca wiersza `'\n'` (znaku o kodzie 10), powoduje w rzeczywistości umieszczenie w pliku pary znaków: `'\r'` (powrót karetki, kod 13) i `'\n'`. Analogicznie, jeżeli w pliku znajduje się taka para znaków, to wczytywany jest znak `'\n'`.

Plik otwarty w trybie binarnym nie posiada tej cechy, wszystkie znaki są do niego zapisywane i odczytywane z niego dosłownie.

Omawiając funkcję **scanf**, czytającą ze standardowego wejścia, wspomniałem, że system operacyjny pozwala zwykle skierować do programu czytającego ze standardowego wejścia dane z pliku. W takim przypadku plik, z którego będzie czytać funkcja **scanf**, będzie się zachowywał jak plik otwarty w trybie tekstowym (a więc będzie wykonywana konwersja pary znaków `'\r' '\n'` na znak `'\n'`).

Istnieje jeszcze jedna, ważniejsza, różnica między plikiem otwartym w trybie tekstowym a plikiem otwartym w trybie binarnym. W pliku tekstowym znak o kodzie 0x1a jest traktowany jako znacznik końca pliku. Ponieważ taki znak zawsze może pojawić się w plikach wykonywalnych, plikach danych i innych plikach nie zawierających normalnego tekstu, pliki takie zawsze powinny być otwierane w trybie binarnym.

Wśród standardowych, predefiniowanych plików znajduje się plik stdin odpowiadający standardowemu wejściu. Wywołanie funkcji fscanf z plikiem stdin jako pierwszym argumentem jest równoważne odpowiedniemu wywołaniu funkcji **scanf**.

Dzięki olbrzymim możliwościom formatowanego przekształcania wejścia funkcja **fscanf** umożliwia realizację w prosty sposób nawet skomplikowanego przetwarzania zawartości plików. Poniżej przedstawię kilka króciutkich programów, które są przykładami wykorzystania możliwości funkcji **fscanf** w przetwarzaniu plików. Zachęcam Czytelnika by przed przeczytaniem dalszego ciągu spróbował napisać te programy samodzielnie.

1. Program wypisujący wszystkie co najmniej pięcioliterowe napisy (ciągi liter i spacji) w dowolnym pliku (także np. pliku **.exe**).
2. Program wypisujący wiersze pliku wejściowego, które nie są komentarzami. Należy przyjąć, że komentarzem jest wiersz zaczynający się od znaku średnika poprzedzonego co najwyżej odstępami.

Program wypisujący wszystkie łańcuchy ujęte w znaki cudzysłowu znajdujące się w pliku tekstowym (np. tekście programu).

Wszystkie programy zostały napisane tak, żeby zasadnicza idea przetwarzania była jak najlepiej widoczna. Programy 2 i 3 są filtrami czytającymi ze standardowego wejścia i piszącymi do standardowego wyjścia. Z tego powodu te dwa programy używają funkcji **scanf** a nie **fscanf**. Używając tych programów należy podmienić standardowe wejście plikiem:

```
rem < tsr16.asm
```

Rozwiązania zadania 1 nie można przedstawić w postaci filtra, ponieważ program ma czytać także pliki binarne (standardowe wejście jest plikiem tekstowym).

**1.**
```
/* plik napisy.c */
#include <stdio.h>
char s[100];                          /* bufor napisu */
void main(int argc,char *argv[])
{
 FILE *in;

 if(argc<1)return;                       /* brak nazwy
pliku */
 in=fopen(argv[1],"rb");     /* otwórz plik o podanej
nazwie */

 while(
        EOF !=
fscanf(in,"%*[^abcdefghijklmnopqrstuvwxyz]"
                          "%[abcdefghijklmnopqrstuvwxyz ]",s)
      )
   if( strlen( s )>4 )   /* jeżeli napis dłuższy niż 4
znaki */
     printf( "%s\n" , s );   /* to wypisz go w osobnej
linii */
}
```

Pętla **while** wykonuje się do chwili, gdy funkcja **scanf** zwróci wartość EOF, czyli dopóki nie zostanie przeczytany cały plik. Funkcja **scanf** pomija znaki do momentu natrafienia na małą literę (aby program wyszukiwał także słowa zawierające duże litery, trzeba je dopisać w nawiasach kwadratowych). Po natrafieniu na małą literę funkcja **scanf** wczytuje do tablicy s ciąg liter. Jeżeli wczytany ciąg jest dłuższy niż 4 znaki, to jest on wypisywany. Cała procedura powtarza się aż do przeczytania całego pliku. Program ten pokazuje, że przy pomocy funkcji **fscanf** można z powodzeniem przetwarzać nawet pliki wykonywalne, które nie są przecież w żaden czytelny sposób sformatowane.

**2.**
```
/* plik rem.c */
#include <stdio.h>

char s[100];                                  /* bufor
linii */
```

```
        void main()
        {
          while(EOF != scanf(" ;%*[^\n]") )
                                /* pomiń linie zaczynajace sie
od ; */
            {
              scanf("%[^\n]%*c",s);         /* wczytaj kolejna
linie */
              printf("%s\n",s);             /* wypisz wczytana
linie */
            }
        }
```

Wywołanie funkcji **scanf** w warunku pętli **while** powoduje najpierw pominięcie początkowych odstępów (spacja na początku ciągu format). Następnie, jeżeli kolejnym znakiem w pliku jest średnik, wykorzystywana jest sekwencja `%*[^\n].` Powoduje ona pominięcie wszystkich znaków do końca wiersza. Proszę zauważyć, że w tym momencie kolejnym wczytywanym znakiem będzie `'\n'.` Z tego właśnie powodu drugie wywołanie funkcji **scanf** w tym obiegu pętli nie wczyta nic do tablicy s i ograniczy się do pominięcia znaku `'\n'` (specyfikacja `%*c` powoduje pominięcie najbliższego znaku). Funkcja **printf** wydrukuje pusty wiersz (żeby nie drukować pustych wierszy, można przed funkcją **printf** postawić warunek `if(*s))`. Jeżeli następny wiersz także zaczyna się od średnika, to zostanie on pominięty w ten sam sposób. Jeżeli któryś wiersz nie będzie zaczynać się średnikiem, to pierwsze wywołanie funkcji **scanf** ograniczy się do pominięcia wiodących odstępów i zostanie przerwane w momencie stwierdzenia niezgodności kolejnego znaku ze średnikiem. W takim przypadku, w drugim wywołaniu funkcji **scanf** zostanie wczytany wiersz, który następnie zostanie wydrukowany w funkcji **printf**. Procedura ta będzie powtarzać się aż do przeczytania całego pliku.

Proszę zwrócić uwagę, że mimo iż pierwsze wywołanie funkcji **scanf**, odpowiedzialne za pomijanie wierszy zaczynających się średnikiem nie jest wykonywane w pętli, program będzie pomijał także komentarze następujące kolejno po sobie. Stanie się tak dlatego, że po opracowaniu specyfikacji `%*[^\n]` nie będzie pominięty znak końca wiersza `'\n'.` Znak ten zostanie odczytany przez drugie wywołanie funkcji **scanf** uniemożliwiając jej przeczytanie wiersza.

**Uwaga!**

Kompilator MS C 6.0 zawiera błąd w funkcji **scanf**, który powoduje, że druga funkcja **scanf** natrafiając od razu na znak końca wiersza nie wpisze na początku tablicy s znaku `'\x0'.` Używając tego kompilatora należy albo na początku pętli zerować pierwszy element tablicy:

```
s[0]='\x0';
```

albo zastąpić funkcję **scanf** funkcją **fscnaf**.

**3.** `/* plik lancuch.c */`

```
        #include <stdio.h>
        char s[100];                                /* bufor
łańcucha */
        void main()
        {
         while( EOF != scanf( "%*[^\"]%*c%[^\"]%*c" , s ) )
                                                    /* wczytaj
łańcuch */
           printf( "\"%s\"\n" , s );                    /* wypisz
go */
        }
```

Program ten jest podobny do programu pierwszego. Łańcuchy formatujące w funkcjach **scanf** i **printf** są trochę nieczytelne, gdyż zawierają znaki cudzysłowu, które muszą tutaj być zapisywane jako `\".` Funkcja **scanf** w tym programie pomija wszystkie znaki aż do napotkania cudzysłowu (specyfikacja `%*[^\"]`), następnie pomija cudzysłów (specyfikacja `%*c`), wczytuje do tablicy s łańcuch do cudzysłowu zamykającego (specyfikacja `%[^\"]`) i w końcu pomija cudzysłów zamykający (specyfikacja `%*c`). Funkcja **printf** wypisuje wczytany łańcuch, otaczając go cudzysłowami. Cała ta procedura jest wykonywana tak długo, aż zostanie przeczytany cały plik.

Jak widać na przykładzie tych programów, nawet jedno czy dwa wywołania funkcji **fscanf** lub **scanf** umożliwiają wykonywanie dosyć skomplikowanych operacji na plikach, i to zarówno tekstowych jak i binarnych.

Zdaję sobie sprawę, że sugestie autora, żeby samemu napisać program, który znajduje się na następnej stronie, są zwykle ignorowane. Dlatego zakładam, że przeciętny Czytelnik nie próbował napisać tych programików sam. Żeby jednak opanować potężne możliwości funkcji **scanf** i wyjść w użyciu poza banalne `%d` trzeba samemu napisać kilka programów. Dlatego zachęcam do wymyślenia sobie kilku zadań i zrealizowania ich przy pomocy funkcji **scanf**.

Na marginesie: kiedy znajomi "pascalowcy" po pierwszym kontakcie z językiem C narzekali, jaki to wielki kod ten C daje, odpowiadałem im mniej więcej tak: jeżeli wlinkowujesz (przepraszam za to słowo) takie potężne narzędzia jak scanf i printf, tylko po to, żeby program wczytał twoje imię i wypisał:

```
Cześć Janek
```

to nie dziw się, że masz przerośnięty kod.

W rozdziale "Kod wynikowy" pokażę, że język C może dawać kod rewelacyjnie mały.

## 3.4. Funkcja sscanf

Deklaracja:

```
int sscanf(char *string, const char *format, ... );
```

Funkcja **sscanf** jest krewniaczką funkcji **scanf**, czytającą dane z łańcucha zawartego w pamięci operacyjnej a nie z pliku. Funkcja ta przyjmuje takie same sekwencje sterujące w ciągu format, i w taki sam sposób przekształca dane wejściowe. Pewną różnicą, wynikającą z charakteru wejścia, jest to, że funkcja **sscanf** zwraca wartość EOF po próbie odczytania znaku `'\x0'`. Dlatego uważam, że lepiej mówić o funkcji **sscanf** jako o funkcji przekształcającej łańcuchy (łańcuch w języku C to ciąg znaków zakończony znakiem `'\x0'`), a nie zawartość pamięci.

Funkcji **sscanf** używa się oczywiście dużo rzadziej niż funkcji **fscanf**. Przenosząc bezpośrednio nawyki nabyte przy użyciu funkcji **fscanf** na konstrukcje z funkcją **sscanf** można łatwo popełnić błąd. Proszę przyjrzeć się następującemu fragmentowi i spróbować wskazać, na czym polega błąd:

```
char *s="42 24 5322 424";

while( EOF != sscanf(s,"%d",&x) )
  printf("%d",x);
```

Oczywiście, zapisaliśmy pętlę nieskończoną. Funkcja **sscanf** będzie w kółko odczytywać z ciągu liczbę 42. Analogiczna pętla czytająca dane z pliku za pomocą funkcji **fscanf** działałaby zgodnie z oczekiwaniami gdyż położenie wskaźnika plikowego zmienia się po każdym odczycie (plik "przewija się"). Łańcuchy oczywiście się nie "przewijają". Co więcej, funkcja **sscanf** w żaden sposób nie ułatwia ręcznego przesunięcia wskaźnika. Jako wartość zwraca ona (jak inne funkcje rodziny **scanf**) liczbę wczytanych pól wejściowych, nie informując w żaden sposób, ile znaków wczytała z łańcucha. Dlatego należy przyjąć, że funkcja **sscanf** nie nadaje się do przetwarzania łańcuchów w sposób iteracyjny.

Zastosowania funkcji **sscanf** są bardzo podobne zastosowań specjalizowanych funkcji do przetwarzania łańcuchów, zadeklarowanych w pliku "**string.h**". W wielu przypadkach funkcja **sscanf** (a także sprintf) pozwala uzyskać odpowiednie przekształcenie w prostszy sposób.

Funkcje formatowanego przetwarzania łańcuchów dobrze nadają się do tworzenia i przekształcania nazw plików. Oto przykłady.

**1.**
```
sscanf( name , "%[^.]", name);
sprintf( name , "%s.wkz", name );
```

Zmienna name wskazująca na nazwę pliku zostanie przekształcona w taki sposób, że nowa nazwa będzie miała rozszerzenie **.wkz** (niezależnie od tego, czy w nazwie wyjściowej było jakieś rozszerzenie czy nie).

**Uwaga!**

Funkcja **sprintf** z kompilatora Turbo C 2.0 "gubi się", jeżeli kazać jej przepisać jakiś łańcuch do samego siebie. Dlatego powyższy przykład nie będzie działał poprawnie po skompilowaniu tym kompilatorem.

2. `sprintf(full_name,"%s%s",path,name);`

Powyższe wywołanie funkcji **sprintf** umieszcza w łańcuchu full_name pełną nazwę pliku utworzoną ze ścieżki, wskazywanej przez path, i nazwy pliku, wskazywanej przez name.

## 3.5. Funkcja cscanf

Deklaracja:

```
int cscanf(const char *format, ... );
```

Funkcja **cscanf** nie należy do funkcji standardu ANSI. Jest to funkcja wejścia z konsoli. Podczas gdy funkcja **scanf** nie odczytuje bezpośrednio klawiatury i korzysta przy tym z usług systemu operacyjnego, funkcja **cscanf** sama interpretuje dane wprowadzane z klawiatury. Większość znaków traktowanych przez system jako znaki sterujące (klawisze kursora, INS, HOME), funkcja **cscanf** wczytuje "dosłownie". Jak z tego wynika, nie daje ona takich możliwości edycji wiersza, jakie oferuje system operacyjny w przypadku użycia funkcji **scanf**. Drugą ważną różnicą pomiędzy funkcjami **scanf** i **cscanf** jest to, że wiersz zakończony klawiszem ENTER jest w przypadku funkcji **cscanf** uzupełniany znakiem `'\r'` (powrót karetki), a nie `'\n'` jak w przypadku funkcji **scanf**. Konsekwencje tego są dwojakie. Po pierwsze, kursor na ekranie nie przechodzi do nowego wiersza a tylko cofa się na początek wiersza bieżącego. Po drugie, żeby wczytać przy pomocy funkcji **cscanf** cały wiersz należy użyć specyfikacji `%[^\r]` a nie `%[^\n]` jak w przypadku funkcji **scanf**.

## 3.6. Funkcje vscanf, vfscanf i vsscanf

Deklaracje:

```
int vscanf(const char *format, va_list arg );
int vfscanf(FILE *plik, const char *format, va_list arg
);
int vsscanf(char *string, const char *foramt, va_list
arg );
```

Funkcje te nie są ujęte w standardzie ANSI. Jest to o tyle dziwne, że analogiczne funkcje w rodzinie printf są funkcjami standardowymi. Ich implementacje dołącza do swoich kompilatorów firma Borland.

Nie są to funkcje o zmiennej liczbie argumentów w odróżnieniu od pozostałych w rodzinie scanf. Trzeci argument, będący typu va_list, służy do wskazania ciągu

argumentów funkcji o zmiennej liczbie argumentów. Funkcje te wykorzystuje się podobnie jak odpowiadające im funkcje z rodziny printf (patrz pkt. III.2.4), jednak zakres zastosowań, także ze względu na ich niestandardowość, jest dużo mniejszy. Mogą one być używane do pisania specyficznych funkcji wejściowych, wykorzystujących możliwości formatujące funkcji **scanf**.

# 4. Zastosowania funkcji formatowanego wejścia/wyjścia.

Klasycznym zastosowaniem, można nawet powiedzieć przeznaczeniem, funkcji formatowanego wejścia/wyjścia była komunikacja z użytkownikiem. Dzisiaj, gdy nowoczesne programy komunikują się z użytkownikiem raczej przez kolorowe okienka niż przez standardowe pliki we/wy, to pole zastosowań funkcje **scanf** i printf już utraciły. W obecnych aplikacjach interfejs użytkownika albo pisze się samemu na niskim poziomie, albo też tworzy się go przy pomocy specjalizowanych funkcji, czy to dołączonych do używanego kompilatora, czy też wyprodukowanych przez firmy trzecie.

Funkcje **scanf** i **printf** są wciąż używane w programach narzędziowych, szczególnie tych projektowanych z myślą o wykorzystaniu mechanizmu potoków. Program taki czyta dane ze standardowego wejścia przy pomocy funkcji **scanf** i zapisuje wyniki do standardowego wyjścia przy pomocy funkcji **printf**. System operacyjny umożliwia skierowanie danych wyjściowych jednego programu na wejście innego.

W programowaniu bardziej wyrafinowanego interfejsu użytkownika przydatna jest niestandardowa funkcja **cprintf**. Szczególnie jej wersja implementowana w kompilatorach firmy Borland posiada szereg cech sprawiających, że funkcja ta doskonale nadaje się do tego celu. Najważniejszą z nich jest możliwość wyznaczenia na ekranie okienka, do którego funkcja **cprintf** będzie wyprowadzała swoje argumenty. Okienko takie lokalnie zachowuje się jak ekran: może być wyczyszczone niezależnie od reszty ekranu, ma własny układ współrzędnych, a po wpisaniu znaku nowego wiersza w ostatnim jego wierszu cała zawartość okienka przewija się do góry. Te cechy sprawiają, że funkcja **cscanf** często służy do organizowania ekranów w programach.

## 4.1. Projektowanie wyglądu ekranu

Zaprojektowanie wyglądu ekranu i napisanie odpowiedniego kodu jest pracą dość żmudną. Istnieją nawet programy automatycznie generujące kod w języku C na podstawie ekranu utworzonego w specjalnym edytorze. Chciałbym tutaj przedstawić techniczną sztuczkę, umożliwiającą bardzo szybkie i łatwe napisanie kodu generującego dowolnie skomplikowany ekran.

Sednem pomysłu jest użycie tylko jednej funkcji **cprintf**, wypisującej długi ciąg, który pokryje cały ekran. W praktyce można to zrealizować następująco.

Przy pomocy jakiegoś edytora projektujemy ekran. Idealnym narzędziem do tego celu jest Word Perfect, po pierwsze dlatego, że widać w nim prawie cały ekran, po drugie zaś ponieważ posiada on dobre narzędzia do rysowania wszelkich ramek (funkcja **Line Draw**). Pamiętać tylko należy, żeby tekst wyeksportować w postaci ASCII.

Gdy mamy już zaprojektowany wygląd ekranu, trzeba go zmienić w łańcuch, który można by umieścić w wywołaniu funkcji **cprintf**. Aby to uzyskać, dopisujemy cudzysłowy na początku i na końcu każdego wiersza zaprojektowanego ekranu. Wykorzystujemy tutaj fakt, że kompilator języka C traktuje sąsiadujące ze sobą literały łańcuchowe jak jeden łańcuch. Na przykład napis:

```
"Adam"  " "  "Sapek"
```

jest równoważny napisowi:

```
"Adam Sapek"
```

Word Perfect umożliwia rysowanie linii dowolnym znakiem, a więc żeby dopisać cudzysłowy wystarczy "narysować" dwie pionowe linie znakiem " . Trzeba pamiętać, żeby każdy wiersz między cudzysłowami miał długość 80 znaków (szerokość ekranu). Teraz jeszcze na samym początku dopisujemy `cprintf(` a na końcu zamykający nawias `);` ... i mamy gotowe wywołanie funkcji **cprintf** budujące od razu cały ekran. Będzie ono wyglądało na przykład tak:

```
clrscr();
cprintf(
" Nazwisko: ................. Imiona: 1. ............. 2. ..............      "
" Imię ojca: ................., imię matki ..............................      "
" Miejsce urodzenia: ......................... data ......................      "
"                                                    dzień, m-c, rok          "
"                                                                             "
" ▒▒▒▒▒▒▒▒▒▒▒                      lub                       ▒▒-▒▒-▒▒-▒      "
" Nr PESEL                                                   Nr id. NIP       "
"                                                                             "
" Miejsce zamieszkania: ...................................................      "
" Gmina/Dzielnica: ........................................................      "
" Ulica: ............................ Nr domu ...... nr mieszkania .......      "
" Kod pocztowy ▒▒-▒▒▒  miejscowość  .....................................      "
"                                                                             "
"                                 DEKLARACJA                                  "
"                na zaliczkę miesięczną na podatek dochodowy                  "
"                                                                             "
"                                                                             "
"                                                                             "
```

```
"                                                                    "
"                                                                    "
"                                                                    "

" ┌──────────────────┬──────────────┬──────────────────┬──────────────┐    "
" | _ródła przychodów |   Przychód   | Koszty uzyskania |    Dochód    |    "
" |                   |     w zł     |       w zł       |     w zł     |    "
" ├══════════════════┼══════════════┼══════════════════┼══════════════┤    "
" |                   |              |                  |              |    "
" |                   |              |                  |              |    "
" |                   |              |                  |              |    "
" |                   |              |                  |              |    "
" |                   |              |                  |              |  ");
```

Jeżeli tekstem ma być pokryty cały ekran, tzn. jeżeli zawiera on pełne 25 wierszy, to ostatni musi być krótszy niż 80 znaków inaczej ekran przewinie się o jeden wiersz w górę.

Na pierwszy rzut oka może wydawać się, że w ten sposób zajmuje się ogromny obszar pamięci. W rzeczywistości, jeżeli ekran jest skomplikowany, taki zapis jest nie tylko czytelniejszy, ale daje także krótszy kod wynikowy; ciąg pokrywający cały ekran zajmuje przecież tylko 2000 bajtów.

Ponieważ używamy funkcji **cprintf**, nie musimy ograniczać się do stałego ciągu znaków. Gdy ekran jest już zaprojektowany, można w odpowiednie miejsca wstawić sekwencje sterujące wypisujące zawartość jakichś zmiennych podanych jako kolejne argumenty funkcji **cprintf**. W jednym z programów rozwiązałem w ten sposób częściowo problem wyświetlania na ekranie polskich liter w różnych standardach. Wszystkie wystąpienia polskich liter w ciągu wypisywanym przez funkcję **cprintf** wystarczyło zamienić na `%c` a następnie dodać do wywołania funkcji odpowiednią ilość argumentów. Argumenty te były elementami tablicy indeksowanej numerem wybranego przez użytkownika standardu i reprezentowały kody odpowiednich polskich znaków w tym standardzie.

## 4.2. Reprezentacja rekordów w pliku

Prawie każdy program korzysta z jakiejś swojej bazy danych. Baza taka składa się zwykle z rekordów zawierających np. dane o pracownikach, klientach itp. Informacje takie są w programie reprezentowane w postaci zmiennych lub tablic typu strukturalnego. Przykładowo, dane osobowe możemy reprezentować strukturą:

```
struct osoba {
                char nazwisko[30],
                imie1[20],
                imie2[20],
                imie_ojca[20],
```

```
    imie_matki[20],
    miejsce_ur[40],
    data_ur[11],
    ulica[30],
    nr_domu[6],
    nr_mieszkania[6],
    kod[7],
    miejscowosc[40];
}podatnik;
```

Najprostszym sposobem reprezentacji rekordów w pliku jest zapisywanie bloków o długości równej wielkości struktury. Rozwiązanie takie jest proste i ma swoje zalety. Ponieważ wszystkie rekordy zajmują tyle samo miejsca można łatwo wyznaczyć położenie dowolnego rekordu względem początku pliku. Z tego samego powodu bardzo łatwe jest uaktualnienie zawartości wybranego rekordu lub wpisanie na jego miejsce innego. Główną wadą tego rozwiązania jest zajmowanie dużej ilości miejsca na dysku. Objawia się to szczególnie w przypadku "rzadkich" struktur, w których większość rekordów zawiera wartości przypisane tylko w niektórych polach, a także struktur o dużej liczbie pól łańcuchowych. Dla przykładu, na nazwę miasta trzeba zarezerwować co najmniej dwadzieścia parę bajtów, na wypadek gdyby ktoś chciał wpisać np. STARGARD SZCZECIŃSKI, lecz większość rekordów będzie zapewne zawierała nazwy kilkubajtowe. Drugą poważną wadą tego rozwiązania jest niemożliwość dokonywanie w przyszłości zmian długości poszczególnych pól[5]. Ogranicza to rozbudowywalność programu.

Innym rozwiązaniem jest zapisywanie każdego pola rekordu w postaci osobnego wiersza tekstu. W ten sposób puste pola zajmują w pliku tylko dwa bajty (jako znak końca wiersza `'\r' '\n'`) a pozostałe pola tyle bajtów, ile wynosi ich rzeczywista długość (plus dwa bajty na koniec wiersza). Metoda ta nadaje się szczególnie do obsługi struktur z dużą ilością pól tekstowych, czyli wtedy, gdy zawodzi sposób poprzedni, ma ona jednak swoje wady. Ponieważ w zależności od swojej zawartości rekordy mogą zajmować w pliku różną ilość miejsca, nie można zlokalizować rekordu inaczej, niż przeszukując plik. Przy zmianie zawartości jakiegoś rekordu należy przepisać cały plik. Ta ostatnia cecha wydaje się na pierwszy rzut oka bardzo poważnym utrudnieniem; proszę jednak zwrócić uwagę, że "poważne" programy, które zostawiają zwykle jedną kopię poprzedniej wersji pliku (**.bak**) i tak muszą przy tej okazji dokonywać takiego przepisywania.

Do obsługi takiej reprezentacji rekordu w pliku idealnie nadają się funkcje **fscanf** i fprintf. Funkcje **save** i **load**, realizujące odpowiednio operacje zapisania i odczytania jednego rekordu z pliku, mogą wyglądać mniej więcej tak:

```
save(FILE *plik,struct osoba *p)
```

[5] Zmiana taka uniemożliwi odczytywanie uprzednio zapisanych plików danych (poza tym efektem, oczywiście, program będzie nadal działał ...)

```
     {
      if(!plik)return 0;

fprintf(plik,"%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n",
                       p->nazwisko,
                       p->imie1,
                       p->imie2,
                       p->imie_ojca,
                       p->imie_matki,
                       p->miejsce_ur,
                       p->data_ur,
                       p->ulica,
                       p->nr_domu,
                       p->nr_mieszkania,
                       p->kod,
                       p->miejscowosc);
      return 1;
     }

     load(FILE *plik,struct osoba *p)
     {
      if(!plik)return 0;

fscanf(plik,"%[\n]%*c%[\n]%*c%[\n]%*c%[\n]%*c%[\n]%*c%[\n]%*c"

"%[\n]%*c%[\n]%*c%[\n]%*c%[\n]%*c%[\n]%*c%[\n]%*c",
                       p->nazwisko,
                       p->imie1,
                       p->imie2,
                       p->imie_ojca,
                       p->imie_matki,
                       p->miejsce_ur,
                       p->data_ur,
                       p->ulica,
                       p->nr_domu,
                       p->nr_mieszkania,
                       p->kod,
                       p->miejscowosc);
      return 1;
     }
```

Uważny Czytelnik spostrzegł z pewnością, że znak końca wiersza `'\n'` służy właściwie tylko do oddzielenia pól rekordów i może być zastąpiony dowolnym innym znakiem, który nie wystąpi w jakimś polu. Wybór znaku `'\n'` do rozdzielania pól jest jednak o tyle wygodny, że powstały w ten sposób plik można łatwo przeglądać i zmieniać przy pomocy dowolnego edytora tekstu.