Wstęp

Niniejszy rękopis stanowi treść wykładu, który po raz pierwszy wygłosiłem dla studentów II roku astronomii w semestrze zimowym roku akademickiego 2001/2002. Jak sam tytuł wskazuje wykład dotyczył programowania w języku C. I tu od razu wyjaśnienie. Autor nie posiada wykształcenia informatycznego, chociaż zna język C i pisze (według niektórych poprawne) programy w tym języku. Jednak fakt ten może być przyczyną wiele niejasności, a być może także błedów w tekście wykładu. Czytelnik przyjmuje nasię odpowiedzialność za jakiekolwiek szkody wywołane zastosowaniem opisanych tutaj własności języka C.

Wykład nie zawiera wstępu do maszyn cyfrowych, zarządzania pamięcią, oraz obsługi systemu operacyjnego. Autor zakłada, że czytelnik jest już zaznajomiony z tymi zagadnieniami.

Rękopis wciąż wymaga uzupełnień. Pierwsza wersja niniejszego rękopisu została zakończona dnia 29 stycznia 2002 roku, druga wersja – dnia 13 grudnia 2002.

Uwaga: Rękopis ten stanowi własność autora i w żadnym razie nie może być rozpowszechniany bez jego uprzedniej wiedzy i zgody! Pogwałcenie tej zasady będzie ścigane kopniakiem w tyłek, a być może nawet bardziej dotkliwymi represjami.

Implementacja języka

Istnieje wiele mniej lub bardziej podobnych do siebie kompilatorów języka C. Wykład poniższy stanowi opis języka C zaimplentowanego w kompilatorze pakietu `djgpp`, który jest przeniesieniem kompilatora GNU `gcc` do systemu operacyjnego MS-DOS (z dostępem do pamięci w trybie chronionym).

Literatura

Przy tworzeniu niniejszej pracy korzystano z następujących książek:

– Kernighan B.W., Ritchie D.M, 'Język C', WNT, Warszawa 1988,

– Bielecki J., 'Język C: interpretacja standardu', WNT Warszawa 1988,

– Lippman S.B., 'Podstawy języka C++', WNT, Warszawa 1994,

oraz dokumentacji pakietu `djgpp`.

Komputer

Komputer składa się, najogólniej pisząc, z procesora CPU (wykonującego operacje i obliczenia), pamięci operacyjnej RAM (w której przechowywane są dane) oraz urządzeń do komunikowania sią z użytkownikiem, urządzeń wejścia INP (np. klawiatura) i wyjścia OUT (np. monitor).

System dwójkowy

Naturalnym układem pozycjnym, za pomocą którego pracuje komputer jest układ dwójkowy zapisu liczb. Dane również przechowywane są w takiej postaci. Poszczególne bity (cyfry dwójkowe: 0 lub 1) grupuje się w bajty (8 bitów), kilobajty, megabajty itd.

Układ pozycyjny o podstawie p zapisu liczby, to sposób zapisu liczb za pomocą cyfr (jest ich p), w którym wartość liczby zależy zarówno od samej cyfry, jak i jej pozycji w zapisie liczby.

Niech $\{c_i\}$, $i = 0, \ldots, n$ są cyframi danego układu pozycyjnego o podstawie p. Wtedy zapis $c_n c_{n-1} ... c_1 c_0$ oznacza liczbę o wartości $\sum_{i=0}^{n} c_i\, p^i$. Podstawę będziemy zapisywać jako indeks przy zapisie liczby, np.

$$110_2 = 0 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 = 6_{10} = 11_5$$

Inne dwa układy pozycyjne, wykorzystywane w pracy z komputerem, to układ ósemkowy i szesnastkowy. Podstawy obu tych systemów są potęgą podstawy układu dwójkowego, $8 = 2^3$, $16 = 8^2 = 2^4$.

Algorytm

Algorytm to przepis na wykonanie pewnego zadania, podzielony na poszczególne kroki (operacje na obiektach).

Przykłady: przepis kulinarny (bierzemy..., dodajemy..., zagniatamy..., pieczemy..., pożeramy.), zagadnienie znalezienia liczby najmniejszej spośród zadanych liczb, algorytmy sortujące, algorytmy kompresji danych itd...

Zagadanienie → algorytm → (język programowania) → program źródłowy → (kompilator) → program wykonywalny → wynik.

Historia języka C

Język programowania C został zaprojektowany w 1972 roku prze Dennisa Ritchie'go, pracownika Bell Laboratories. Język ten wywodzi się od Algol'a 60 (z roku 1960), CPL (1963), BCPL (1963) i B (1970). Pierwotnie język C powstał na potrzeby systemu operacyjnego UNIX i dlatego zwykle uznawany jest za język programowania systemowego, mimo iż jest on w istocie językiem ogólnego przeznaczenia.

Rosnące zainteresowanie językiem C spowodowało, że w 1983 roku komitet ANSI powołał komisję do spraw standardyzacji języka. Ostatecznie pod koniec 1988 roku powstało opracowanie tzw. standardu ANSI C. W skład tego standardu wchodzi zarówno dokładne określenie samego języka C, jak i definicja standardowych bibliotek funkcji towarzyszących kompilatorowi języka C.

Wprowadzenie: pierwszy program

Na początek prosty program obrazujący wszystkie podstawowe elementy języka C.

```
/* FtoC converter */
;
#include<stdio.h>
;
void main(void)
 {
 float tempC, tempF ;
 printf(" temperature in F: ");
 scanf("%f", &tempF);
 tempC=(tempF-32.0)*(5.0/9.0) ;
 printf(" temperature in C: %6.1f\n", tempC);
 }
```

Program w języku C zbudowany jest ze ***zmiennych*** i ***funkcji***. Ponadto mamy stałe, operatory i instrukcje.

```
#include<stdio.h>
;
void main(void)
 {
 float tC, tF ;
 float tFbeg, tFend, tFstep ;
 ;
 tFstep=10 ;  /* step */
 ;
 printf(" Temperature in F:\n");
 printf("  from = "); scanf("%f", &tFbeg);
 printf("  to   = "); scanf("%f", &tFend);
 printf("\n      F      C\n");
 ;
 tF=tFbeg ;
 while(tF<=tFend)
   {
   tC=(tF-32.0)*(5.0/9.0) ;
   printf("  %6.1f %6.1f\n", tF, tC);
   tF+=tFstep ;
   }
 }
```

Oto przykładowy wynik działania powyższego programu:

```
Temperature in F:
 from = 0
 to   = 50
     F      C
    0.0  -17.8
   10.0  -12.2
   20.0   -6.7
   30.0   -1.1
   40.0    4.4
   50.0   10.0
```

Funkcja `scanf` pozwala na pobranie danych z domyślnego urządzenie wejścia (jest nim klawiatura), natomiast funkcja `printf` służy do wysłania danych do domyślnego urządzenia wyjścia, czyli na monitor. Odpowiedniki plikowe tych funkcji nazywają się `fscanf` oraz `fprintf` (urządzeniami wejścia–wyjścia są wtedy pliki), odpowiedniki napisowe to `sscanf` oraz `sprintf` (dane pobierane są z lub zapisywane do napisów tekstowych). Prototypy tych funkcji podane są w ***pliku nagłówkowym*** `stdio.h` a do naszego programu dołączone zostały za pomocą dyrektywy ***preprocesora*** `#include`.

Tekst programu, zwany także jego wersją źródłową, przechowywany jest w pliku, którego nazwa zwykle zakończona jest przyrostkiem (w systemie MS-DOS zwanym także 'rozszerzeniem' nazwy pliku) `.c`. Niech nasz plik źródłowy nazywa się `test.c`. W kolejnym kroku należy go skompilować za pomocą polecenia:

```
gcc test.c
```

Jednym z zadań kompilatora jest sprawdzenie tekstu pod względem jego poprawności. Najczęściej spotyka się dwa rodzaje błędów: (a) błędy składniowe, które powstają podczas naruszania zasad gramatycznych języka, oraz (b) błędy typu, polegające na niezgodności typów np. argumentów przekazywanych funkcjom.

Poprawny pod względem formalnym tekst programu kompilator tłumaczy zazwyczaj na moduł w języku maszynowym komputera. Czynność ta nazywa się generowaniem kodu wynikowego. Wynikiem pomyślnej kompilacji (oraz łaczenia) jest plik wykonywalny, któremu automatycznie nadana zostaje nazwa `a.exe` lub `a.out` (pod UNIX'em). Program `a.exe` można uruchomić i w ten sposób wykonane zostaną czynności zapisane w pliku `test.c`. Plikowi wykonywalnemu można nadać inną niż domyślna nazwę, np. `name`, za pomocą opcji `-o name`, tak jak poniżej.

```
gcc -Wall -o test.exe test.c
```

Ponadto, opcja `-Wall` wypisuje wszystkie ostrzeżenia kompilatora.

Tekst źródłowy zapisany w języku C++ zwykle przechowuje się w plikach o nazwach zakończonych na `.cpp` lub `.cxx`. Do ich kompilacji używa się programu `gxx` z pakietu `djgpp`. Jego wywołanie może wyglądać następująco:

```
gxx -Wall -o test.exe test.cpp
```

Alfabet języka C

W języku C używa się w zasadzie wszystkich sensownych znaków kodu ASCII. Więcej o tym w trakcie wykładu.

Jednostki składniowe

W języku C wyróżnia się 6 rodzajów jednostek leksykalnych: słowa kluczowe, identyfikatory, stałe, napisy (pseudostałe napisowe, tekstowe), operatory oraz separatory.

Kompilator ignoruje znaki 'białe', tj. spację, znak tabulacji i nowego wiersza, chyba że rozdzielają jednostki leksykalne. Dowolny ciąg takich znaków (oraz komentarzy) jest traktowany tak jak pojedynczy znak spacji i dalej ogólnie będzie nazywany odstępem.

Niekiedy co najmniej jeden znak 'biały' jest konieczny do rozdzielenia sąsiadujących identyfikatorów, słów kluczowych lub stałych.

Zatem każdy program zapisany w języku C składa się z jednostek leksykalnych oraz odstępów.

Jeżeli rozbiór składniowy tekstu programu przeprowadzono do pewnego znaku włącznie, to za następną jednostkę leksykalną uważa się najdłuższy ciąg znaków, z których można utworzyć jednostkę.

`x---y--` rozkłada się na `x -- - y --`

Odstępy przeważnie są pomijane, ale mają oczywiście wpływ na interpretację programu. Na przykład wiersz

```
var=index+++len;
```

jest równoważny

```
var = index ++ + len ;
```

i jest różny od

```
var=index+ ++len;
```

który rozkłada się na

```
var = index + ++ len ;
```

W pierwszym przypadku operator `++` dotyczy zmiennej `index`, a w drugim zmiennej `len`.

Słowa kluczowe

Stanowią one niewielki zestaw słów, których znaczenie jest ściśle określone, np. `for`, `float`. Słów tych nie można używać do nazywania własnych obiektów takich jak zmienne, czy funkcje. Poniżej podany jest pełny zestaw słów kluczowych w rozszerzonej wersji języka C/C++.

```
asm auto break case catch char class const
continue default delete do double else enum
extern float for friend goto if inline int
long new operator private protected public
register return short signed sizeof static
struct switch template this throw try typedef
union unsigned virtual void volatile while
```

Wszystkie słowa kluczowe pisze się małymi literami. Zatem np. słowa `Char`, `CHAR` itd. mogą zostać użyte jako identyfikatory obiektów własnych.

Komentarze

Komentarz rozpoczyna się znakami `/*`, a kończy `*/`. Mogą one wystąpić w dowolnym miejscu programu źródłowego i w zasadzie nie można ich zagnieżdżać.

W nowszych kompilatorach C oraz C++ część programu do końca lini można wykomentować za pomocą znaków `//`.

```
/*
   int i ;
*/ int j ;

int k ; // licznik
```

Identyfikatory, nazwy

Obiekty własne, takie jak stałe symboliczne, zmienne i funkcje identyfikowane są w programie źródłowym poprzez ***nazwy***.

Nazwa może składać się z liter (`a`, ..., `z`, `A`, ..., `Z`), cyfr (`0`, ..., `9`,) oraz znaku podkreślenia (`_`), traktowanego jak litera.

Nazwa musi zaczynać się od litery.

Rozróżnia się małe i duże litery. Zatem `temp`, `Temp`, `TEMP` są nazwami różnych obiektów.

Nazwa (wewnętrzna) może być dowolnej długości, jednak tylko 31 (lub inna ilość) początkowych znaków ma znaczenie.

Wyrażenia i instrukcje

Odpowiednikiem określonej czynności jest w języku C ***wyrażenie***.

Instrukcje są najmniejszymi samodzielnymi jednostkami wykonywalnymi programu. W języku C są one wyrażeniami zakończonymi średnikiem (`;`). Średnik nie jest traktowany jako separator, lecz jako znacznik końca instrukcji.

Najprostszą instrukcją jest instrukcja pusta.

```
; // instrukcja pusta
```

Budowa programu i grupowanie

Program źródłowy w języku C składa się z kilku ***funkcji*** (i zmiennych). Nazwa `main` zarezerwowana jest dla funkcji, od której rozpoczyna się działanie programu wykonywalnego (wynikowego). Każdy program musi taką funkcję zawierać.

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

Do ***grupowania*** instrukcji używa się nawiasów klamrowych, `{}`.

Typy danych

Typ danej określa jej rozmiar oraz sposób kodowania. W języku C wyróżnia się 4 podstawowe, zdefiniowane pierwotnie typy danych:

`char` – jeden znak, czyli jeden bajt.

`int` – typ całkowity; jego rozmiar zależy od maszyny; zwykle są to 2 albo 4 bajty.

`float` – typ wymierny, zmiennoprzecinkowy pojedynczej precyzji (4 bajty).

`double` – typ wymierny, zmiennoprzecinkowy podwójnej precyzji (8 bajtów).

Typy `char` oraz `int` służą do przechowywania liczb całkowitych, natomiast typy `float` oraz `double` – ogólnie liczb rzeczywistych wymiernych.

Za pomocą słów `short`, `long`, `signed` oraz `unsigned` można modyfikować typ całkowity. Dwa pierwsze słowa pozwalają zmienić rozmiar typu, natomiast `unsigned` odnosi się do typu całkowitego z liczbami nieujemnymi.

Jeśli typ `int` zajmuje 4 bajty (jest tak w `djgpp`), to typ `long int` zajmuje również 4 bajty (ponieważ jest to maksymalny rozmiar typu całkowitego w języku C), natomiast typ `short int` zajmuje 2 bajty.

Ogólna zasada jest taka, że rozmiar typu `short int` nie jest większy od rozmiaru `int`, a ten z kolei nie jest większy od rozmiaru `long int`

```
short int <= int <= long int
```

Różnica między typami całkowitymi ze znakiem (`signed`) a typami bez znaku (`unsigned`) polega na odmiennej interpretacji pierwszego od lewej, najbardziej znaczącego bitu w ciągu bitów reprezentującym liczbę. Dla liczby ze znakiem (`signed`) bit ten traktowany jest jako znak liczby: `1` oznacza liczbę ujemną, `0` dodatnią. Obiekty typu `signed char` mogą zatem przyjmować wartości z przedziału od -128 do 127, natomiast obiekty typu `unsigned char` – z przedziału od 0 do 255.

```
short int
long int
unsigned char
```

```
unsigned int
unsigned short int
unsigned long int
```

W powyższych przykładach słowo `int` może być pominięte.

Rozmiar poszczególnych typów danych zależy od maszyny.

Wprowadzony został również typ `long double` przeznaczony dla liczb zmiennopozycyjnych o rozszerzonej precyzji.

Wartości stałe, literały

Literałami nazywamy dane zapisane w sposób dosłowny, które nie mogą się zmieniać podczas wykonywania programu, np. `1`, `23.123`.

Każda stała jest pewnego typu, a więc zajmuje określony rozmiar pamięci.

Stałe są nieadresowalne, ponieważ nie są dostępne adresy miejsc w pamięci przydzielonych tym wartościom.

Stałe całkowite

Są zarezerwowane dla liczb całkowitych, np. `100`, `32L`.

Stałe ósemkowe (zapisane w ósemkowym systemie pozycyjnym) oznacza się początkowym zerem (`0`), np. `010` (= `8`).

Stałe szesnastkowe oznacza się początkowymi znakami `0x` lub `0X`, np. `0xa` (= `10`), `0Xabc`.

Stałe typu `long int` oznacza się literą `L` lub `l`, np. `11L`, `011L`, `0x011L`.

Stałe typu `unsigned` oznacza się literą `U` lub `u`, np. `11U`, `011UL`, `0x011Lu`.

Stała nie mieszcząca się w zakresie typu `int` jest traktowana jako typu `long int`.

Stałe zmiennoprzecinkowe

Za ich pomocą zapisuje się liczby 'rzeczywiste', np. `1.0`, `32.0`, `123.4e56`, `-1.1E-1`.

Występująca na końcu stałej zmiennopozycyjnej litera `f` lub `F` oznacza stałą typu `float`, a litera `l` lub `L` – obiekt typu `long double`, np. `1.0f`, `-5.555L`.

Każda stała zmiennoprzecinkowa bez modyfikatora traktowana jest jak liczba o podwójnej precyzji.

Uwaga na różnicę pomiędzy stałymi z i bez kropki dziesiętnej, np. `123` ***i*** `123.0`***.***

Stałe znakowe

Stałą znakową tworzy jeden znak ujęty w apostrofy, np. `'a'`. Jej wartością jest numer (kod) znaku w maszynowym zestawie znaków (czyli najczęściej ASCII).

Pewne znaki niegraficzne lub zarezerwowane przedstawia się w nastepujący sposoób za pomocą sekwencji znaków:

`\n` – znak nowego wiersza,

`\r` – powrót na początek wiersza,

`\t` – tabulacja,

`\b` – cofnięcie o jedną spację (backspace),

`\0` – znak 'zerowy' (null),

`\'` – apostrof (`'`),

`\"` – cudzysłów (`"`),

`\\` – znak \ (backslash).

Dowolny znak można utworzyć także poprzez jego kod ósemkowy *ddd* następująco: `'\`*ddd*`'`, np. `'\014'` (= `'\n'`) oraz poprzez kod szesnastkowy *hh* – `'\x`*hh*`'`, np. `'\x14'`.

Napisy, (pseudo)stałe napisowe, tekstowe

Stała napisowa jest ciągiem znaków. W języku C wyróżnia się ją cudzysłowami, np. `"This is text.\n"`.

Fizycznie stała tekstowa jest tablicą, której elementami są poszczególne znaki. Ostatnim znakiem, dodawanym automatycznie przez kompilator, jest znak zerowy `'\0'`.

Uwaga! Napisy w zasadzie nie są stałymi, ponieważ można pobrać adresy tworzących je znaków.

Literał napisowy jest adresem pierwszego znaku napisu.

W języku C wszystkie napisy, także wyglądające dokładnie tak samo (np. dwa wystąpienia tekstu `"text"` w programie), są różne, w więc zajmują różne obszary pamięci.

Uwaga! W `djgpp` ***jest inaczej! Identyczne napisy są odwołaniem do tej samej tablicy znakowej!***

W napisie mogą wystąpić wszystkie sekwencje specjalne wymienione dla stałych znakowych, w szczególności znak cudzysłowu wstawia się jako `\"`, np. `"file \"name\""`.

Sekwencja złożona ze znaku \ i występującego bezpośrednio po nim znaku nowego wiersza jest ignorowana.

Zauważmy, że stałe `'"'` lub `'\"'` oraz `"'"`, `'\''` i `"\""` są poprawne, natomiast `'''` i `"""` nie.

Uwaga! Stała znakowa i stała napisowa zawierająca jeden znak to dwa odmienne obiekty, np. `'x'` ***oraz*** `"x"`***. Pierwsza jest typu*** `char`***, druga*** `char []`***.***

Deklaracja i definicja zmiennej

Zmienna to obiekt, który pozwala pobierać, przechowywać i przetwarzać dane. W programie źródłowym rozpoznawana jest poprzez nazwę (identyfikator).

W odróżnieniu od stałej, zmienna to obiekt adresowalny. Oznacza to, że w przypadku zmiennej mamy dostęp zarówno do jej wartości, jak i adresu.

Adres wskazuje na obszar pamięci przydzielony zmiennej, natomiast ***wartość*** zmiennej stanowi zawartość tego obszaru pamięci.

Uwaga: Adres dowolnej zmiennej jest wartością stałą.

Definicja zmiennej polega na określeniu jej typu (za pomocą specifikatora typu), nadaniu nazwy oraz przydzieleniu określonej ilości pamięci. Dodatkowo można jej nadać wartość początkową, czyli zainicjować.

```
int i ;
```

```
float sum=0.0, dv ;
char string[]="... and we get...", c ;
char backslash='\\' ;
```

Deklaracja zmiennej polega na oznajmieniu, że zmienna ta istnieje i została zdefiniowana w innym miejscu.

Deklaracja składa się ze specyfikatora typu oraz nazwy zmiennej, poprzedzonych słowem `extern`.

```
extern int i ;
```

Uwaga! Każdą zmienną, zanim się jej użyje, trzeba wpierw zdefiniować lub zadeklarować.

Operatory

Operator jest symbolem służącym do wykonania pewnej operacji. Może być jedno- lub dwuargumentowy, przemienny, łączny oraz lewo- lub prawostronnie łączny.

Operator **O** jest

przemienny, jeśli $a\,\mathbf{O}\,b = b\,\mathbf{O}\,a$,

łączny, jeśli kolejność wykonywania operacji jest dowolna, tj. $a\,\mathbf{O}\,b\,\mathbf{O}\,c = (a\,\mathbf{O}\,b)\,\mathbf{O}\,c = a\,\mathbf{O}\,(b\,\mathbf{O}\,c)$,

lewostronnie łączny, jeśli najpierw wykonywana jest operacja lewa, tj. $a\,\mathbf{O}\,b\,\mathbf{O}\,c = (a\,\mathbf{O}\,b)\,\mathbf{O}\,c$,

prawostronnie łączny, jeśli najpierw wykonywana jest operacja prawa, tj. $a\,\mathbf{O}\,b\,\mathbf{O}\,c = a\,\mathbf{O}\,(b\,\mathbf{O}\,c)$.

Operator przypisania

Przypisanie: `=`.

```
int x, y ;
;
x=5 ;
x=x ;
y=x ;
```

Uwaga: W języku C operacja przypisania posiada wartość równą wartości lewego argumentu. Jej typ jest taki sam jak typ lewego argumentu.

```
int x, y ;
;
x=y=5 ; x=(y=5) ;
```

Przypisania wykonuje się od prawej strony!

Operatory arytmetyczne

Dwuargumentowe: `+ - * / %`

Jednoargumentowe: `- +`

Przy dzieleniu liczb całkowitych wynik jest zaokrąglany w kierunku zera, np. `8/3` i `-8/-3` dają `2`, a `8/-3` i `-8/3` dają `-2`.

Operator dzielenia modulo `%` można stosować jedynie dla danych typu całkowitego.

Wynik dzielenia modulo ma ten sam znak co dzielna, np. `8%3` i `8%-3` dają `2`, a `-8%3` i `-8%-3` dają `-2`.

Jeśli `a` i `b` są liczbami całkowitymi oraz `b` nie jest zerem, to zawsze `(a/b)*b+a%b` równa się `a`.

```
int a, b, x ;
;
x=-a ; x=+b ;
;
// x=a--b ; // error
x=a-(-b) ;
;
x=-a*b+c ;
x=-a+b ; x=-(a+b) ; // - before expression
```

Uwaga: Kolejność wykonywania obliczeń dla operatorów łącznych i przemiennych (np. `*` i `+`) nie jest określona. Kompilator może zmienić kolejność obliczania wyrażeń nawet jeśli ujęte są w nawiasy. Aby wymusić określoną kolejność należy stosować zmienne tymczasowe.

```
int a, b, c ;
int x, temp ;
;
x=a+(b+c) ; // may be computed as (a+b)+c !
            // and solution is:
temp=b+c ;
x=a+temp ;
```

Priorytety (od najmniejszego): operator przypisania ; `+ -` ; `* / %` ; `-`.

W celu wymuszenia innej niż wynikająca z zasad pierwszeństwa kolejności wykonywania operacji, należy posłużyć się nawiasami okrągłymi `()`. Niekiedy jest to konieczne aby osiągnąć zamierzony cel.

```
int x, y ;
;
y=x=3+8 ;   // x=11 y=11
;
y=(x=3)+8 ; // x= 3 y=11
```

Operatory relacji i przyrównania

Uwaga: W języku C nie ma odrębnego typu logicznego. Za 'prawdę' uważa się dowolną wartość całkowitą różną od `0`, a 'fałsz' jest równoważny wartości `0`.

Relacje: `> >= < <=`

Przyrównania: `== !=`

Wynikiem działania powyższych operatorów jest `1`, jeżeli relacja lub przyrównanie są prawdziwe, oraz `0`, gdy są fałszywe. Typem wyniku jest `int`.

Operatory relacji mają wyższy priorytet od operatorów przyrównania. Wszystkie są lewostronnie łączne.

```
int x ;
;
x=3<2<1 ;     // x=0<1 ; x=1 ;
x=(3<2)<1 ;
;
x=1==3==3 ;   // x=0==3 ; x=0 ;
x=1==(3==3) ; // x=1==1 ; x=1 ;
;
x=1<2!=2<1 ;  // x=1!=0 ; x=1 ;
```

Priorytety (od najmniejszego): operator przypisania ; `== !=` ; `> >= < <=` ; operatory arytmetyczne.

```
int i, n ;
```

```
;
if(i<=n-1) ...  // is equivalent to
if(i<=(n-1)) ...
```

Uwaga: O ile za 'prawdę' uważa się dowolną wartość całkowitą różną od `0`, a 'fałsz' jest równoważny wartości `0`, o tyle wartością dowolnego wyrażenia logicznego może być albo `1` ('prawda') albo `0` ('fałsz').

```
int x=3, y=5, z ;
;
z=x<y ;             // z=1
;
z=x<y-8 ;           // z=0
z=x<(y-8) ;
;
z=(x<y)-8 ;         // z=-7
;
z=x>=y+y>=x ;       // z=0
z=(x>=(y+y))>=x ;
;
z=x>=y+(y>=x) ;     // z=0
z=x>=(y+(y>=x)) ;
;
z=(x>=y)+(y>=x) ; // z=1
;
z=x==y ;            // z=0
;
z=x==y-2 ;          // z=1
z=x==(y-2) ;
```

Operatory logiczne

Łączniki: 'i' – `&&`, 'lub' – `||`.

Wyrażenia połączone powyższymi operatorami oblicza się od lewej do prawej.

Wynikiem działania operatora ***iloczynu logicznego*** `&&` jest `1`, jeżeli oba argumenty są różne od `0`; w pozostałych przypadkach wynikiem jest `0`. Drugi argument nie zostanie obliczony, jeśli pierwszy ma wartość `0`.

Wynikiem działania operatora ***sumy logicznej*** `||` jest `1`, jeżeli co najmniej jeden z argumentów jest różny od `0`; w przeciwnym przypadku wynikiem jest `0`. Drugi argument nie zostanie obliczony, jeśli pierwszy ma wartość różną od `0`.

```
if((c=getchar())!='\n' && c!=EOF)
  ; // c has to be known before using it
```

Negacja `!`.

```
if(!(file=fopen("dane.dat","rt")))
  {
  printf(" Error: file not open.\n");
  return(0);
  }
int found=0 ;
;
if(!found)
  ;
if(found==0)
  ;
```

Priorytety (od najmniejszego): operator przypisania ; `||` ; `&&` ; operatory przyrównania ; operatory relacji ; operatory arytmetyczne ; `!` ;

Bitowe operatory logiczne

Pozwalają one na manipulację bitami zmiennych tylko całkowitych.

`&` – bitowa koniunkcja ('i'),

`|` – bitowa alternatywa ('lub'),

`^` – bitowa różnica symetryczna,

`<<` – przesunięcie w lewo,

`>>` – przesunięcie w prawo,

`~` – uzupełnienie jedynkowe,

Operator koniunkcji używany jest do 'zerowania' pewnego zbioru bitów (`0&0==0&1==1&0==0`, `1&1==1`).

Operator alternatywy używany jest do 'ustawiania' pewnego zestawu bitów (`1|1==0|1==1|0==1`, `0|0==0`).

Operator różnicy symetrycznej ustawia `1` na każdej pozycji bitowej tam, gdzie bity obu argumentów są różne, a `0` tam, gdzie są takie same (`1^1==0^0==0`, `0^1==1^0==1`).

Jeśli prawy argument operatorów przesunięcia `<<` i `>>` jest ujemny, bądź większy lub równy długości lewego argumentu w bitach, to wynik operacji jest nieokreślony.

```
int x=222, y ;
;
y=x&0177 ;
y=x|0177 ;
y=~x ; // 0 onto 1, and 1 onto 0
y=x<<2 ; // two right bits are set to 0
y=x>>2 ; // two left bits are set to 0
```

Uwaga: na różnicę pomiędzy `&&` `||` oraz `&` `|`.

```
int x=1, y=2, z ;
;
z=x&y ;           // 0
z=x&&y ;          // 1
```

Operatory zmniejszania i zwiększania

Jednoargumentowy operator zwiększania `++` dodaje do swego argumentu liczbę 1, operator zmniejszania `--` odejmuje odeń 1.

Operatory te stosuje się tylko do zmiennych.

```
int c, nlines=0 ;
;
if(c=='\n') nlines++ ;
```

Operator przedrostkowy zmienia argument przed jego użyciem, operator przyrostkowy po jego użyciu.

```
int x, n=5 ;
;
x=n++ ;       // x=5
// x=n ; n++ ;
n-- ;
x=++n ;       // x=6
// n++ ; x=n ;
;
```

```
(n+x)++ ;     // error: expression
x++=n ;       // error: invalid lvalue
++x=n ;       // error: invalid lvalue
int x=3, z=0 ;
;
z=x--+x ;      // 2 6
// z=x+x ; x-- ;
;
z=--x+x ;      // 1 2
// x-- ; z=x+x ;
;
z=x+++x++ ;    // 3 2
// z=x+x ; x++ ; x++ ;
;
z=++x+x++ ;    // 5 8
// x++ ; z=x+x ; x++ ;
;
z=++x+(++x) ; // 7 14
// x++ ; x++ ; z=x+x ;
```

Uwaga: Wszystkie operatory jednoargumentowe mają taki sam priorytet oraz są prawostronnie łączne.

Zatem znane nam już operatory `!` (negacja logiczna), `~` (negacja bitowa), `-` (negacja arytmetyczna), `++` oraz `--` są prawostronnie łączne. Z tego powodu wyrażenie `---x` rozumiane jest jako `--(-x)` i jest oczywiście niepoprawne.

```
int x=5, y=3, z ;
;
// z=---x ;      // error: invalid lvalue
// z=--(-x) ;    // -x: negation of x
;
z=-(--x) ;       // x=4 z=-4
;
z=x---y ;        // x=3 y=3 z=1
// z=(x--)-y ;   // -y: subtracting y
;
z=x-(--y) ;      // x=3 y=2 z=1
;
x=5, y=3 ;
;
// z=x-----y ;   // error: invalid lvalue
// z=((x--)--)-y ;
;
z=x---(--y) ;    // x=4 y=2 z=3
// z=(x--)-(--y) ;
;
z=x---(-(-y)) ; // x=3 y=2 z=2
;
z=x--+--y ;      // x=2 y=1 z=4
;
// z=x--+++y ;   // error: invalid lvalue
// z=((x--)++)+y ;
;
z=x--+(++y) ;    // x=1 y=2 z=4
```

Rozszerzone operatory przypisania

Dla większości operatorów dwuargumentowych istnieje w języku C operator przypisania **O**=, gdzie **O** jest jednym z operatorów `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&` i `|`.

Jeśli ***object*** jest zmienną, a ***expression*** pewnym wyrażeniem, to

object* O= *expression

jest równoważne

***object* = *object* O (*expression*)**

```
int n=2 ;
float x, y=1, z=2 ;
;
x+=2 ;
x=x+2 ;
;
n<<=3 ;
n=n<<3 ;
;
x*=y+z ;
x=x*(x+z) ;
```

Rozszerzone operatory przypisania mają taki sam priorytet jak operator przypisania oraz są prawostronnie łączne.

```
int x=5, z=0 ;
;
x+=x ; // 10
// x*=2 ;
;
x=5 ;
z=(x-=1)+3 ; // 4 7
;
x=5 ;
z=x-=1+3 ;    // 1 1
// z=(x-=(1+3)) ;
;
z+=x+=2 ;     // 3 4
// z+=(x+=2) ;
;
z=x=5 ;
z-=x+=--x ;   // 8 -3
// x-- ; x+=x ; z-=x ;
```

Przekształcenia typów

Jeśli w danym wyrażeniu występują obiekty o różnych typach, to są one przekształcane do typu wspólnego według kilku reguł.

```
int a=3, x ;
float b=5.8, y ;
;
x=a+b ;
y=a+b ;
```

W przypadku niejawnych przekształceń typów, typ 'węższy' zamieniany jest przed wykonaniem operacji na typ 'szerszy'.

Ogólnie, argumenty wszystkich operatorów arytmetycznych podlegają następującym regułom przekształceń typów:

`char` i `short int` przekształcane są na `int`,

`float` jest przekształcany na `double`.

Jeśli `double` i x, to x na `double` i wynik jest `double`, w przeciwnym przypadku, jeśli `long int` i x, to x na `long int` i wynik jest `long int`,

w przeciwnym przypadku, jeśli `unsinged int` i x, to x na `unsinged int` i wynik jest `unsinged int`,

w przeciwnym przypadku, jeśli `int` i x, to x na `int` i wynik jest `int`.

Wartość prawej strony przypisania jest przekształcana do typu lewej strony, który będzie typem wyniku.

Zamiana typów `float` i `double` na typ `int` prowadzi do obcięcia części ułamkowej.

```
int i ;
float f=5.2 ;
double d=13.9 ;
;
i=f ; // i==5
i=d ; // i==13
```

Typ `double` przekształcany jest na `float` przez zaokrąglenie.

Dłuższe liczby całkowite są przekształcane na krótsze lub na typ `char` przez odrzucenie 'wystających', bardziej znaczących bitów.

Podobnie jest w przypadku przekształcania typów argumentów funkcji.

Operator rzutu

Rzutem nazywamy konstrukcję pozwalającą jawnie wymusić zmianę typu. Ma ona postać:

(***type***) ***expression***

Wyrażenie ***expression*** jest przekształcane według podanych reguł do typu określonego przez ***type***.

```
int i, j ;
float x=3.3, y=3.7, z ;
;
i=x+y ;                 // 7
j=(int)x+(int)y ;       // 6
z=x+y ;                 // 7.0
```

Rzut ma taki sam priorytet jak inne operatory jednoargumentowe.

Operator rozmiaru

Operator `sizeof` przekazuje ilość bajtów będącą rozmiarem obiektu lub typu danych.

```
long int i ;
double x ;
;
i=sizeof(int) ;
i=sizeof(x) ;
```

Zadanie: Napisać program, który wypisze rozmiary wszystkich poznanych typów danych.

Operator przecinkowy

Przecinek **,** służy do rozdzielania wyrażeń.

Wyrażenie przecinkowe jest ciągiem wyrażeń rozdzielonych przecinkami.

```
int i, j ;
;
i=j=0 ;
j=0 ; i=j ;
;
i=0, j=0 ;
;
i=(0,j=5) ;  // i=5
i=0,j=5 ;    // i=0
```

Wyrażenia w wyrażeniu przecinkowym oblicza się kolejno, zaczynając od pierwszego wyrażenia z lewej strony.

Wynikiem wyrażenia przecinkowego jest wartość ostatniego wyrażenia.

```
char x, y ;
;
y=1 ; x=3 ;
y=x++,y+=x ;           // x=4 y=7
// (y=x++),(y+=x) ;
;
y=1 ; x=3 ;
y=(x++,y+=x) ;         // x=4 y=5
```

Różnice w działaniu operatorów przed- i przyrostkowych `++` i `--` doskonale są widoczne po zapisaniu zawierających je instrukcji za pomocą wyrażeń przecinkowych.

```
int x=3, y=5, z ;
;
y=x++ ;      // x=4 y=3
// y=x,x++ ;
;
y=++x ;      // x=5 y=5
// ++x,y=x ;
;
z=--x*++y ;  // x=4 y=6 z=24
// z=(--x,++y,x*y) ;
// --x,++y,z=x*y ;
;
z=x--*--y ;  // x=3 y=5 z=20
// z=(--y,x--*y) ;
// --y,z=x*y,x-- ;
```

Wyrażenie warunkowe

Wyrażenie warunkowe jest uproszczoną wersją konstrukcji typu:

```
if(a>b) z=a ;
else    z=b ;
```

Wyrażenie warunkowe tworzy się za pomocą trzyargumentowego operatora `?:` następująco:

expr`?`***expr-true***`:`***expr-false***

Jeśli wyrażenie ***expr*** jest prawdziwe to wartością operacji będzie wartość wyrażenie ***expr-true***; w przeciwnym przypadku wartością operacji będzie wartość wyrażenia ***expr-false***.

```
z=(a>b)?a:b ;  // max(a,b)
;
for(i=0 ; i<n ; i++)
  printf("%3d%c", i,
    (!(i%5) || i==n-1)?'\n':' ');
```

Obiekty i l-wartości

Wprawdzie podczas wykładu nie będziemy używać pojęcia l-wartości, jednak przytaczam poniżej jej definicję.

Obiekt jest obszarem pamięci, na którym wykonuje się pewne operacje; ***l-wartość*** (ang. lvalue) jest wyrażeniem odnoszącym się do obiektu. Najbardziej oczywistym przykładem l-wartości jest identyfikator (zmiennej). Również niektóre operatory dają l-wartość.

Nazwa 'l-wartość' wywodzi się stąd, że lewy argument operatora przypisania musi być właśnie l-wartością.

Większość operatorów wymagających jako (lewego) argumentu l-wartości nie daje w wyniku l-wartości, np. operatory `++`, `--`, operatory przypisania `=`, `+=`, `*=` itd.

Priorytety

Tabela poniższa zawiera podsumowanie zasad pierszeństwa i łączności wszystkich operatorów. Wiersze tabeli są uporządkowane według malejącego priorytetu (od dołu do góry).

Operator	Łączność
`() [] -> .`	L
`! ~ + - ++ --` (*type*) `* & sizeof`	P
`* / %`	L
`+ -`	L
`<< >>`	L
`< <= > >=`	L
`== !=`	L
`&`	L
`^`	L
`|`	L
`&&`	L
`||`	L
`?:`	P
`= += -= *= /= %= <<= >>= &= |=`	P
`,`	L

Uwaga: Dobrze jest jednak używać nawiasów.

Operatory jednoargumentowe

Jak pamiętamy wszystkie operatory jednoargumentowe mają taki sam priorytet oraz są prawostronnie łączne. Ponadto wszystkie, oprócz operatorów `++` i `--`, wymagają argumentu z prawej strony. `++` oraz `--` jako jedyne mogą działać na argument z obu stron.

Prawostronna łączność operatorów jednoargumentowych polega na tym, że jeśli dany obiekt otoczony jest z obu stron takimi operatorami, to pierwszeństwo ma ten stojący z prawej strony. Zatem `-x--` oznacza `-(x--)`, a `!x++` oznacza `!(x++)` itd.

```
int x=3, y ;
;
y=!-x ;
;
y=-!(char)sizeof(x) ;
;
y=-x-- ;
y=-++x ;
y=!x++ ;
y=!++x ;
;
// y=---x ; // error: invalid lvalue
// y=--(-x) ;
;
y=(char)x++ ;
y=(char)--x ;
```

Kolejność obliczeń

W języku C nie określa się kolejności obliczania wartości argumentów operatora. Np. w instrukcji

```
x=f()+g() ;
```

funkcja `f` może być wykonana przed funkcją `g` lub odwrotnie.

Należy podkreślić również to, że zasady pierwszeństwa oraz łączności określają jedynie sposób interpretowania wyrażeń języka, jednak nie ustalają porządku opracowywania elementów wyrażeń, w szczególności kolejności obliczania argumentów operatorów.

Wyrażenie arytmetyczne

```
a - b + c * d
```

jest interpretowane tak jak wyrażenie

```
(a - b) + (c * d)
```

ponieważ po pierwsze operator mnożenia `*` ma priorytet wyższy od operatorów `+` oraz `-`, a po drugie operatory dodawania i odejmowania są lewostronnie łączne. Mimo to, nadal nieokreślona pozostaje kolejność obliczania argumentów operatora dodawania.

Ponadto w języku C zezwala się na dowolne przestawianie argumentów operatorów łącznych i przemiennych (`*`, `+`, `&`, `|`, `^`) nie bacząc na obecność nawiasów.

```
int x=5, y ;
;
x=5 ;
y=(x+=2)+(x=2) ;
// x+=2 : 7  +  x=2  : 2  =  9
// x=2  : 2  +  x+=2 : 4  =  6
;
x=5 ;
y=x+(x=2) ;         // x=2 y=4
// y=(x=2,x+x);
;
x=5 ;
y=++x+(x=2) ;       // x=2 y=4 ???
// y=(++x,x+(x=2)) ;
// y=(++x,x=2,x+x) ;
;
x=5 ;
y=++x ; y+=(x=2) ; // x=2 y=8
;
x=5 ;
y=(x=2)+(++x) ;    // x=3 y=5
// x=2 ; ++x ; y=2+3 ;
;
y=(x=2) ; y+=++x ; // x=3 y=5
```

Wyjątek stanowią operatory logiczne `&&` i `||` oraz operator przecinkowy `,`, których argumenty obliczane są od lewej do prawej, a także operator warunkowy, dla którego najpierw obliczany jest pierwszy argument.

Podobnie kolejność obliczania wartości argumentów funkcji nie jest określona.

```
printf("%d %d", ++n, power(2,n));
```

Uwaga: Programy należy pisać w taki sposób aby nie zależały od kolejności wykonywania obliczeń.

Sterowanie

Sterowanie odbywa się za pomocą instrukcji sterujących.

Instrukcją prostą nazywamy wyrażenie zakończone średnikiem.

Do grupowania instrukcji prostych w jedną instrukcję złożoną (blok) używane są nawiasy klamrowe {}.

Instrukcja złożona jest ciągiem instrukcji prostych ujętym w parę nawiasów klamrowych.

Instrukcja złożona jest równoważna składniowo jednej instrukcji.

Instrukcji złożonej nie trzeba kończyć średnikiem.

W dalszym ciągu wykładu pod pojęciem instrukcji będziemy rozumieć także blok instrukcji prostych.

Instrukcja pusta

Instrukcja pusta składa się tylko ze średnika.

```
int i ; ;
;
i=3 ;
```

Powyżej mamy dwie instrukcje puste.

Instrukcja wyrażeniowa

Instrukcja wyrażeniowa ma postać

expr ;

gdzie ***expr*** jest dowolnym wyrażeniem. Wykonanie instrukcji wyrażeniowej powoduje wyznaczenie wartości wyrażenia ***expr***, a następnie zignorowanie jej.

```
int a ;
;
2+(a=5) ;  // equivalent to
a=5 ;
```

Instrukcja grupująca

Instrukcja grupująca jest w zasadzie instrukcją złożoną ujętą w parę nawiasów klamrowych.

{ ***statements*** }

Wykonanie instrukcji grupującej polega na sekwencyjnym wykonaniu zawartych w niej instrukcji ***statements***.

Instrukcja warunkowa `if-else`

```
if(expression)
  statement-true
else
  statement-false
```

Jeżeli wyrażenie ***expression*** jest prawdziwe (tzn. jego wartość jest różna od zera), to wykonana zostanie instrukcja ***statement-true***. W przypadku przeciwnym wykonana zostanie instrukcja ***statement-false***.

Część `else` można pominąć.

```
if(expression)
  statement
```

Jeżeli wynik wyrażenia ***expression*** jest różny od zera, to znaczy że warunek instrukcji `if` jest spełniony i instrukcja ***statement*** zostanie wykonana. W przeciwnym przypadku instrukcja ***statement*** zostanie pominięta.

```
int x, y, z ;
;
z=y ;
if(x>y) z=x ;
if(x>0 && y>0)
  {
  printf(" x and y are both positive.\n");
  z=x-y ;
  }
else
  {
  printf(" x and/or y is/are negative.\n");
  z=y-x ;
  }
int d, m, s ;
float dec ;
;
dec=d ;
if(d>=0) dec+=(m+s/60.0)/60.0 ;
else     dec-=(m+s/60.0)/60.0 ;
;
dec=d+(d>=0?1.0:-1.0)*(m+s/60.0)/60.0 ;
```

Każda z części `else` jest przyporządkowana najbliższej z poprzednich instrukcji `if` nie zawierających części `else`.

```
if(n>0)
  {
  if(a>b) z=a ;
  else    z=b ;
  }
```

W powyższym przykładzie parę nawiasów {} można pominąć, natomiast w przykładzie poniżej nawiasy {} są konieczne.

```
if(n>0)
  {
  if(a>b) z=a ;
  }
else z=b ;
```

Ciąg instrukcji `if-else` jest najbardziej ogólnym sposobem podejmowania decyzji wielowariantowych.

Instrukcja wyboru `switch`

```
switch(expression)
  {
  case const1  :
    statement1
  case const2  :
    statement2
  ...
  default :
    statement
```

```
  }
```

Instrukcja `switch` oblicza wartość całkowitą wyrażenia ***expression*** i porównuje ją po kolei z przypadkami `case`. Po znalezieniu zgodnego przypadku, wykonany zostanie blok instrukcji ***statement*** związany z tym przypadkiem i wszystkie następne aż do najbliższego rozkazu `break` lub końca instrukcji `switch`. Przypadek `default` zostanie wykonany wtedy, gdy żaden inny przypadek nie jest zgodny z wartością wyrażenia ***expression***.

Z każdym przypadkiem (`case`) musi być związana stała całkowita lub znakowa.

Wszystkie przypadki muszą być różne.

```
int c, nlines=0, nwhite=0, nother=0, na=0 ;
;
switch(c) ;
  {
  case 'a' :
    na++ ;
    break ;
  case '\n' :
    nlines++ ;
  case ' ' :
  case '\t' :
    nwhite++ ;
    break ;
  default :
    nother++ ;
    break ;
  }
;
if(c=='a') na++ ;
else if(c=='\n') { nlines++ ; nwhite++ ; }
else if(c==' ' || c=='\t') nwhite++ ;
else nother++ ;
```

Pętla `while`

Pętla to cykliczne wykonywanie bloku instrukcji dopóty, dopóki spełniony jest pewien warunek.

```
while(expression)
  statement
```

W pętli `while` instrukcja ***statement*** wykonywana jest krok po kroku dopóty, dopóki wyrażenie ***expression*** jest prawdziwe, czyli różne od `0`.

Kolejność działań jest następująca: (1) oblicza się wartość wyrażenia ***expression***, (2) jeśli warunek jest prawdziwy (***expression***`!=0`), to wykonuje się instrukcję ***statement***, ponownie wykonuje się (1) itd., aż do momentu, gdy warunek staje się fałszywy.

Jeżeli już przy pierwszym obliczeniu wyrażenia ***expression*** okazuje się ono fałszywe, to instrukcja ***statement*** nie zostanie wykonana ani razu.

```
int arg=4, n=3, power ;
;
power=1 ;
while(n)
  { power*=arg ; n-- ; }
```

Pętla `for`

```
for(expr-init ; expr-cond ; expr-modif)
  statement
```

Powyższe jest równoważne:

```
expr-init ;
while(expr-cond)
  {
  statement
  expr-modif ;
  }
```

Zatem w instrukcji `for` najpierw wykonywane jest (tylko raz, przed wejściem do właściwej pętli `while`) wyrażenie inicjujące ***expr-init***. Następnie mamy pętlę `while`, której każdy krok polega na sprawdzeniu warunku w wyrażeniu ***expr-cond***, a następnie, jeśli warunek ten jest prawdziwy, na wykonaniu instrukcji ***statement*** oraz zaraz po niej wyrażenia modyfikującego ***expr-modif***. Zakończenie pętli następuje wtedy, gdy wyrażenie warunkowe ***expr-cond*** staje się nieprawdziwe.

```
int arg=4, n=3, power ;
;
for(power=1 ; n ; n--)
  power*=arg ;
```

```
int nx=20, ny=20, x, y ;
float result ;
;
for(x=0 ; x<nx ; x++)
  for(y=0 ; y<ny ; y++)
    result+=x-y ;
```

```
int n ;
float avr, x, xb=20, xe=400 ;
;
for(avr=n=0, x=xb ; x<=xe ; x+=2, n++)
  avr+=x ;
if(n) avr/=n ;
```

Dowolną z części ***expr*** (także wszyskie) można pominąć. Jeśli brakuje ***expr-cond***, to przyjmuje się, że warunek jest zawsze prawdziwy.

```
for(;;) { }  // we are heading to infinity...
for(;1;) ;   // we are still heading there...
```

```
signed char sc ;
unsigned char uc ;
unsigned short int i=~0 ; // (~0)&~(1<<7) ;
int n ;
;
i=1<<7|1<<3 ; // (1<<7)|(1<<3)
i=~0&~(1<<7)&~(1<<3) ;
;
sc=i ;
uc=i ;
;
for(n=sizeof(i)*8-1 ; n>=0 ; n--)
  {
  if(1<<n&i) { printf("1"); continue ; }
  printf("0");
  }
printf("\n");
;
```

```
printf("        ");
for(n=sizeof(sc)*8-1 ; n>=0 ; n--)
  {
  if(1<<n&sc) { printf("1"); continue ; }
  printf("0");
  }
printf("\n");
;
printf("        ");
for(n=sizeof(uc)*8-1 ; n>=0 ; n--)
  {
  if(1<<n&uc) { printf("1"); continue ; }
  printf("0");
  }
printf("\n");
;
printf(" %d %d %d\n", i, (int)sc, (int)uc);
```

Pętla `do-while`

```
do
  statement
while(expression);
```

Pętle `while` i `for` mają tę cechę, że warunek ich zatrzymania sprawdzany jest na początku kroku. W pętli `do-while` warunek ten sprawdzany jest na końcu kroku, a zatem będzie ona wykonana przynajmniej raz.

```
int arg=4, n=3, power ;
;
power=1 ;
do
  { power*=arg ; n-- ; }
while(n>0);
;
power=1 ;
while(power*=arg,--n>0);
```

```
char *line, **text ;
FILE *file ;
int n=0 ;
for( ; !feof(file) ; )
  {
  do
    { getln(file,line); }
  while(*line!='%');
  strcopy(*text+n,line);
  n++ ;
  }
```

Instrukcja `break`

Instrukcja `break` wymusza natychmiastowe wyjście z pętli lub instrukcji wyboru `switch`.

```
// desperately looking for... %
char *line ;
int n=1 ;
FILE *file ;
for( ; getln(file,line)!=EOF ; n++)
  {
  if(*line=='%') break ;
  }
```

Instrukcja `continue`

Instrukcję `continue` stosuje się wewnątrz pętli. Powoduje ona przerwanie bieżącego i wykonanie od początku następnego kroku pętli.

Dla pętli `while` i `do-while` oznacza to natychmiastowe sprawdzenie warunku, natomiast w pętli `for` powoduje przekazanie sterowania do części modyfikującej.

```
// lines beginning with '%' are comments
char *line ;
int nlines=1, ncommnets=0 ; ;
FILE *file ;
for( ; getln(file,line)!=EOF ; nlines++)
  {
  if(*line=='%') { ncomments++ ; continue ; }
  printf(" %s\n", line);
  }
```

Tablice

Tablica jest zestawem obiektów tego samego typu. Formalnie, każdy element tablicy jest zwykłą zmienną, choć nie posiada odrębnej nazwy. Dostęp do elementu tablicy możliwy jest poprzez ***indeksowanie***, czyli podanie pozycji.

Tablice tworzy się za pomocą pary nawiasów `[]`. Jak się później przekonamy, symbol `[]` jest także operatorem, zwanym operatorem indeksowania.

```
int ndigit[20] ;
char name[]="Grzegorz Brzeczyszczykiewicz" ;
double sum[]={1.0, 2.0, 3.0} ;
double x[] ;
```

Liczba w nawiasach `[]` (musi być wyrażeniem stałym) oznacza ***wymiar*** tablicy, czyli ilość jej elementów. Wymiar można pominąć, ale wtedy podczas deklarowania tablicy należy ją zainicjować. W takim przypadku kompilator sam obliczy sobie ilość elementów.

Tablice inicjuje się poprzez podanie w nawiasach klamrowych listy wartości poszczególnych elementów rozdzielonych przecinkami.

Uwaga! Indeksowanie tablic w języku C rozpoczyna się od zera, a nie od jedynki. Zatem pierwszy element danej tablicy wybierany jest poprzez `[0]` ***itd.***

```
#define NX 30
int i, x[NX] ;
for(i=0 ; i<NX ; i++) x[i]=i ;
```

Indeks tablicy może być dowolnym wyrażeniem o wartości całkowitej.

Przypominam, że stała napisowa jest równoważna tablicy znakowej o długości o 1 większej od ilości znaków napisu.

```
char str[]="krasa" ;
char tab[]={'k', 'r', 'a', 's', 'a', '\0'} ;
char text[20] ;
int i, n=sizeof(str)/sizeof(char) ;
;
for(i=0 ; i<n ; i++)
  if(tab[i]!=str[i]) break ;
if(i==n) printf("tab and str are the same.");
```

```
else     printf("tab and str differ.");
;
for(i=0 ; text[i]=str[i] ; i++) ;
```

Uwaga: Ani podczas tłumaczenia, ani podczas wykonywania programu system operacyjny nie sprawdza, czy wartość indeksu tablicy należy do właściwego zakresu!

Tablice wielowymiarowe

W języku C, z definicji, tablica wielowymiarowa jest w istocie tablicą jednowymiarową, której elementami są także tablice. Z tego powodu do elementu `(i,j)` tablicy dwuwymiarowej dostajemy się poprzez naturalną konstrukcję `[i][j]` a nie `[i,j]`.

Przykładowe definicje tablic wielowymiarowych podane są poniżej.

```
int itab2[20][30] ;
float ftab3[10][20][30] ;
unsigned char uchtab4[2][4][6][8] ;
```

Rozważmy dla prostoty tylko tablice dwu- i trójwymiarowe:

```
#define NR  5
#define NC 20
#define NH 50
;
int i, j ;
float (ftab[NR])[NC] ;
int ((itab[NR])[NC])[NH] ;
```

Powyżej `ftab` jest jednowymiarową tablicą tablic jednowymiarowych. Z kolei `itab` jest jednowymiarową tablicą tablic dwuwymiarowych. Dokładniej mówiąc `itab` jest tablicą o `NR` elementach, z których każdy jest tablicą o `NC` elementach, z których każdy jest tablicą o `NH` elementach będących liczbami typu `int`.

Jak widać wyrażenia `ftab[i]`, `i=0, ..., NR-1`, są typu `float [NC]`, a więc oznaczają jednowymiarowe tablice `NC` obiektów typu `float`. Podobnie, wyrażenia `itab[i][j]`, `i=0, ..., NR-1`, `j=0, ..., NC-1`, są typu `int [NH]`, a zatem również są jednowymiarowymi tablicami, tym razem `NH` obiektów typu `int`. Ponadto, wyrażenia `itab[i]`, `i=0, ..., NR-1`, są dwuwymiarowymi tablicami typu `int [NC][NH]`.

Elementy tablic wielowymiarowych umieszczane są w pamięci wierszami, co oznacza, że skrajnie prawy indeks zmienia się najszybciej w przypadku, jeśli elementy obsługiwane są w kolejności ich położenia w pamięci. Np. elementy tablicy `t` zdefiniowanej jako `int t[2][3][4];` układane są pamięci w następującej kolejności:

```
t[0][0][0] t[0][0][1] t[0][0][2] t[0][0][3]
t[0][1][0] t[0][1][1] t[0][1][2] t[0][1][3]
t[0][2][0] t[0][2][1] t[0][2][2] t[0][2][3]
t[1][0][0] t[1][0][1] t[1][0][2] t[1][0][3]
t[1][1][0] t[1][1][1] t[1][1][2] t[1][1][3]
t[1][2][0] t[1][2][1] t[1][2][2] t[1][2][3]
```

Tablice wielowymiarowe inicjuje się podobnie jak jednowymiarowe: każdy wiersz tablicy dwuwymiarowej inicjowany jest odpowiednią podlistą wartości początkowych.

```
int sum[5], nrow=2, ncol=5, i, j ;
int tab[2][5]={ {1, 2, 3, 4, 5},
                {5, 4, 3, 2, 1} } ;
;
for(i=0 ; i<ncol ; i++)
  sum[i]=tab[0][i]+tab[1][i] ;
;
for(i=0 ; i<nrow ; i++)
  for(j=0 ; j<ncol ; j++)
    tab[i][j]+=10 ;
```

Poniżej przytaczam kilka ciekawych wyrażeń z elementami tablic oraz operatorem zwiększania. Niekiedy kompilator zachowuje się nie tak, jak byśmy tego oczekiwali. Dlaczego?

```
#define NT 50
;
int i=1 ;
float ftab[NT] ;
double dtab[NT][NT] ;
;
ftab[i]=i++ ;   // ftab[i]=i ; i++ ;
;
ftab[i++]=i ;
// *(ftab+i++)=i ;
// ftab[i]=i ; i++ ;
// why not: ftab[i]=(i++,i) == ftab[i]=++i ;
;
ftab[i]=++i ;
// *(ftab+i)=++i ;
//ftab[i]=i+1 ; ++i ;
;
ftab[++i]=i ;
// *(ftab+(++i))=i ;
// ++i ; ftab[i]=i ;
;
ftab[i++]=i++ ;
// *(ftab+i++)=i++ ;
// ftab[i]=i ; i+=2 ;
// why not: ftab[i]=(i++,i) ; i++ ;
;
ftab[++i]=++i ;
// ++i ; ftab[i]=++i ;
// ++i ; ftab[i]=i+1 ; ++i ;
;
ftab[i++]=++i ;
// ftab[i]=++i ; i++ ;
// ftab[i]=i+1 ; i+=2 ;
// why not: ftab[i]=(i++,++i) ;
;
ftab[++i]=i++ ;   // ++i ; ftab[i]=i ; i++ ;
;
dtab[i][i]=i++ ;  // dtab[i][i]=i ; i++ ;
dtab[i++][i]=i ;
// *(*(dtab+i++)+i)=i ;
// why not: dtab[i][i+1]=i+1 ; i++ ;
dtab[i][i++]=i ;
// *(*(dtab+i)+i++)=i ;
// why not: dtab[i][i]=i+1 ; i++ ;
;
dtab[++i][i]=i ;  // ++i ; dtab[i][i]=i ;
```

```
;
dtab[i][++i]=i ;  // dtab[i][i+1]=i+1 ; i++ ;
;
dtab[i][i]=++i ;  // dtab[i][i]=i+1 ; i++ ;
;
dtab[i++][i++]=i ;
// dtab[i][i]=i ; i+=2 ;
// why not: dtab[i][i+1]=i+2 ; i+=2 ;
;
dtab[i++][++i]=i ;
// dtab[i][++i]=i ; i++ ;
// dtab[i][i+1]=i+1 ; i+=2 ;
// why not: dtab[i][i+2]=i+2 ; i+=2 ;
;
dtab[++i][++i]=i ;
// dtab[i+1][i+2]=i+2 ; i+=2 ;
;
dtab[++i][i++]=i ;
// ++i ; dtab[i][i]=i ; i++ ;
;
dtab[i++][i]=++i ;
// dtab[i][i]=i+1 ; i+=2 ;
;
dtab[++i][i]=++i ;
// i++ ; dtab[i][i]=i+1 ; i++ ;
;
dtab[++i][++i]=++i ;
// i++ ; dtab[i][i+1]=i+2 ; i+=2 ;
;
dtab[i++][i++]=i++ ;
// dtab[i][i]=i ; i+=3 ;
```

Dlaczego np. poniższe instrukcje w parach nie są sobie równoważne? Operatory przedrostkowe działają zgodnie z oczekiwaniami, tzn. zawsze zwiększają argument o `1` przed jego użyciem. Natomiast operatory przyrostkowe działają dopiero po złożeniu operacji przypisania. Przyznaję, że zupełnie nie rozumiem.

```
ftab[i]=++i ;  // ftab[i]=i+1 ; i++ ;
ftab[i++]=i ;  // ftab[i]=i ; i++ ;
;
dtab[i][++i]=i ;  // dtab[i][i+1]=i+1 ; i++ ;
dtab[i++][i]=i ;  // dtab[i][i]=i ; i++ ;
;
dtab[i][i]=++i ;  // dtab[i][i]=i+1 ; i++ ;
dtab[i][i++]=i ;  // dtab[i][i]=i ; i++ ;
```

Zadanie: Do czego prowadzi wykonanie pętli `for` w przykładzie poniżej.

```
const int NT=10 ;
int i ;
float ftab[NT] ;
;
for(i=0 ; i<NT ; ftab[i++]=i+1) ;
;
for(i=0 ; i<NT ; ftab[++i-1]=i) ;
;
for(i=0 ; i<NT ; i++)
  ftab[++i-1]=i-- ;
;
for(i=1 ; i<=NT ; ftab[--i]=i+++i++) ;
```

Inicjowanie tablic

Wskaźniki

Wskaźnik jest zmienną, której wartością jest adres pewnego obiektu, czyli np. innej zmiennej lub funkcji.

Zatem wskaźnik jest zwykłą zmienną, do której można odwołać się poprzez jej adres, ale także wartość tej zmiennej jest pewnym adresem.

Uwaga! Łatwo stworzyć wskaźnik, który wskazuje na niewiadmo co. Albo operować wskaźnikiem, któremu uprzednio nie przydzielono pamięci.

Zmienną wskaźnikową deklaruje się za pomocą symbolu `*` umieszczonym bezpośrednio przed nazwą zmiennej.

```
int *pint ;
long int *plongint ;
float *pfloat ;
double *pdouble ;
```

W powyższym przykładzie wszystkie wskaźniki, bez względu na typ obiektu na jaki wskazują, mają taki sam rozmiar, tzn. zajmują taką samą, ustaloną dla danej implementacji języka i/lub maszyny, ilość bajtów pamięci. Różnią się one jednak typem, a więc wskazują także na obiekty odmiennych typów. Np. wskaźnik `pint` może zostać skojarzony tylko ze zmienną typu `int`.

Zatem: Od wskaźnika wymaga się wskazywania obiektów ustalonego typu, takiego dla którego wskaźnik został zadeklarowany.

Rozmiar zmiennej wskaźnikowej zwykle jest równy rozmiarowi zmiennej całkowitej typu `int`, a więc w `djgpp` wynosi 4 bajty.

```
int size ;
;
size=sizeof(char) ;      // 1
size=sizeof(int) ;       // 4
size=sizeof(double) ;    // 8
;
size=sizeof(char *) ;    // 4
size=sizeof(int *) ;     // 4
size=sizeof(double *) ;  // 4
```

Dalej, określenie 'wskaźnik do obiektu typu ...' będziemy skracać do postaci 'wskaźnik typu ...'. Np. zamiast 'wskaźnik do liczb całkowitych typu `int`' będziemy używać skrótów 'wskaźnik całkowity' lub 'wskaźnik typu `int`'.

Uwaga: Każdy wskaźnik jest adresem, ale nie każdy adres jest wskaźnikiem!

Operatory wyłuskania i referencji (adresu)

Operator wyłuskania `*` zastosowany do zmiennej wskaźnikowej zwraca poprzez adres przechowywany przez ten wskaźnik wartość ukrytą pod tym adresem.

Operator referencji (adresu) `&` podaje adres obiektu. Można go stosować tylko do zmiennych i elementów tablic.

```
int x=1, y, *px ;
;
px=&x ;
```

```
y=*px ;
;
y=x ;
```

Początkowo, jak każda niezainicjowana zmienna, wskaźnik px zawiera jako swoją wartość sieczkę (tzn. losową zawartość pamięci). Podobnie jest ze zmienną y. Wartości obu zmiennych, px oraz y, są nieokreślone. Z tym że wartość px jest traktowana jako adres, a wartość y jako liczba całkowita.

Instrukcja `px=&x;` przypisuje zmiennej wskaźnikowej px adres, pod którym w pamięciu znajduje się wartość zmiennej x. Zatem teraz px wskazuje na zmienną x.

Instrukcja `y=*px;` przypisuje zmiennej y zawartość obiektu wskazywanego przez wskaźnik px, a więc wartość zmiennej x.

Ciąg obu tych instrukcji przypisuje zatem w pośredni sposoób zmiennej y wartość zmiennej x, co jest równoważne instrukcji `y=x;`.

Jednoargumentowe operatory * i & wiążą silniej niż operatory arytmetyczne i inne. Ponadto, działają na argument od strony prawej.

W ogólności, jak to zobaczymy później, operator wyłuskania można poprawnie zastosować do dowolnego wyrażenia będącego adresem. Poniżej tylko prosty przykład.

```
int i=5, j, *p ;
;
j=*&i ;      // equivalent to
j=*(&i) ;
j=i ;
;
p=&*&i ;
```

Warto zwrócić uwagę na to, że złożenia operatorów: adresu (&) i wyłuskania (*), czyli &* oraz *& są operatorami tożsamościowymi (jedynkowymi). A więc jeśli x jest obiektem, to `*&x==x`, a gdy x jest adresem obiektu, to `&*x=x`.

Zmienne wskaźnikowe mogą oczywiście występować w wyrażeniach.

```
int x=1, *px ;
;
px=&x ;
x=*px+1 ;    // x=(*px)+1 ;
             // x=*(px+1) ;
*px=33 ;
*px+=7 ;
(*px)++ ;    // *px++ ;
*p*=2**p ;   // (*p)*=2*(*p) ;
             // x*=2*x ;
```

Wskaźnikami można posługiwać się podobnie, jak innymi zmiennymi.

```
int x=5, *px, *py ;
;
px=&x ;
py=px ;
;
(*px)++ ;    // x++ ;
(*py)-- ;    // x-- ;
;
*py=*px+5 ;  // x+=5 ;
;
x=*px==*py&&*px==x ;
// x=(*px==*py)&&(*px==x) ;
```

W powyższym przykładzie py wskazuje na to samo co i px, a więc na x. Zatem wszystkie operacje zarówno na *px, jak i *py będą prowadziły do zmian wartości zmiennej x.

Zauważmy, że zarówno px, jak i *px są l-wartościami.

Wskaźniki a tablice

W języku C istnieje ścisła zależność między wskaźnikami i tablicami. Każda operacja, która może być wykonana przez indeksowanie tablicy, może być również dokonana za pomocą wskaźnika. Wersja wskaźnikowa jest szybsza.

```
int a[10], *pa, x ;
;
pa=&a[0] ;        // [] &
x=*pa ;           // x=a[0] ;
```

Powyżej pa jest wskaźnikiem do obiektów całkowitych, a a dziesięcioelementową tablicą obiektów tego samego typu.

Instrukcja `pa=&a[0];` ustawia wskaźnik pa tak, aby wskazywał na pierwszy element (a[0]) tablicy a. To znaczy, że wartością pa będzie adres do a[0].

Przypisanie `x=*pa;` skopiuje zawartość, na którą wskazuje pa, a więc a[0], do x.

Niech a jest tablicą obiektów określonego typu, a pa wskaźnikiem do obiektów tego samego typu. Wtedy, jeśli pa wskazuje na pewien element tablicy a, to z definicji pa+1 wskazuje na element następny. Ogólnie pa-i wskazuje na i-ty element przed, a pa+i na i-ty element po pa.

Innymi słowy:

type-specifier `*pa, a[] ;`
`pa==&a[i]` ⟹ `pa+1==&a[i+1]`
`pa==&a[i]` ⟹ `pa+j==&a[i+j]`
`pa==&a[i]` ⟹ `pa-j==&a[i-j]`

Zadanie: Udowodnić metodą indukcji naturalnej powyższe twierdzenie.

Jeśli pa wskazuje na a[0], to *(pa+1) jest równe wartości a[1], pa+i oznacza adres a[i], a *(pa+i) jest wartością a[i].

Innymi słowy:

type-specifier `*pa, a[] ;`
`*pa==a[0]` ⟹ `*(pa+1)==a[1]`
`*pa==a[0]` ⟹ `*(pa+i)==a[i]`
`*pa==a[i]` ⟹ `*(pa+j)==a[i+j]`

Zadanie: Udowodnić metodą indukcji naturalnej powyższe twierdzenie.

Zauważmy, że twierdzenie powyższe wynika bezpośrednio z poprzedniego, jeśli zastosujemy do niego lewostronnie operator wyłuskania *.

Uwaga: W rzeczywistości kompilator zawsze zamienia odwołanie do tablicy (jednowymiarowej) na adres jej pierwszego elementu.

```
int a[10], *pa, x ;
;
pa=&a[0] ;
pa=a ;
;
x=*a ;   // x=a[0] ;
;
pa=&a[4] ;
*pa=5 ;  // a[4]=5 ;
```

W powyższym przykładzie instrukcje: `pa=&a[0];` oraz `pa=a;` są sobie równoważne, ponieważ nazwa tablicy jest po prostu adresem jej pierwszego elementu. Ponadto, `*a` jest wartością pierwszego elementu tablicy `a`.

```
int a[5], i ;
;
i=a==&a[0] ;  // i=1 ;
// a==&a[0] is true for every table
```

Uwaga: Wyrażenia typu `a[i]` ***są zawsze przekształcane na*** `*(a+i)` ***– obie formy są całkowicie równoważne. Zatem, odwołania do tablicy poprzez indeks można zapisać za pomocą adresu i przesunięcia.***

```
float x[20], *px ;
int i, n=sizeof(x)/sizeof(float) ;
;
px=x ;
for(i=0 ; i<n ; i++)
  {
  x[i]=i*i ;    // or
  *(x+i)=i*i ;  // or
  px[i]=i*i ;   // or
  *(px+i)=i*i ;
  }
```

Bez względu na to czy `v` jest wskaźnikiem, czy też nazwą tablicy, zawsze moża doń stosować zarówno operator wyłuskania `*` jak i operator indeksowania `[]`. Obie konstrukcje są sobie całkowicie równoważne.

```
v[i] == *(v+i) ;
```

Uwaga: Pomimo tylu podobieństw, istnieje jednak istotna różnica między tablicą a wskaźnikiem. Wskaźnik jest zmienną, więc może pojawiać po lewej stronie operatora przypisania. Natomiast nazwa tablicy jest stałą, a nie zmienną. Zatem może wystąpić jedynie po prawej stronie operatora przypisania.

```
int a[10], *pa ;
;
pa=a ;
;
*pa++=10 ; // ++ * ; a[0]=10, pa++ ;
// *pa=10, pa++ ;
;
*++pa=20 ; // ++ * ; pa++, a[2]=20 ;
// pa++, *pa=20 ;
;
a=pa ;  // error
a++ ;   // error
```

Tablica jest obiektem statycznym, tzn. zajmuje ustalony obszar pamięci. Elementy tablicy są zwykłymi zmiennymi, których adres można pobrać, ale nie można go zmienić. Zmienić można jedynie wartość takiego elementu. Podobnie nazwa tablicy, jako adres pierwszego elementu, jest wartością stałą.

Arytmetyka na adresach

Operowanie wskaźnikami pozwala na bezpośrednie działanie na poszczególnych elementach pamięci poprzez ich adresy.

Cała arytmetyka na wskaźnikach opiera się na tym, że przyrost jest skalowany przez rozmiar pamięci zajmowanej przez wskazywany obiekt. Zatem w wyrażeniu `pa+i` (`pa` jest wskaźnikiem, a `i` liczbą całkowitą), wartość `i` jest najpierw mnożona przez rozmiar obiektu wskazywanego przez `pa`, a dopiero potem dodawana do wartości `pa`.

Jeśli `p` jest wskaźnikiem, to po wykonaniu operacji `p++` wskazuje on na następny element pamięci o rozmiarze określonym przez typ wskaźnika.

Uwaga: Język C gwarantuje, że żaden wskaźnik poprawnie wskazujący na jakąś daną nie będzie zerem. Zatem wartość zero może zostać użyta do zasygnalizowania jakiegoś błędu, np. braku pamięci podczas przedzielania jej wskaźnikowi (więcej o tym dalej).

Wskaźniki można porównywać. Jeśli `p` i `q` wskazują na elementy tej samej tablicy, to relacje i przyrównania działają poprawnie. Np. `p<q` jest prawdą, o ile `p` wskazuje na element wcześniejszy niż `q`. Zawsze można także przyrównać wskaźnik do `0`.

Wskaźniki (`p`) można także dodawać do liczb całkowitych (`i`). Co oznacza `p+i` była już mowa wcześniej.

Poprawne jest także odejmowanie wskaźników: jeśli `p` i `q` wskazują na elementy tej samej tablicy, to `p-q` jest liczbą elementów pomiędzy `p` i `q`.

```
#define NX 20
int i ;
float x[NX] ;
float *p=x, *q=0 ;  // init
;
if(p && !q) printf(" OK.");  // or
if(p!=0 && q==0) printf(" OK.");
;
q=p+3 ;  // *q=x[3] ;
if(p<q) printf(" OK.");
;
p+=10 ; q=x ; i=2 ;
if(p+i<=x+NX) *q+=i ;
;
i=p-q ;
```

Uwaga: Lepiej nie porównuj wskaźników odwołujących się do różnych tablic, bo zginiesz marnie!

Także wszystkie inne operacje na wskaźnikach, poza wymienionymi wyżej, są niepoprawne.

W zasadzie wskaźniki mogą być przypisywane zmiennym całkowitym (typu `int`) i z powrotem bez zmiany wartości. Prowadzi to niestety do zbyt dużej swobody w przypadku funkcji zwracających wskaźnik. Dla niektórych maszyn może to prowadzić do błędnego działania programu.

```
int n ;
double x=3, *p=&x, *q ;
;
n=(int)p ;
n-=(int)(p+=5) ;
n=-n/sizeof(double) ;  // n=5
;
q=p+8 ;
n=q-p ;                // n=8
```

Uwaga: Zapamiętaj, że wskaźniki nie są obiektami całkowitymi.

Operator indeksowania

Możemy teraz podać pełną definicję operatora indeksowania `[]`. Wymaga on dwóch argumentów: przed nawiasami wyrażenia będącego adresem (***expr-ptr***), a w nawiasach wyrażenia dającego w wyniku liczbę całkowitą (***expr-int***).

expr-ptr [***expr-int***]

Z definicji powyższy zapis jest równoważny

*(***expr-ptr***+***expr-int***)

Skoro ***expr-ptr*** ma być adresem, może zatem być dowolnym ale także poprawnym wyrażeniem zawierającym wskaźnik lub nazwę tablicy.

Jeśli `a` jest tablicą albo wskaźnikiem, a `i` liczbą całkowitą, to np.

```
(a+3)[2*i+1] == *(a+3+2*i+1) ==
*(a+2*i+4)   == a[2*i+4]
```

Należy tu dodać, że operator indeksowania jest symetryczny w sensie takim, że zamiana miejscami wyrażeń ***expr-ptr*** i ***expr-int*** daje dokładnie taki sam wynik, ponieważ zgodnie z definicją powyżej

expr-int [***expr-ptr***] ==
*(***expr-int***+***expr-ptr***) ==
*(***expr-ptr***+***expr-int***) ==
expr-ptr [***expr-int***]

Zatem jeśli `a` jest tablicą, to np. wyrażenie `a[2]` jest równoważne `2[a]`, itd.

```
int a[5] ;
;
2[a]=3 ;        // a[2]=3 ;
;
(2+1)[a]=2 ;    // a[3]=2 ;
;
(5-2)[a+1]=6 ; // a[4]=6 ;
```

Poniżej inne przykłady.

```
int i=0 ;
float *p, v ;
float tab[]={1.1,2.2,3.3,4.4,5.5,6.6} ;
;
p=tab ;
;
v=(3+tab)[i-2] ; // v=tab[1] ;
;
v=p++[i++] ; // v=tab[0] ; p=tab+1 ; i=1 ;
// v=*(p+++i++) ;
;
v=++p[--i] ;
   // i=0 ; p==tab+1 ; tab[1]++ ; v=tab[1] ;
// v=++(p[--i]) ;
// v=++*(p+--i) ;
;
v=(++p)[i++] ; // p=tab+2 ; v=tab[2] ; i=1 ;
// v=*(++p+i++) ;
;
v=(--p)[i--]++ ;
   // p=tab+1 ; v=tab[2] ; i=0 ; tab[2]++ ;
// v=((--p)[i--])++ ;
// v=(*(--p+i--))++ ;
;
v=(p+=2)[i+=2] ;
   // p=tab+3 ; i=2 ; v=tab[5] ;
// v=*((p+=2)+(i+=2)) ;
;
p=tab ; i=1 ;
;
v=p++[i] ; // v=tab[1] ; p=tab+1 ;
// v=p[i] ; p++ ;
// v=*(p+++i) ;
;
v=p++[i]++ ;
   // v=tab[2] ; tab[2]++ ; p=tab+2 ;
// v=p[i]++ ; p++ ;
// v=(*(p+++i))++ ;
;
v=++p++[i] ;
   // tab[3]++ ; v=tab[3] ; p=tab+3 ;
// v=++(p++[i]) ;
// v=++(p++)[i] ;
// v=++p[i] ; p++ ;
// v=++*(p+++i) ;
;
v=++p++[i++] ;
   // tab[4]++ ; v=tab[4] ; p=tab+4 ; i=2 ;
// v=++(p++[i++]) ;
// v=++(p++)[i++] ;
// v=++p[i] ; p++ ; i++ ;
// v=++*(p+++i++) ;
;
v=++p--[--i] ;
   // i=1 ; tab[5]++ ; v=tab[5] ; p=tab+3 ;
// i-- ; v=++p[i] ; p-- ;
// v=++*(p--+--i) ;
;
v=p--[i--]++ ;
   // v=tab[4] ; tab[4]++ ; p=tab+2 ; i=0 ;
// v=p[i]++ ; p-- ; i-- ;
// v=(*(p--+i--))++ ;
```

```
;
// v=--++p[++i] ;   // error: invalid lvalue
// v=--(++p[++i]) ; // error: invalid lvalue
;
v=--(++p)[++i] ;
   // p=tab+3 ; i=1 ; tab[4]-- ; v=tab[4] ;
// p++ ; i++ ; v=--p[i] ;
// v=--*(++p+(++i)) ;
;
v=++(++p)[i--] ;
   // p=tab+4 ; tab[5]++ ; v=tab[5] ; i=0 ;
// p++ ; v=++p[i] ; i-- ;
// v=++*(++p+i--) ;
```

Zadanie: Rozważyć poniższy program. Jakie będą wartości poszczególnych elementów tablicy `a`? Na co będą wskazywać zmienne wskaźnikowe `p` i `q`?

```
int a[]={1,2,3,4,5,6,7}, *p, *q ;
;
p=a+3 ; q=p-1 ;  // p=a+3 ; q=a+2 ;
;
q--[1]=*++p-5 ;
;
// p++ ; q[1]=*p-5 ; q-- ;
// p=a+4 ; a[3]=a[4]+3 ; q=a+1 ;
```

Stałe napisowe a wskaźniki

Stała napisowa jest z definicji tablicą znakową. W języku C oznacza ona zatem adres pierwszego znaku napisu.

```
char *text, tab[]="Let's begin..." ;
;
text="Let's go..." ;
tab="Let's go..." ;  // error
```

Jak łatwo się domyślić instrukcja powyżej przypisuje zmiennej `text` adres pierwszego znaku stałej napisowej. Nie jest to kopiowanie tekstu stałej. W języku C nie ma operatorów do obsługi tekstów traktowanych jako całość.

```
char *s, *t, buf[64], text[]="something", i ;
;
for(i=0 ; buf[i]=text[i] ; i++) ;
;
s=buf ; t=text ;
while(*s++=*t++) ; // or
for( ; *s=*t ; s++, t++) ;
; // length of text:
s=text ;
while(*s++) ;
i=--s-text ;
printf(" text contains %d characters.", i);
```

Stała napisowa jest adresem, więc moża doń stosować zarówno operator wyłuskania `*`, jak i operator indeksowania `[]`.

```
char c ;
;
c=*"text" ;        // 't'
c=*"text"+2 ;      // 'v'
c=*("text"+2) ;    // 'x'
c=*("text"+2)-5 ;  // 's'
;
c="text"[0] ;      // 't'
c="text"[0]+2 ;    // 'v'
c="text"[2] ;      // 'x'
c="text"[2]-5 ;    // 's'
```

W przykładzie poniżej udawadniamy, że wszystkie wystąpienia takiego samego napisu są odowłaniem do tej samej tablicy znakowej. Była o tym mowa już wcześniej.

```
char *pc, c ;
;
pc="text" ;
pc+=2 ;
c=pc-"text" ;
printf(" %d %d", c, *pc);
pc-=2 ;
printf(" %d %d", (int)pc, (int)"text");
```

Poniżej jeszcze jeden przykład.

```
while(*"a"<=*"z") printf(" %c", (*"a")++);
printf("\n");
while(*"A"<=*"Z") printf(" %c", (*"A")++);
```

Skoro możemy pobrać adres dowolnego znaku stałej napisowej, to możemy także zmienić jego wartość. Ta własność napisów sprawia, że tak na prawdę nie są one stałymi! Zmiany takie prowadzą do zabawnych (i trochę niebezpiecznych dla początkującego programisty) wyników. Co ciekawe z niektórymi całkowicie poprawnymi wyrażeniami kompilator `djgpp` zupełnie sobie nie radzi. Może wręcz wprawić go w zakłopotanie, po którym odmawia współpracy!

```
char *pc ;
;
printf(" text == %s\n", "text");
;
pc="text" ;
*pc='z' ;
printf(" <%s> <%s>\n", "text", pc);
;
"text"[2]='d' ;
printf(" <%s> <%s>\n", "text", pc);
;
// strange case: why in this way?
"text"[2]='s' ;
"text"[2]="text"[2]+2 ;
"text"[2]+=2 ;
printf(" <%s> <%s>\n", "text", pc);
// <zezt> <zezt>
;
"text"[2]='s' ;
pc[2]+=2 ;
printf(" <%s> <%s>\n", "text", pc);
;
*"text"='a' ;
++*"text" ;
// ++"text"[0] ;  // djgpp aborts!
printf(" <%s> <%s>\n", "text", pc);
;
++*("text"+3) ;
printf(" <%s> <%s>\n", "text", pc);
;
```

```
*("text"+2)+=2 ;
// "text"[2]+=2 ; // does not work properly
printf(" <%s> <%s>\n", "text", pc);
;
printf(" text == %s\n", "text");
```

I oto największe z możliwych osiągnięcie programisty! Po napotkaniu instrukcji (zupełnie poprawnych)

```
"text"[0]++ ;
++"text"[0] ;
```

kompilator `djgpp` rozkłada ręce, twierdząc, że nie rozumie zapisu i żegna nas komunikatem `Abort!`. Co ciekawe, powyższe wyrażenia zapisane za pomocą operatora wyłuskania `*` rozumiane są przez kompilator poprawnie!

Rzutowanie wskaźników

Wskaźniki jako zmienne można rzutować za pomocą operatora `()` do wskaźników innego typu. W wyniku takiej operacji wartość wskaźnika (adres) pozostaje oczywiście taka sama, zmienia się jednak typ obiektu (rozmiar pamięci) na jaki wskaźnik będzie wskazywał.

```
char *pc, c='a', tabc[8]="1234567" ;
double *pd, d=111.0 ;
;
pc=(char *)&d ;  // pc[8]
pd=&d ;
pc=(char *)pd ;
;
pd=(double *)&c ; // problems
;
pd=(double *)tabc ; // or
pd=(double *)&tabc[0] ;
```

Uwaga: Ostrożnie z rzutowaniem wskaźnika 'szerszego' do wskaźnika 'węższego'.

Przykład: odczyt i zapis sygnału z obrazu FITS

Obrazy uzyskane z kamery CCD przechowuje się często w plikach formatu FITS. Plik taki składa się z dwóch części: nagłówka i dwuwymiarowej mapy binarnej rozkładu oświetlenia. Nagłówek zawiera kilka istotnych informacji o sposobie zapisu mapy binarnej, a więc parametr `bitpix`, który określa sposób kodowania wartości strumienia w każdym pikselu, parametry `bscale` i `bzero`, które definiują liniowe przekształcenie między wartością zapisu a sygnałem, oraz rozmiar obrazka, określony przez ilości wierszy (`nrows`) i kolumn (`ncols`).

Uwaga: Format FITS wymaga aby wartość sygnału zapisywana była zaczynając od najbardziej znaczącego bajtu. Zatem, pod DOS'em zarówno przy odczycie jak i przy zapisie należy odwracać kolejność bajtów.

Odczyt syngału:

```
char cp[4], c ;
short int *pint, bitpix ;
float *pfloat, pixval, bscale, bezero ;
FILE *file ;
;
file=fopen("image.fits","rb") ;
// header reading: bitpix, bscale, bzero ;
;
switch(bitpix)
  {
  case(-32):
    {
    fread(cp,sizeof(float),1,file);
    c=cp[0] ; cp[0]=cp[3] ; cp[3]=c ;
    c=cp[1] ; cp[1]=cp[2] ; cp[2]=c ;
    pfloat=(float *)&cp[0] ;
    pixval=(float)(*pfloat)*bscale+bzero ;
    break ;
    }
  case( 16):
    {
    fread(cp,sizeof(short int),1,file);
    c=cp[0] ; cp[0]=cp[1] ; cp[1]=c ;
    pint=(short int *)&cp[0] ;
    pixval=(float)(*pint)*bscale+bzero ;
    break ;
    }
  default : pixval=0.0 ; break ;
  }
```

Zapis sygnału:

```
char *cp, c ;
short int vint, *pint, bitpix, wrtidx=0 ;
float *pfloat, pixval, bscale, bezero ;
double MaxSInt=(double)((0x1<<15)-1) ;
FILE *file ;
;
file=fopen("image.fits","wb") ;
// header writing ;
;
switch(bitpix)
  {
  case( 16):
    {
    pixval=(pixval-bzero)/bscale ;
    pixval=intconv(pixval) ;
    if(pixval>MaxSInt)
      { pixval=MaxSInt ; wrtidx++ ; }
    if(pixval<-MaxSInt-1)
      { pixval=-MaxSInt-1 ; wrtidx++ ; }
    vint=(short int)pixval ;
    cp=(char *)&vint ;
    c=cp[0] ; cp[0]=cp[1] ; cp[1]=c ;
    pint=(short int *)&cp[0] ;
    fwrite(pint,sizeof(short int),1,file);
    break ;
    }
  case(-32):
  default  :
    {
    bitpix=-32 ;
    pixval=(pixval-bzero)/bscale ;
    cp=(char *)&pixval ;
    c=cp[0] ; cp[0]=cp[3] ; cp[3]=c ;
    c=cp[1] ; cp[1]=cp[2] ; cp[2]=c ;
    pfloat=(float *)&cp[0] ;
    fwrite(pfloat,sizeof(float),1,file);
    break ;
    }
```

```
}
```

Wskaźnik uniwersalny

Wskaźnik uniwersalny `void *` stosuje się wówczas, gdy typ wskazywanego obiektu albo nie jest znany, albo może się zmieniać zależnie od okoliczności. Zatem wskaźnikiem takim można wskazywać obiekty dowolnych typów danych.

```
int i=5, *pi=0 ;
char *pc=0, *text="napis" ;
void *pv ;
;
pv=pi ;
pv=pc ;
;
pv=&i ;
printf(" %d", *pv);  // error
;
pv=text ;
pc=(char *)pv ;
;
pi=&i ;
pv=pi ;
pc=(char *)pv ;
printf(" %s", pc);  // problems
```

Jednakże nie można wprost wyłuskać wartości adresowanej przez wskaźnik typu `void *` (z braku informacji o typie danej). Błędne są zatem wyrażenia typu `*pv`. Wskaźnik typu `void *` musi zostać najpierw przekształcony na wskaźnik do konkretnego typu danych.

W zasadzie, w C++ nie ma zdefiniowanego niejawnego przekształcenia wskaźnika typu `void *` na wskaźnik do typu innego. Dlatego należy robić to jawnie poprzez rzut.

Dynamiczny przydział pamięci

Wskaźnikom można przydzielać pamięć, tak aby wskazywały na tablicę o zadanej ilości elementów danego typu. Służą do tego funkcje ze standardowej biblioteki języka C.

```
#include<stdlib.h>
int np=30 ;
float *p ;
p=(float *)malloc(np*sizeof(float)) ; // or
p=(float *)calloc(np,sizeof(float)) ;
;
free(p);
```

Funkcje `malloc()` oraz `calloc()` zwracają adres do obszaru pamięci mogący pomieścić zadaną ilość (`np`) obiektów określonego rozmiaru lub `0`, jeśli jest to niemożliwe (np. z braku pamięci). Zwracany adres powinien zostać zrzutowany do wskaźnika do właściwego typu danych.

Funkcja `calloc()` inicjuje ponadto wszystkie elementy przydzielonej pamięci zerami.

Pamięć przydzieloną wskaźnikowi zwalnia się za pomocą funkcji `free()`.

Uwaga: Poważnym błędem jest zwalnianie za pomocą funkcji `free()` ***pamięci, która uprzednio nie została zarezerwowana przez funkcje*** `malloc()` ***lub*** `calloc()`.

Tablice wskaźników

Wskaźniki są zmiennymi, zatem istnieją również tablice wskaźników. Tablicę taką deklaruje się następująco:

```
#define NTAB 10
float *tabp[NTAB] ;  // [] *
float *(tabp[NTAB]) ;
```

Pamiętając o tym, że operator `[]` ma większy priorytet od operatora `*`, instrukcje powyższe definiują tablicę `tabp` o `NTAB` elementach, z których każdy jest wskaźnikiem do obiektów typu `float`.

```
int i=2, j=3 ;
float *tabp[10], t[30], *p, val=3.3 ;
;
*tabp=&val ;   // or
tab[0]=&val ;
;
p=&val ;
*tabp=p ;      // float * ; or
tabp[0]=p ;
;
tabp[i]=p ;    // or
*(tabp+i)=p ;
;
val=**tabp ;     // or
val=*tabp[0] ;   // or
val=tabp[0][0] ;
;
tabp[i]=t ;
;
val=tabp[i][j] ;     // or
val=*(tabp[i]+j) ;   // or
val=(*(tabp+i))[j] ; // or
val=*(*(tabp+i)+j) ;
;
tabp[i]++ ;
++tabp[i] ;
;
*tabp++ ;      // error
(*tabp)++ ;
;
++*tabp ;
val=*++*tabp ;
;
**tabp++ ;     // error
(**tabp)++ ;
;
++**tabp ;
```

W przykładzie powyżej elementy tablicy `tabp` są wskaźnikami do obiektów typu `float`. Zatem dowolny z tych elementów, np. `tabp[0]` (inaczej `*tabp`), a ogólnie `tabp[i]`, można zainicjować adresem zmiennej typy `float`, tak jak w wyrażeniu `tabp[i]=&val`.

Ponadto, z tego samego powodu, pod dowolny element tablicy `tabp` można podstawić dowolny wskaźnik do obiektu typu `float`, tak jak w wyrażeniu `tabp[i]=p`. Odwrotne podstawienie oczywiście też jest dozwolone.

Ponadto, z tego samego powodu, pod dowolny element tablicy `tabp` można podstawić nazwę tablicy obiektów typu `float`, czyli adres jej pierwszego elementu, tak jak w wyrażeniu `tabp[i]=t` (inaczej `tabp[i]=&t[0]`).

Zobaczmy dokładnie, co oznacza wyrażenie `**tabp`. Jest ono równoważne

```
**tabp == *(*tabp) == *(*(tabp+0)) ==
*(tabp[0]) == *tabp[0].
```

Zatem `**tabp` jest po prostu wyłuskaniem wartości obiektu, na który wskazuje pierwszy element (`tabp[0]`) tablicy `tabp`.

Natomiast dla wyrażenia `tabp[i][j]` mamy

```
tabp[i][j] == *(tabp[i]+j).
```

Czyli `tabp[i][j]` oznacza obiekt przechowywany w pamięci pod adresem `(int)tabp[i]+j*sizeof(float)` (dlaczego rzut do typu `int` jest tu niezbędny; co wyrażenie to by oznaczało bez tego rzutu?). Jeśli `tabp[i]` wskazuje na `t[0]` (czyli `tabp[i]==&t[0]`), to `tabp[i][j]==t[j]`, ponieważ

```
tabp[i][j] == (&t[0])[j] == (&*(t+0))[j] ==
((t))[j] == t[j].
```

Zadanie: Rozważyć ponownie podany wyżej przykład. Co oznaczają wyrażenia

`(*(tabp+i))[j]` oraz `*(tabp+i)[j])`?

Tablice wskaźników inicjuje się tak samo jak i inne tablice poprzez podanie listy adresów. Jeśli rozmiar tablicy zostanie pominięty, to kompilator określi go sam zliczając wartości początkowe.

```
char *dayname[]={ "bad name",
                  "Dydd Llyn",
                  "Dydd Mawrth",
                  "Dydd Merchyr",
                  "Dydd Ieu",
                  "Dydd Gwener",
                  "Dydd Sadwrn",
                  "Dydd Sul"      } ;
char *pc, c ;
;
int nday=7, i=3 ;
printf(" %s\n",
  (i<1 || i>7)?dayname[0]:dayname[i]);
;
dayname[3]+=5 ;
c=*dayname[3] ;          // c='M'
p=dayname[3] ;
printf(" %s\n", p);      // "Merchyr"
;
printf(" %s\n", dayname[1]+5);  // "Llyn"
```

Uwaga na różnicę pomiędzy tablicą wskaźników a wskaźnikiem do tablic.

```
int *tabp[10] ;
int (*ptab)[10] ;
```

Powyżej `tabp` jest tablicą 10 wskaźników do liczb całkowitych. Natomiast `ptab` jest wskaźnikiem do obiektów, które są 10-elementowymi tablicami liczb typu `int`.

Przykład: ładowanie całego obrazu FITS (1)

Podamy tutaj pierwszy sposób ładowania do pamięci rozkładu strumienia z mapy binarnej obrazu FITS, za pomocą tablicy wskaźników. Ograniczymy się właściwie tylko do przydziału pamięci.

```
#define MaxNofRows 2222
;
int row, col, nrows, ncols ;
float *frame[MaxNofRows] ;
;
if(nrows>=MaxNofRows)
  {
  printf("Err: Too many rows! [max=%d]",
    MaxNofRows-1);
  return(0);
  }
for(row=0 ; row<nrows ; row++)
  {
  frame[row]=
    (float *)calloc(ncols,sizeof(float)) ;
  if(!frame[row])
    {
    printf("Err: No memory for row no %d!",
      row);
    return(0);
    }
;
for(col=0 ; col<ncols ; col++)
  for(row=0 ; row<nrows ; row++)
    frame[row][col]=0.0 ;
;
for(row=0 ; row<nrows ; )
  free(frame[row++]);
```

Warto zwrócić uwagę na to, że bez względu na rozmiary obrazu FITS, nasza tablica `frame` zajmuje taką samą niewielką ilość pamięci (`MaxNofRows*sizeof(float *)`). Jesteśmy ograniczeni jedynie maksymalną ilością wierszy (zmienimy to niebawem), kolumn może być dowolnie wiele.

Uwaga: Ogromną zaletą wskaźników jest to, że za ich pomocą można tworzyć tablice dynamiczne, a więc takie, których rozmiarów z góry nie znamy.

Wskaźniki do wskaźników

Wskaźnik do wskaźnika określonego typu deklaruje się następująco:

```
int **pp, i ;
// ...
i=*(*pp) ;  // or just
i=**pp ;
```

Zmienna `pp` jest wskaźnikiem do wskaźnika do obiektu typu `int`. Zatem `pp` wskazuje na obiekty typu `int *`.

Adres zmiennej `pp` jest wartością stałą. `pp` wskazuje na wskaźnik, a więc wyłuskanie `*pp` zwraca wartość tego wskaźnika, czyli jakiś adres. Wartość wskazywaną przez `*pp` (a więc także pośrednio przez `pp`) uzyskujemy poprzez powtórne użycie operatora wyłuskania, czyli `**pp`.

Podsumowując, wartość na jaką wskazuje wskaźnik do wskaźnika uzyskujemy poprzez dwukrotne użycie operatora wyłuskania.

Zauważmy, że wskaźnik do wskaźnika wskazuje bezpośrednio na zmienne wskaźnikowe, z których każdą można skojarzyć z tablicą jednowymiarową. Zatem wskaźnik do wskaźnika może zostać użyty do konstrukcji tablic dwuwymiarowych.

```
int *pa, *pb, i=5 ;
int **pp ;       // int *(*pp) ;
;
pa=&pb ;        // error
;
pp=&i ;         // error
;
pb=&i ;
pp=&pb ;
(**pp)++ ;       // **pp=i=6 ;
**pp+=4 ;        // **pp=i=10 ;
printf("%d %d", **pp, i);
```

Powyżej przypisanie `pa=&pb` jest niepoprawne, ponieważ `pa` jest wskaźnikiem do obiektu całkowitego, a `&pb` zwraca adres zmiennej `pb`, który jest traktowany jako wskazanie na wskaźnik do obiektu typu `int`. Mamy tutaj więc do czynienia z niezgodnością typów.

Podobnie jest w przypadku instrukcji `pp=&i;`. `pp` jest wskaźnikiem do wskaźnika, natomiast wyrażenie `&i` podaje adres zmiennej `i`, więc jest podobnego typu co i wskaźnik (ale nim nie jest!).

Zatem, `pp` jest tego samego typu obiektem co i `&pa`. Ale `pp` jest zmienną, a `&pa` wartością stałą.

Wskaźniki do wskaźników podlegają takim samym zasadom arytmetyki jak każdy wskaźnik. Wskaźnik do wskaźnika `pp` wskazuje na `*pp`, który jest wskaźnikiem do obiektu określonego typu danych. Obiekt `*pp` zajmuje określony obszar pamięci. Zwiększanie (lub zmniejszanie) zmiennej `pp` o pewną liczbę całkowitą `i` powoduje, że `pp` zaczyna wskazywać na obszar pamięci przesunięty w przód (lub tył) względem pozycji początkowej o ilość bajtów równą liczbie `i` przeskalowanej rozmiarem obiektów wskazywanych przez `pp`, które jakoże są wskaźnikami zajmują zwykle 4 bajty pamięci.

```
int n ;
double x=1, *p=&x, **pp=&p, **qq ;
;
n=(int)pp ;
n-=(int)(pp+=5) ;
n=-n/sizeof(double *) ;  // n=5
;
pp=&p ;
n=(int)*pp ;
n-=(int)(*pp+=5) ;
n=-n/sizeof(double) ;    // n=5
;
qq=pp+8 ;
n=qq-pp ;                // n=8
```

Zadanie: Czy powyższy program działałby poprawnie, gdyby usunięto z niego instrukcję `pp=&p;`? Dlaczego?

Bez względu na to, na jakie dane wskaźnik do wskaźnika pośrednio wskazuje, jego przesunięcia zawsze są skalownane takim samym czynnikiem, będącym rozmiarem wskaźnika w pamięci.

Różnica pomiędzy wskaźnikiem do wskaźnika `pp` a wskaźnikiem do danej `p` polega na tym, że obszar pamięci wskazywany przez `pp+i` traktowany jest jako wskaźnik, natomiast obszar pamięci wskazywany przez `p+i` traktowany jest jako dana (np. liczba lub struktura). Zatem odpowiednie wyłuskania odnoszą się także do obszarów pamięci o różnych rozmiarach: `*(pp+i)` zajmuje zwykle 4 bajty, natomiast `*(p+i)` zajmuje określoną dla danego typu danych ilość bajtów, np. dla typu `double` 8.

Skoro `pp` wskazuje na wskaźniki, to `*pp` jest wskaźnikiem. `*pp` wskazuje na dane określonego typu. Jest to zatem zwykły wskaźnik, który także można poddać operacjom przesunięcia oraz wyłuskania, np. `(*pp)++`, `*pp+i` `(==(*pp)+i)`, `*(*pp+i)` (uwaga na nawiasy!). Co oznaczają wymienione przykłady łatwo jest się domyślić, więc nie wyjaśniam.

```
int i=5 ;
double **pp ;
;
pp++ ;
pp-- ;
pp+=i ;
;
(*pp)++ ;
*pp+=i ;
```

Powyżej operacje na zmiennej `pp` powodują 'przesunięcia' w pamięci, które są skalowane rozmiarem `sizeof(double *)==4` bajty. Natomiast operacjom na wskaźniku `*pp` towarzyszą 'przesunięcia' skalowane rozmiarem `sizeof(double)==8` bajtów.

Nazwa tablicy oznacza adres jej pierwszego elementu. Zatem nazwa tablicy wskaźników jest adresem (wskazaniem) pierwszego elementu, który z kolei jest wskaźnikiem do obiektu danego typu. Mówiąc inaczej, nazwa tablicy wskaźników jest po prostu adresem adresu, a więc wskaźnikiem do wskaźnika. Zatem nazwę tablicy wskaźników można podstawić pod wskaźnik do wskaźnika.

```
#define NR 2
#define NC 4
;
int *(pi[NR]), taba[NC], tabb[NC], i, j ;
int **pp ;
;
for(i=0 ; i<NC ; i++)
  { taba[i]=i+1 ; tabb[i]=NC+i+1 ; }
;
pp=pi ;
*pp++=taba ;  // pp[0]=taba ; pp++ ;
*pp--=tabb ;  // pp[1]=tabb ; pp-- ;
;
for(i=0 ; i<NR ; i++)
  for(j=0 ; j<NC ; j++)
    printf("%d%c", pp[i][j],
               j==NC-1?'\n':' ');
```

```
;
i=**pp ;         // 1
i=**++pp ;       // 5  pp[1][0]
pp-- ;
i=(*pp++)[1] ;  // 2  pp[0][1]
i=(*pp--)[3] ;  // 8  pp[1][3]
;
i=(*++pp)[0] ;  // 5  pp[1][0]
pp-- ;
i=*++pp[0] ;    // 2  pp[0][1]
// --*pp ;
i=*(*++pp+2) ;  // 7  pp[1][2]
i=*(*--pp+2) ;  // 4  pp[0][3]
--*pp ;
;
i=*++*pp ;      // 2  pp[0][1]
--*pp ;
;
++**pp=i ;      // error
i=++**pp ;
```

Powyżej wskaźnik do wskaźnika kojarzymy z dwoma tablicami jednowymiarowymi. Poniżej pokazujemy, jak wskaźnik do wskaźnika skojarzyć z tablicą dwuwymiarową. Zauważmy, że w obu przypadkach niezbędna jest tablica wskaźników (dlaczego?).

```
#define NR 4
#define NC 5
;
int *pi[NR], tab[NR][NC], i, j, n ;
int **pp ;
;
for(n=i=0 ; i<NR ; i++)
  for(j=0 ; j<NC ; tab[i][j++]=n++) ;
;
pp=pi ;
for(i=0 ; i<NR ; *pp++=tab[i++]) ;
pp-=i ;
;
n=--**pp ; // tab[0][0]-- ; n=tab[0][0] ;
;
n=*++*pp ; // pi[0]++ ; n=tab[0][1] ;
;
n=*++*++pp ; // ++pp == pp=&pi[1] ;
             // ++*pp == pi[1]++ ;
             // n=tab[1][1] ;
;
n=++*++*++pp ; // ++pp == pp=&pi[2] ;
               // ++*pp == pi[2]++ ;
               // tab[2][1]++ ; n=tab[2][1] ;
;
n=*(*pp--)-- ; // n=**pp == n=tab[2][1] ;
               // (*pp)-- == pi[2]-- ;
               // pp-- == pp=&pi[1] ;
n=**pp ;       // n=tab[1][1] ;
;
n=(*(*pp++)++)++ ; // n=**pp == n=tab[1][1] ;
                   // ++**pp == tab[1][1]++ ;
                   // (*pp)++ == pi[1]++ ;
                   // pp++ == pp=&pi[2] ;
n=**pp ;           // n=tab[2][0] ;
```

```
;
for(pp=pi, i=0 ; i<NR ; pp[i]=tab[i], i++) ;
;
n=pp++[1][1] ; // n=tab[1][1] ;
               // pp++ == pp=&pi[1] ;
               // **pp == tab[1][0] ;
;
n=pp[1]++[1] ; // n=tab[2][1] ;
               // pp[1]++ == pi[2]++ ==
               // == pi[2]=&tab[2][1]
               // == pi[2]=tab[2]+1 ;
               // **pp == tab[1][0]
;
n=pp++[1]++[2] ;
  // n=pp[1][2] ;
  // pp == &pi[1] == &*(pi+1) == pi+1 ;
  // pp[1] == *(pp+1) == *(pi+1+1)
  // == *(pi+2) == pi[2] ;
  //
  // pp[1][2] == (pi+1)[1][2] == *(pi+1+1)[2]
  // == pi[2][2] == (&tab[2][1])[2]
  // == (& (tab[2])[1] )[2]
  // == (& *(tab[2]+1) )[2]
  // == (tab[2]+1)[2] == *(tab[2]+1+2)
  // == *(tab[2]+3) == tab[2][3] ;
  //
  // pp[1]++ == pi[2]++
  // == pi[2]=&tab[2][2] == pi[2]=tab[2]+2 ;
  //
  // pp++ == pp=&pi[2] ;
  //
  // **pp == **&pi[2] == *(tab[2]+2)
  // == tab[2][2] ;
;
for(pp=pi, i=0 ; i<NR ; pp[i]=tab[i], i++) ;
i=1 ; j=2 ;
;
n=++pp[i]++[j] ;
  // n=++((pp[i]++)[j]) ;
  // pp[i][j]++ ; n=pp[i][j] ; pp[i]++ ;
  //
  // pp[i][j]++ == pi[i][j]++
  // == tab[i][j]++ ;
  // pp[i]++ == pi[i]++ == pi[i]=tab[i]+1 ;
  // *pi[i] == pi[i][0] == tab[i][1] ;
  //
  // pp==pi ; pi[1]==tab[1]+1 ;
  // **pp==tab[0][0] ;
;
n=(++pp)[i]++[j] ;
  // n=((++pp)[i])++[j] ;
  // pp++ ; n=pp[i][j] ; pp[i]++ ;
  //
  // pp++ == pp=pi+1 ; *pp=pi[1] ;
  // pp[i][j] == (pi+1)[i][j]
  // == (*(pi+1+i))[j] == pi[i+1][j]
  // == tab[i+1][j] ;
  //
  // pp[i]++ == pi[i+1]++
  // == pi[i+1]=tab[2]+1 ;
```

```
  //
  // pp==pi+1 ; pi[2]=tab[2]+1 ;
  // **pp==tab[1][1] ;
;
n=(++pp[i])[j] ;
  // ++pp[i] ; n=pp[i][j] ;
;
n=(++(pp--)[i])[j] ;
  // ++pp[i] ; n=pp[i][j] ; pp-- ;
;
for(pp=pi, i=0 ; i<NR ; pp[i]=tab[i], i++) ;
i=1 ; j=2 ;
;
n=++pp++[++i]++[++j] ;
  // i++ ; j++ ; pp[i][j]++ ; n=pp[i][j] ;
  // pp++ ; pp[i]++ ;
```

Zauważmy, że operacje przyrostkowego zwiększania `++` i zmniejszania `--` wykonywane są w kolejności od zewnątrz, czyli od strony prawej ku lewej, np. instrukcja

```
n=(*(*pp++)++)++ ;
```

jest równoważna ciągowi instrukcji

```
n=**pp ; ++**pp ; ++*pp ; ++pp ;
```

Podobnie

```
n=pp++[i]++[j]++ ;
```

jest równoważne

```
n=pp[i][j] ; pp[i][j]++ ; pp[i]++ ; pp++ ;
```

Zadanie: Rozważyć zmienną `pp` z ostatniego przykładu. Co oznaczają wyrażenia

```
&pp[i], (&pp)[i],
pp[i], (*pp)[i],
&pp[i][j], (&pp)[i][j], (&pp[i])[j] ?
```

Jaki jest ich typ? Pod jakiego typu zmienne można je podstawić?

Zadanie: Zamienić poniższe instrukcje na bloki instrukcji prostych.

```
int ***ppp, **pp, *p ;
;
*p++=*++*pp++ ;
++p=*++pp=*++*++ppp ;
**pp++=++*++*++*pp++ ;
++pp=++*ppp++ ;
```

Przykład: ładowanie całego obrazu FITS (2)

Podamy poniżej drugi sposób ładowania do pamięci rozkładu strumienia z mapy binarnej obrazu FITS, tym razem za pomocą wskaźnika do wskaźników. Podobnie jak poprzednio, ograniczymy się tylko do przydziału pamięci.

```
int row, nrows, ncols ;
float **frame ;
;
frame=
  (float **)calloc(nrows,sizeof(float *)) ;
if(!frame)
  { printf(" Error"); return(0); }
frame[0]=
  (float *)
  calloc(nrows*ncols,sizeof(float)) ;
if(!frame[0])
  { printf(" Error"); return(0); }
for(row=1 ; row<nrows ; row++)
  frame[row]=frame[row-1]+ncols ;
;
free(*frame);
free(frame);
```

lub

```
int row, nrows, ncols ;
float **frame ;
;
frame=
  (float **)calloc(nrow,sizeof(float *)) ;
if(!frame)
  { printf(" Error"); return(0); }
for(row=0 ; row<nrow ; row++)
  {
  frame[row]=
    (float *)calloc(ncol,sizeof(float)) ;
  if(!frame[row])
    { printf(" Error"); return(0); }
  }
;
for(row=0 ; row<nrow ; row++)
  free(frame[row]);
free(frame);
```

Jak widać, tym razem nie ma jakichkolwiek ograniczeń na rozmiary obrazu.

Wskaźniki a tablice wielowymiarowe

Ograniczymy się tutaj do tablic dwuwymiarowych.

```
int tab[10][10] ;
int *ptab[10] ;
```

Pomimo, że do elementów obu powyższych tablic możemy odwołać się tak samo, poprzez konstrukcję `[i][j]`, a więc np. `tab[3][3]` lub `ptab[3][3]`, to jednak pomiędzy obydwoma typami obiektów istnieją pewne różnice.

`tab` jest prawdziwą tablicą statyczną 100 elementów ułożonych w 10 rzędów i 10 kolumn. Każdemu elementowi przydzielono pamięć i, jeśli został właściwie zainicjowany, ma także sensowną wartość.

Natomiast `ptab` jest jedynie tablicą 10 wskaźników, z których każdy *może* wskazywać na tablicę o elementach całkowitych (dokładniej mówiąc, na jej pierwszy element). Jeśli każdy wskaźnik wskazuje poprawnie na tablicę 10-elementową, to mamy wtedy zarezerwowaną pamięć dla 100 wartości całkowitych oraz pamięć dla 10 adresów. Właściwie zainicjowana tablica wskaźników wykorzystuje zatem odrobinę więcej pamięci niż podobna tablica statyczna.

W ogólności jednak, elementy tablicy `ptab` mogą wskazywać na tablice o dowolnym rozmiarze. Ponadto rozmiary te mogą być różne. Tak jak pokazano poniżej.

```
#define NT 10 ;
int i, j, tab[NT][NT] ;
```

```
int *ptab[NT] ;
int *tmp1={1, 2, 3}, *tmp2={4, 5} ;
;
for(i=0 ; i<NT ; i++)
  for(j=0 ; j<NT ; j++) *(*(tab+i)+j)=i+j ;
// or                   tab[i][j]=i+j ;
// or                   *(tab[i]+j)=i+j ;
// or                   (*(tab+i))[j]=i+j ;
// problems:            *(tab+i)[j]=i+j ;
// the same:            *(*(tab+i+j)+0)=i+j ;
;
for(i=0 ; i<NT ; i++) ptab[i]=&tab[i][0] ;
// or                 *(ptab+i)=*(tab+i) ;
;
for(i=0 ; i<NT ; i++) ptab[i]=0 ;
;
*ptab=tmp1 ;
*++ptab=tmp2 ;
*(*ptab)+1=5 ;  // ptab[1][1]=5 ;
```

Mamy jeszcze wskaźnik do wskaźnika. Instrukcja

```
int **pptab ;
```

definuje zmienną `pptab` jako wskaźnik do wskaźnika do obiektów całkowitych. Zauważmy, że `*pptab` jest typu `int *`, a więc wskazuje na wskaźniki, które z kolei mogą stanowić odwołanie do pierwszych elementów tablic jednowymiarowych. Zatem poprzez `pptab` także możemy utworzyć tablicę dwuwymiarową: `*pptab` numerują wiersze, a `**pptab` odnoszą się do poszczególnych elementów.

Aby utworzyć tablicę 10×10 elementów musimy najpierw zainicjować poprawnie wskaźnik do wskaźnika, tak aby wskazywał na początek obszaru pamięci z 10 wskaźnikami. Następnie każdemu z tych 10 wskaźników należy przyporządkować początek obszaru pamięci mogący pomieścić 10 obiektów danego typu. Zatem, w porównaniu do tablicy statycznej o takim samym rozmiarze, obiekt nasz zajmuje `11*sizeof(int *)` bajtów pamięci więcej.

Wskaźniki do tablic

Jak pamiętamy tablicę wskaźników definiuje się następująco

```
int *tabp[3] ;
```

Zmienna `tabp` oznacza tablicę 3-ech wskaźników do obiektów typu `int`. W języku C można zdefiniować także wskaźnik do tablicy (o ustalonym z góry rozmiarze). Robi się to tak:

```
int (*ptab)[3] ;
```

Powyżej `ptab` jest wskaźnikiem do trójelementowych tablic, których elementami są liczby `int`. Zauważmy, że dodanie do wskaźnika `ptab` liczby całkowitej `i` daje w wyniku adres przesunięty względem wartości `ptab` o `i` przeskalowane rozmiarem wskazywanego obiektu, czyli `sizeof(int [3])`, czyli `3*sizeof(int)`. Zatem

```
(int)(ptab+i) == (int)ptab+i*sizeof(int [3])
```

Pamiętając o tym, że operator indeksowania `[]` ma priorytet wyższy od operatora wyłuskania `*` można tworzyć bardzo złożone obiekty wskaźnikowo-tablicowe, tak jak poniżej.

```
int **tabpp[5] ;
int (**pptab)[5] ;
int *(*ptabp)[5] ;
;
int *ttabp[3][5] ;
int (*pttab)[3][5] ;
;
int (*tabpt[3])[5] ;
```

Tak więc `tabpp` jest tablicą (5-cio elementową) wskaźników do wskaźników (do obiektów typu `int`), `pptab` jest wskaźnikiem do wskaźnika do tablic (elementów typu `int`), a `ptabp` jest wskaźnikiem do tablic, których elementami są wskaźniki do obiektów całkowitych typu `int`. `ttabp` jest dwuwymiarową tablicą wskaźników typu `int *`, a `pttab` jest wskaźnikiem do dwuwymiarowych tablic obiektów typu `int`. Natomiast `tabpt` jest tablicą 3-ech wskaźników do 5-cio elementowych tablic liczb typu `int`.

Z tego co powiedziano wyżej wynika, że podstawianie nazw tablic wielowymiarowych (także wskaźników) do odpowiednich wskaźników zawsze prowadzi do ostrzeżeń lub komunikatów o błędzie. Powodem jest odmienny typ tablic oraz wskaźników. Np. nazwa tablicy dwuwymiarowej jest typu wskazanie na tablicę (jednowymiarową) pewnych elementów, i nie jest wskazaniem na elementy, czy też wskazaniem na wskazanie na elementy. Dokładniej pokazano to w przykładzie poniżej.

```
float x[2], *px ;
float y[2][2], *py[2], **ppy ;
float z[2][2][2], **ppz[2], *pz[2][2],
      ***pppz ;
;
// erorrs :
x=px ;    // constant lvalue
pz=x ;    // constant lvalue
ppz=ppy ; // constant lvalue
py=z ;    // constant lvalue, different types
;
// warnings : incompatible pointer types
ppy=y ;   // ** != [][]
pppz=z ;  // *** != [][][]
ppy=pz ;  // ** !=  *[][]
px=y ;    // * != [][]
px=z ;    // * != [][][]
;
**pppz=*ppy=px=x ;   // float *
*pppz=ppy=py ;       // float **
pppz=ppz ;           // float ***
;
**pppz=*ppy=*py=pz[1][1] ;  // float *
*pppz=ppy=ppz[1] ;          // float **
;
*py=x ;       // float * ; or
py[0]=x ;
*py=*ppy=px=x ;
;
**pz=x ;      // float * ; or
pz[0][0]=x ;
```

```
**pz=*ppy=px=x ;
;
*ppz=ppy ;    //  float ** ; or
ppz[0]=ppy ;
*ppz=*pppz=ppy ;
;
*px=*x ;         // float
**ppy=**y ;      // float
***pppz=***z ;   // float
```

Uwaga: Kompilator podczas tłumaczenia programu źródłowego sprawdza nie tylko zgodność typów danych, ale także zgodność typów adresowych. Niezgodność typów obiektów adresowych zwykle nie jest sygnalizowana jako błąd, lecz jako ostrzeżenie.

Poprawne przypisania pokazujemy poniżej.

```
const NR=3, NC=5 ;
int i=1, j=2, PTR, CHAR ;
// ;
char **pp, tab[NC], ttab[NR][NC] ;
// ;
char (*ptab)[NC], *tabp[NC] ;
char **tabpp[NC] ;
char (**pptab)[NC] ;
char *(*ptabp)[NC] ;
// ;
char *ttabp[NR][NC] ;
char (*pttab)[NR][NC] ;
char *(*pttabp)[NR][NC] ;
// ;
char (*tabpt[NR])[NC] ;
;
ptab=&tab ;        // char (* )[NC]
tabpp[i]=pp ;      // char **
*tabpp=pp ;        // char **
pptab=&ptab ;      // char (** )[NC]
*pptab=ptab ;      // char (* )[NC]
*pptab=&tab ;      // char (* )[NC]
ptabp=&tabp ;      // char *(* )[NC]
;
(*ptabp)[i]=tab ;  // char *
*ptabp[i]=tab ;    // char *
        // ptab[i][0] == (*(ptabp+i))[0]
;
*tabpt=ptab ;      // char (* )[NC]
;
ttabp[i][j]=tab ;  // char *
**ttabp=tabp[i] ;  // char *
pttab=&ttab ;      // char * [NR][NC]
pttabp=&ttabp ;    // char *(* )[NR][NC]
;
tabp[i]=tab ;      // char *
*tabp=tab ;        // char *
*tabpp[i]=tab ;    // char *
**tabpp=tabp[i] ;  // char *
;
PTR=sizeof(char *) ;
CHAR=sizeof(char) ;
;
i=sizeof(ptabp);    // PTR
i=sizeof(*ptabp);   // PTR*NC
i=sizeof(**ptabp);  // PTR
i=sizeof(***ptabp); // CHAR
;
i=sizeof(pptab);    // PTR
i=sizeof(*pptab);   // PTR
i=sizeof(**pptab);  // CHAR*NC
;
i=sizeof(pttab);    // PTR
i=sizeof(*pttab);   // CHAR*NR*NC
i=sizeof(*pttabp);  // PTR*NR*NC
;
i=sizeof(tab);      // CHAR*NC
i=sizeof(ttab);     // CHAR*NR*NC
i=sizeof(*ttab);    // CHAR*NC
i=sizeof(**ttab);   // CHAR
;
i=ttab[2]-(char *)ttab ;     // i=2*CHAR*NC ;
i=(char (*)[NC])ttab[2]-ttab ;  // i=2 ;
;
i=ttabp[2]-(char **)ttabp ;  // i=2*CHAR*NC ;
i=(char *(*)[NC])ttabp[2]-ttabp ;  // i=2 ;
```

Mimo że nazwa tablicy jest wskazaniem (czyli adresem) pierwszego jej elementu, to operator `sizeof` zastosowany do nazwy tablicy zwraca fizyczny rozmiar *całej* tablicy, a nie rozmiar wskaźnika. Zatem dla tablic z przykładu powyżej mamy

```
sizeof(tab)   == sizeof(char [NC])
              == NC*sizeof(char)
sizeof(ttab)  == sizeof(char [NR][NC])
              == NR*NC*sizeof(char)
sizeof(tabp)  == sizeof(char * [NC])
              == NC*sizeof(char *)
sizeof(ttabp) == sizeof(char * [NR][NC])
              == NR*NC*sizeof(char *)
```

Zadanie: Jak kompilator zareagowałby na instrukcję poniżej? Dlaczego? Jak należałoby ją zmienić?

```
float ttab[20][40] ;
i=ttab[i=2]-ttab ;
```

Tablice wielowymiarowe raz jeszcze

Rozważmy następujące tablice dwu- i trójwymiarowe:

```
const int NR=5, NC=20, NH=50 ;
int i, j ;
;
float (ftab[NR])[NC] ;
int ((itab[NR])[NC])[NH] ;
```

Jak widać wyrażenia `ftab[i]`, `i=0, ..., NR-1`, są typu `float [NC]`, a więc oznaczają jednowymiarowe tablice `NC` obiektów typu `float`. Podobnie, wyrażenia `itab[i][j]`, `i=0, ..., NR-1`, `j=0, ..., NC-1`, są typu `int [NH]`, a zatem również są jednowymiarowymi tablicami, tym razem `NH` obiektów typu `int`.

Skoro tak, to `ftab[i]` jest adresem pierwszego elementu `i`-tego wiersza, złożonego z `NC` obiektów typu `float`. Podobnie `itab[i][j]` oznacza adres początku pewnego obszaru pamięci (identyfikowanego poprzez parę indeksów (i,j)), w którym mieści się NH liczb typu int.

Skoro tak, to zarówno ftab[i], jak i itab[i][j] można podstawić do wskaźnika do obiektu odpowiedniego typu: ftab[i] do wskaźnika do obiektów typu float, a itab[i][j] – do wskaźnika do obiektów typu int.

```
const int NR=5, NC=10, NH=3 ;
;
int i=2, j=3, k=3, n=0 ;
float ftab[NR][NC], *pf ;
int itab[NR][NC][NH], *pi ;
;
n=0 ;
for(i=0 ; i<NR ; i++)
  for(j=0 ; j<NC ; j++)
    {
    ftab[i][j]=n ;
    for(k=0 ; k<NH ; k++, n++)
      itab[i][j][k]=n ;
    }
;
pf=ftab[i] ;
pi=itab[i][j] ;
;
if((float *)ftab==&ftab[0][0]) printf("ok");
if((int *)itab==&itab[0][0][0]) printf("ok");
;
for(i=0 ; i<NR-1 ; i++)
  {
  pf=ftab[i+1] ;
  printf(" %2d: %s\n", i,
    (pf-ftab[i]==NC)?"ok":"wrong");
  }
;
for(i=0 ; i<NR-1 ; i++)
  for(j=0 ; j<NC-1 ; j++)
  {
  pi=itab[i][j+1] ;
  printf(" %2d %2d: %s\n", i, j,
    (pi-itab[i][j]==NH)?"ok":"wrong");
  }
;
for(i=0 ; i<NR ; i++)
  for(j=0 ; j<NC ; j++)
    {
    pf=&**ftab+i*NC+j ;
    if(*pf!=ftab[i][j])
      printf(" %2d %2d: wrong\n", i, j);
    }
;
for(n=i=0 ; i<NR ; i++)
  for(j=0 ; j<NC ; j++, n++)
    for(k=0 ; k<NH ; k++)
      {
      // n == i*NC+j
      pi=&***itab+(i*NC+j)*NH+k ;
      if(*pi!=itab[i][j][k])
        printf(" %d %d %d: wrong\n",i,j,k);
      }
```

Kompilator języka C zawsze zamienia nazwę tablicy jednowymiarowej na adres jej pierwszego elementu, czyli na obiekt adresowy podobny do wskaźnika do danej. Podobnie nazwa tablicy dwuwymiarowej ftab jest adresem pierwszego elementu, który tym razem jest tablicą jednowymiarową. Inaczej mówiąc ftab wskazuje na ftab[0], czyli tablicę NC liczb typu float. Zatem ftab jest obiektem adresowym podobnym do wskaźnika do tablic NC liczb typu float, czyli wskaźnika typu float (*)[NC] (nawiasy () są niezbędne!). Idąc tym samym tropem, dojdziemy łatwo do wniosku, że nazwa tablicy trójwymiarowej itab jest obiektem adresowym podobnym do wskaźnika do dwuwymiarowych tablic NC na NH liczb typu int, czyli wskaźnika typu int (*)[NC][NH].

```
const int NR=5, NC=10, NH=3 ;
;
float (t)[NC], *pf ;
float (ftab[NR])[NC], (*pft)[NC], fv ;
int (itab[NR])[NC][NH], (*pit)[NC][NH], iv ;
;
pf=t ;      // (* )      []
pft=ftab ;  // (* )[]    [][]
pit=itab ;  // (* )[][]  [][][]
;
fv=*pft[2] ;           // ftab[2][0]
fv=(*pft)[2] ;         // ftab[0][2]
;
// fv=*++pft[2] ;      // error
fv=*(++pft)[2] ;       // ftab[3][0]
pft-- ;
fv=(*(++pft))[2] ;     // ftab[1][2]
;
fv=**++pft ;           // ftab[2][0]
// fv=*++*pft ;        // error
;
iv=(*pit)[2][4] ;      // itab[0][2][4]
iv=*pit[2][4] ;        // itab[2][4][0]
;
// iv=*++pit[2][4] ;   // error
iv=*(++pit)[2][4] ;    // itab[3][4][0]
pit-- ;
iv=(*(++pit))[2][4] ;  // itab[1][2][4]
;
iv=***++pit ;          // itab[2][0][0]
;
n=sizeof(pft) ;            // 4
n=sizeof(float * [NC]) ;   // 4*NC
n=sizeof(float (* )[NC]) ; // 4
n=sizeof(float [NC]) ;     // 4*NC
;
n=sizeof(pit) ;                // 4
n=sizeof(float * [NC][NH]) ;   // 4*NC*NH
n=sizeof(float (* )[NC][NH]) ; // 4
n=sizeof(float [NC][NH]) ;     // 4*NC*NH
;
n=(int)(pft=ftab) ;
pft+=2 ;
n=(int)pft-n ;
n/=sizeof(float [NC]) ;
;
n=(int)(pit=itab) ;
```

```
pit+=2 ;
n=(int)pit-n ;
n/=sizeof(float [NC][NH]) ;
;
iv=*(itab[2]+1)[3] ;
// *((itab[2]+1)[3]) == *(*(itab[2]+1+3)) ==
// *(*(itab[2]+4)) == *(itab[2][4]) ==
// itab[2][4][0]
;
iv=*(itab+1)[2][3] ;
// *((itab+1)[2])[3] == *(*(itab+1+2))[3] ==
// *(*(itab+3))[3] == *(itab[3])[3] ==
// *itab[3][3] == itab[3][3][0]
;
iv=(*itab+1)[2][3] ;
// (itab[0]+1)[2][3] ==
// ((itab[0]+1)[2])[3] ==
// (*(itab[0]+1+2))[3] ==
// (*(itab[0]+3))[3] ==
// (itab[0][3])[3] == itab[0][3][3]
```

Powyżej wyrażenia `ftab[i]` oraz `ptf[i]` mają wartość adresu do tablic `NC` liczb typu `int`, więc ich typem jest `int [NC]`. Wyrażenia `itab[i][j]` oraz `pit[i][j]` mają wartość adresu do tablic `NH` liczb typu `float`, więc ich typem jest `float [NH]`. Wyrażenia `itab[i]` oraz `pit[i]` mają wartość adresu do dwuwymiarowych tablic `NC` na `NH` elementów całkowitych. Ich typem jest zatem `int [NC][NH]`. Wszystkie wymienione w tym paragrafie wyrżenia są obiektami stałymi.

`pft` jest wskaźnikiem do tablic zawierających `NC` liczb typu `float`, a `pit` jest wskaźnikiem do dwuwymiarowych tablic zawierających `NC` na `NH` liczb całkowitych. Oba obiekty są zmiennymi, więc mogą podlegać zmianom.

Zwróćmy uwagę na różnicę między np. `pft` i `pft[i]`: `pft` wskazuje na tablicę, a `pft[i]` jest tablicą. Czyli `pft` oznacza adres do adresu tablicy, a `pft[i]` jest adresem tablicy.

Zadanie: Omówić dokładnie wszystkie przypisania w przykładzie poniżej. Wskazać wśród nich niepoprawne. Czy wszystkie poprawne przypisania są całkowicie bezpieczne? Zwrócić uwagę na sposób inicjowania zmiennej `pt`.

```
double *p, **pp, t[20], T[10], (*pt)[20]=&t ;
;
pt=(double (*)[20])&T ;
pt=(double (*)[20])&p ;
;
p=pt ;
p=*pt+2 ;
p=&*pt[2] ;
;
pp=pt ;
pp=(double **)pt ;
;
*pp=pt ;
*pp=(double *)pt ;
```

Zauważmy ponadto, że np. zwiększenie zmiennej `pit` o `i`, czyli wykonanie instrukcji `pit+=i;`, powoduje zwiększenie adresu o `i` przeskalowane rozmiarem obiektu, na który `pit` wskazuje, czyli o `i*sizeof(int [NC][NH]) == i*NC*NH*sizeof(int)`.

```
const int NR=2, NC=3, NH=4 ;
;
char *pc ;
int i, j, k, n ;
int itab[NR][NC][NH], *pi ;
;
n=0 ;
for(i=0 ; i<NR ; i++)
  for(j=0 ; j<NC ; j++)
    for(k=0 ; k<NH ; itab[i][j][k++]=n++);
;
for(i=0 ; i<NR ; i++)
  for(j=0 ; j<NC ; j++)
    for(k=0 ; k<NH ; k++)
      {
      pc=(char *)&***itab ;
      n=0 ;
      n+=i*sizeof(int [NC][NH]) ;
      n+=j*sizeof(int [NH]) ;
      n+=k*sizeof(int) ;
      pc+=n ;
      pi=(int *)pc ;
      if(*pi!=itab[i][j][k])
        printf("%d %d %d: wrong\n", i, j, k);
      }
```

Zadanie: Do czego służy w przykładzie wyżej zmienna wskaźnikowa `pc`?

Zobaczmy jeszcze, jak zasada równoważności oraz definicja operatora indeksowania `v[i] == *(v+i)` (`v` jest adresem) przepięknie działa w przypadku tablic wielowymiarowych.

`itab[i][j][k]`	
`(*(itab+i))[j][k]`	`(i+1)` blok `NC`×`NH` liczb `int`
`(*(*itab[i]+j))[k]`	`(j+1)` tablica `NH` liczb `int`
`*(itab[j][j]+k)`	`(k+1)` liczba `int`

`itab[i]` oznacza `i+1` tablicę dwuwymiarową `NC` na `NH` liczb typu `int`. `itab[i][j]` oznacza `j+1` tablicę `NH` liczb typu `int`, która jest składową `i+1` tablicy dwuwymiarowej. `itab[i][j][k]` oznacza `k+1` element typu `int` `j+1` tablicy jednowymiarowej.

Jak widać, wszystko co napisano w tym rozdziale, wynika bezpośrednio z własności najprostszego wskaźnika do danej. To niesamowite jak 'złożoność' wyrażeń w C wynika z kilku podstawowych zasad!

Typ stały, stałe symboliczne

Stałe symboliczne nie są składową języka C. Występują jedynie w wersjach rozszerzonych oraz w C++.

Stałą symboliczną tworzy się poprzez dodanie do deklaracji zmiennej specjalnego modyfikatora typu, słowa `const`. Wszelkie próby zmiany wartości takiego obiektu są zabronione. Zostaną wykryte podczas kompilacji jako błąd.

```
const int nofp=512 ;
nofp=100 ;            // error
```

Z powodu swych własności stałe symboliczne nazywa się także 'zmiennymi tylko do odczytu'.

Definiując stałą trzeba koniecznie nadać jej wartość początkową, ponieważ później będzie to niemożliwe. Niezainicjowane stałe symboliczne są błędem wykrywanym przez kompilator.

Uwaga: Kompilator* `djgpp` *nie wykrywa niezainicjowanych stałych symbolicznych. Nadaje im po prostu wartość* `0`*.

Błędem jest także przypisywanie wskaźnikowi adresu stałej symbolicznej. Zakaz ten wprowadzono po to, aby zapobiec niejawnym zmianom wartości stałej pośrednio poprzez operacje na wskaźniku.

```
const int nofp=512 ;
const float pi ;     // error, ok in djgpp
int *p ;
;
p=&nofp ;            // warning
*p+=100 ;            // nofp=612
```

Uwaga: Kompilator pakietu* `djgpp` *niestety umożliwia przypisywanie wskaźnikowi adresu stałej symbolicznej! Dokonuje tego poprzez pominięcie słowa* `const` *w określeniu stałej, a więc chwilowo traktuje jej adres jak adres zwykłej zmiennej. W wyniku tego za pomocą tak zainicjowanego wskaźnika swobodnie można zmieniać wartość stałej symbolicznej! Jak łatwo się domyślić jest to bardzo niebezpieczne.

Można utworzyć także specjalny wskaźnik, który będzie wskazywał na stałą.

```
const int *pconst ;
```

Zmienna `pconst` jest ***wskaźnikiem do stałych*** symbolicznych typu `int`. Sama jednak nie jest stałą. Oznacza to, że wartość zmiennej `pconst` można dowolnie zmieniać, przypisując jej adres jakiegoś obiektu typu `int`. Nie wolno jednak zmieniać wartości obiektu wskazywanego przez `pconst`.

```
const float pi=3.1415, *pconst ;
float x=3.14 ;
;
pconst=&pi ;
;
pconst=&x ;
x=3.0 ;
;
*pconst=3.1 ;  // error
```

Uwaga: Adres stałej symbolicznej można przypisywać jedynie wskaźnikowi do stałej. Ale dozwolone jest także przypisywanie wskaźnikowi do stałej adresu zwykłej zmiennej.

Wskaźników do obiektów stałych używa się najczęściej jako argumentów formalnych funkcji.

Istnieje także ***stały wskaźnik***, który definiujemy następująco:

```
int num, i ;
int *const pnum=&num ;
;
*pnum=0 ;
pnum=&i ;  // error
```

Zmienna `pnum` jest stałym wskaźnikiem do obiektów typu `int`. Można zatem zmieniać wartość obiektu, na który wskazuje, ale nie wolno zmieniać adresu będącego wartością wskaźnika `pnum`.

Stały wskaźnik wymaga oczywiście zainicjowania.

Stały wskaźnik do stałej tworzy się następująco:

```
const int idx=1 ;
const int *const yes=&idx ;
```

W takim przypadku nie jest dozwolona zmiana ani wartości obiektu wskazywanego przez stałą `yes`, ani wartości tej stałej.

```
int i=2, j=5 ;
const int c=i ;
const int *pc=&i ;
int *const cp=&i ;
const int *const cpc=cp ;
;
printf("%d%d%d%d%d", i, c, *pc, *cp, *cpc);
                   // 2  2   2    2    2
;
i=7 ;
printf("%d%d%d%d%d", i, c, *pc, *cp, *cpc);
                   // 7  2   7    7    7
;
pc=&j ;
*pc=j ;    // warning
printf("%d%d%d%d%d", i, c, *pc, *cp, *cpc);
;                  // 7  2   5    7    7
cp=&j ;    // warning
*cp++ ;    // warning
(*cp)++ ;
printf("%d%d%d%d%d", i, c, *pc, *cp, *cpc);
                   // 8  2   5    8    8
;
i=2 ; j=5 ;
printf("%d%d%d%d%d", i, c, *pc, *cp, *cpc);
                   // 2  2   5    2    2
;
cpc=&j ;   // warning
*cpc=j ;   // warning
```

Gdyby stała `pc` powyżej została zainicjowana następująco:

```
const int *pc=&c ;
```

to miałoby to wpływ na działanie programu (jaki?).

Zauważmy, że stałe napisowe, np. `"maybe I am in love with you."`, należą do typu `char *`, a nie `const char *`. Gdyby było inaczej, to poniższa instrukcja przypisania nie byłaby poprawna:

```
char *love="maybe I am in love with you." ;
```

Stałe a rozmiar tablicy

Stałe symboliczne, jako obiekty, których wartość może zostać obliczona przez kompilator, mogą być używane przy ustalaniu rozmiaru tablicy statycznej. Odnosi się to także do wskaźników do stałych, stałych wskaźników oraz stałych wskaźników do stałych. Przestrzegam jednak przed ich nierozważnym stosowaniem!

```
int i=2, j=5, n ;
const int c=i ;
const int *pc_c=&c, *pc_v=&i ;
int *const cp_c=&c, *const cp_v=&i ;
int *const cp_pc_c=pc_c ;
int *const cp_pc_v=pc_v ;
const int *const cpc_pc=pc_c ;
const int *const cpc_c=&c, *const cpc_v=&i ;
;
float tabc[c] ;
float tabpc_c[*pc_c] ;
float tabpc_v[*pc_v] ;
float tabcp_c[*cp_c] ;
float tabcpc_v[*cpc_v] ;
;
for(n=0 ; n<c ; n++) tabc[n]=n ;
for(n=0 ; n<*cp_c ; n++) tabcp_c[n]=n*n ;
;
printf(" %d %d %d", i, c, *cp_c);
;
pc_c=&j ;
pc_v=&j ;
*cp_pc_c=j ;   // c=j ;
*cp_pc_v=j ;   // i=j ;
*cp_c=j ;      // c=j ;
*cp_v=j ;      // i=j ;
;
printf(" %d %d %d", i, c, *cp_c);
;
// problems :
for(n=0 ; n<c ; n++) tabc[n]+=2 ;
for(n=0 ; n<*cp_c ; n++) tabcp_c[n]*=2 ;
```

Zauważmy, że deklaracje stałych `cp_c`, `cp_pc_c` oraz `cp_pc_v` wymagają pominięcia słowa `const` w wartościach początkowych, jakie są im nadawane. Kompilator `djgpp` powiadamia o tym następującym ostrzeżeniem:

```
warning: initialization discards ‘const’
from pointer target type
```

W wyniku tego staje się możliwy dostęp nawet do wartości stałej, co jest szczególnie niebezpieczne przy operowaniu tablicami statycznymi.

Uwaga: Tablice statyczne, których rozmiar określony został za pomocą stałej, mają oczywiście ustalony rozmiar (w sensie ilości przydzielonej im pamięci), jednakże wartości samych stałych mogą w ogólności ulegać zmianie podczas wykonywania programu! Jak widać powyżej nawet stała `const` ***może zmienić swą wartość!***

Typ wyliczeniowy

Typ wyliczeniowy występuje jedynie w wersjach rozszerzonych języka C/C++.

Typ wyliczeniowy stanowi zestaw stałych całkowitych (ale nie symbolicznych), zwanych ***elementami wyliczenia***.

Różnica między elementem wyliczenia a stałą symboliczną (zadeklarowaną jako `const`) polega na tym, że elementowi wyliczenia nie przydziela się adresowalnego obszaru pamięci. Wobec tego zastosowanie operatora adresu `&` do takiego elementu jest błędem.

Deklaracja wyliczenia w swej najprostszej postaci zaczyna się słowem kluczowym `enum`, po którym następuje lista elementów ujęta w nawiasy klamrowe `{}`.

```
enum { false, true } ;
```

Domyślną wartością pierwszego elementu jest `0`. Każdy następny element, jeśli nie został zainicjowany jawnie, otrzymuje wartość o jeden większą od wartości poprzednika w liście.

Każdemu elementowi można także jawnie przypisać określoną wartość całkowitą. Nie jest zabronione przypisywanie takiej samej wartości kilku elementom.

W przypadku listy zawierającej zarówno jawne jak i niejawne przypisania, te elementy, które nie zostały zainicjowane w sposób jawny, otrzymują wartości o jeden większe od wartości elementów bezpośrednio je poprzedzających.

```
enum { noa, nob=3, noc, nod, noe=2, nof } ;
```

Wyliczeniu można przydzielić nazwę, czyli ***etykietę wyliczenia***.

```
enum Logic { true=1, false=0 } idx, found ;
```

Każde nazwane wyliczenie definiuje nowy typ danych (całkowitych). Dlatego etykiet wyliczeń (poprzedzonych słowem `enum`) można używać jako specyfikatorów typu.

```
enum Logic { false, true } ;
enum TestResult
     { notdone=-1, defeat, success } ;
;
extern enum TestResult testit(int) ;
extern enum TestResult searchfor(int) ;
;
int i ;
const int noftests=100 ;
enum TestResult result[noftests] ;
enum Logic found=false ;
;
for(i=0 ; i<noftests ; i++)
  {
  result[i]=notdone ;
  if((result[i]=testit(i))!=success)
    printf(" test no %d failed.\n", i);
  }
;
for(i=0 ; i<noftests ; i++)
  {
  if(result[i]==defeat) continue ;
  if(searchfor(i)==success)
    { found=true ; break ; }
;
if(found) printf("OK.\n");
```

Zmiennym typu określonego przez wyliczenie można przypisywać tylko wartości ze zbioru elementów tego wyliczenia. Jest tak tylko w niektórych wersjach C++. W pakiecie djgpp wyliczenie traktowane jest dokładnie tak samo jak typ całkowity int.

```
enum A { aa, ab, ac } a=aa ;
enum B { ba=2, bb } b=ba ;
int i=bb ;
;
a=-1 ;  // error only in C++
a=ba ;
a=bb ;  // error only in C++
```

Uwaga: Elementy wyliczeń mogą stanowić rozmiar tablicy statycznej.

Przykład: obsługa nagłówka obrazu FITS

O nagłówku obrazu w formacie FITS mowa była już wcześniej. Składa się on z zestawu 80-bajtowych bloków, z których każdy ma następujący format:

keyword=value ,

gdzie ***keyword*** oznacza słowo przypisane pewnemu parametrowi, a ***value*** wartość tego parametru. Przez parametr rozumie się tutaj zarówno liczbę, jak i tekst.

Taki nagłówek zwykle wygląda następująco (pokazuję tylko część):

```
SIMPLE  =                    T /
BITPIX  =                  -32 /
NAXIS   =                    2 /
NAXIS1  =                  574 /
NAXIS2  =                  380 /
```

Powyżej znak / oznacza początek komentarza (do końca bloku). Znak = zajmuje stałą pozycję w 9 bajcie bloku.

Zatem czytanie nagłówka sprowadza się do przeanalizowania zgodnie ze wzorcem podanym wyżej wszystkich jego bloków i uzyskania wartości poszczególnych parametrów, takich jak np. czas naświetlania obrazu exptime, filtr filter, typ obrazka type i jego rozmiary xend i yend.

```
#define FTSK_END "END"
;
enum { FTSK_SIMPLE, FTSK_BITPIX, FTSK_NAXIS,
       FTSK_NAXIS1, FTSK_NAXIS2, FTSK_NAXIS3,
       FTSK_BSCALE, FTSK_BZERO, FTSK_FILTER,
       FTSK_EXPTIME, FTSK_DATA_TYP,
       NofFTSKeyWords } ;
enum { FTST_O, FTST_D, FTST_B, FTST_F,
       FTST_I, NofFTSFrameTypes } ;
enum { FTSF_U, FTSF_0, FTSF_MU, FTSF_MB,
       FTSF_MV, FTSF_MR, FTSF_MI, FTSF_Su,
       FTSF_Sv, FTSF_Sb, FTSF_Sy, FTSF_SHan,
       FTSF_SHaw, NofFTSFilters } ;
;
static char *FTSKeyWord[NofFTSKeyWords]=
  { "SIMPLE", "BITPIX", "NAXIS",
    "NAXIS1", "NAXIS2", "NAXIS3",
    "BSCALE", "BZERO", "FILTER",
    "EXPTIME", "DATA-TYP" } ;
static char *FTSFrameType[NofFTSFrameTypes]=
  { "OBJECT", "DARK", "BIAS", "FLAT",
    "ILLUM" } ;
static char *FTSFilter[NofFTSFilters]=
  { "?", "-", "U", "B", "V", "R", "I",
    "u", "v", "b", "y",
    "Ha narrow", "Ha wide" } ;
;
int iline, ikey, i ;
char keyword[256], value[256] ;
char bitpixidx=0, idxENDfound=0 ;
FILE *file=fopen("image.fits","rb") ;
;
int bitpix, nofaxis, nofframes, filter ;
int xorig, xend, yorig, yend, type ;
double bscale, bzero, exptime ;
;
for(iline=1 ; !feof(file) ; iline++)
  {
  if(iline>1300) break ;
  if(!freadFTShline(file,keyword,value))
    { printf(" HErr: Header reading error!");
    break ; }
  if(strcomp(keyword,FTSK_END))
    { idxENDfound=1 ; break ; }
  for(ikey=0 ; ikey<NofFTSKeyWords ; ikey++)
    if(strcomp(keyword,FTSKeyWord[ikey]))
      break ;
  if(ikey==NofFTSKeyWords) continue ;
  switch(ikey)
    {
    case FTSK_BITPIX :
      bitpix=atoi(value) ; bitpixidx=1 ;
      break ;
    case FTSK_NAXIS :
      nofaxis=atoi(value) ;
      if(nofaxis<2) nofaxis=2 ;
      break ;
    case FTSK_NAXIS1 :
      xend=atoi(value) ; xorig=1 ; break ;
    case FTSK_NAXIS2 :
      yend=atoi(value) ; yorig=1 ; break ;
    case FTSK_NAXIS3 :
      nofframes=atoi(value) ;
      if(nofframes<1) nofframes=1 ;
      break ;
    case FTSK_BSCALE :
      bscale=atof(value) ; break ;
    case FTSK_BZERO :
      bzero=atof(value) ; break ;
    case FTSK_FILTER :
      for(i=0 ; i<NofFTSFilters ; i++)
        if(strcomp(value,FTSFilter[i]))
          break ;
      if(i==NofFTSFilters)
        { filter=FTSF_U ; break ; }
      filter=i ; break ;
    case FTSK_EXPTIME :
      exptime=atof(value) ; break ;
    case FTSK_DATA_TYP :
      for(i=0 ; i<NofFTSFrameTypes ; i++)
        if(strcomp(value,FTSFrameType[i]))
          break;
      if(i==NofFTSFrameTypes)
```

```
        { type=FTST_I ; break ; }
      type=i ; break ;
    }
  }
if(!idxENDfound)
  printf(" HErr: END not found!");
if(!bitpixidx)
  printf(" HErr: No Bit/Pixel Rate!");
if(nofaxis<2)
  printf(" HErr: Invalid number of axis!");
if(!xend || !yend)
  printf(" HErr: No FrameBox!");
```

Proszę zauważyć, że elementy wyliczeń, jako obiekty stałe, zostały powyżej użyte jako rozmiary tablic statycznych.

Struktury

Struktura jest najbardziej ogólnym typem danych w języku C, typem definiowanym przez programistę. Jest to obiekt złożony z jednej lub kilku zmiennych, w ogólności różnych typów.

Struktury stosuje się zwykle do grupowania pewnego zestawu własności w jedną całość określoną nazwą struktury. Np. może to być pozycja listy płac (zawierająca imię, nazwisko, adres, wynagrodzenie, itd.), zestaw parametrów fotometrycznych gwiazdy (nazwa, jasności i wskaźniki barwy w wybranym systemie fotometrycznym), czy też data (dzień, miesiąc i rok).

Struktury znacznie ułatwiają zorganizowanie złożonych danych, ponieważ grupa związanych ze sobą zmiennych dzięki strukturze traktowana jest jak jeden obiekt, a nie zestaw oddzielnych obiektów, operowanie którymi zwykle jest bardziej niewygodne.

Deklarację ***struktury*** rozpoczyna słowo kluczowe `struct`. Po nim może wystąpić (ale nie musi) nazwa, zwana ***etykietą struktury***. Dalej następuje ujęta w nawiasy klamrowe `{}` lista deklaracji poszczególnych elementów (zmiennych) struktury, zwanych ***składowymi struktury***. No i na końcu może pojawić się lista identyfikatorów nowych zmiennych.

```
struct Date { int day, month, year ; } ;
struct Time { int hour, minute ;
              float second ; } ;
```

Deklaracja struktury, po której nie występuje lista zmiennych opisuje tylko wzorzec struktury.

Etykieta struktury identyfikuje ten rodzaj struktury traktowany jako nowy typ danych. Może zatem dalej być używana jako skrót deklaracji szczegółowej.

```
struct Date { int day, month, year ; } d ;
;
struct Date birthdate, deathdate ;
struct Date *daywhenyoufellinlove ;
```

Składowej struktury, etykiecie i normalnej (tj. nie będącej składową struktury) zmiennej można nadać tę samą nazwę. Są one zawsze odróżnialne przez kontekst, w jakim mogą się pojawić. Ponadto, składowe różnych struktur także mogą mieć taką samą nazwę.

```
struct d { int y, m, d ; } ;
struct t { int h, m ; float s ; } ;
;
int t, d ;
struct d date ;
struct t time ;
```

Struktury mogą być zagnieżdżone. Oznacza to, że składową struktury może być inna, zdefiniowana wcześniej struktura.

```
struct position { float x, y ; } ;
struct photometry { float mag, err ; } ;
;
struct starphoto { int no ;
                   struct position r ;
                   struct photometry photo ;
                   float sky ; } ;
// or
struct starphoto
       { int no ;
          struct { float x, y ; } r ;
          struct { float mag, err ; } photo ;
          float sky ; } ;
```

```
struct date { int day, month, year ; } ;
struct person { char *firstname, *lastname ;
                struct data birthdate ;
                struct date hiredate ; }
```

Struktury mogą być inicjowane podobnie jak i inne zmienne poprzez podanie listy wartości początkowych, np.

```
struct date d={ 29, 6, 1970 } ;
struct person gk={ "Grzegorz", "Kopacki",
                   29,  6, 1970,
                   15, 12, 1992 } ;
```

Proszę zwrócić uwagę na kolejność wartości początkowych. Musi ona odpowiadać kolejności deklarowania składowych struktury. Wiąże się to ściśle ze sposobem przydzielania pamięci strukturze: poszczególne składowe umieszczane są w pamięci jedna za drugą dokładnie w kolejności ich deklarowania. Cały taki obszar pamięci traktowany jest jako jeden obiekt. Można się do niego odwołać poprzez wskazanie na jego początek. O wskaźnikach do struktur powiemy więcej za chwilę.

Dostęp do składowej struktury

Dostęp do składowej struktury umożliwia operator składowej `.`, który wiąże nazwę struktury ***structure*** z nazwą jej składowej ***component***. Konstrukcja postaci

structure*.*component

zwraca *wartość* składowej struktury.

Operator struktury `.` (wraz z drugim operatorem strukturowym `->` oraz operatorami `()` i `[]`) znajduje się na szczycie hierarchii pierwszeństwa, a więc wiąże najsilniej argumenty wyrażenia.

```
struct date { int day, month, year ; } ;
struct name { char *firstname ;
              char *lastname ; } ;
struct person { struct name fullname ;
                char *address ;
```

```
                struct date birthdate ;
                struct date hiredate ; } ;
;
char *gkfn="Grzegorz", *gkln="Kopacki" ;
char *gkadr="top secret" ;
;
struct date d ;
struct person gk ;
int leap, c ;
;
d.day=26 ; d.month=11 ; d.year=2100 ;
;
d.day++ ;  // (d.day)++ ;
--d.day ;  // --(d.day) ;
;
leap=(!(d.year%4) && d.year%100) ||
      !(d.year%400) ;
if(leap) printf(" %d : leap year.", d.year);
else printf(" %d : not leap year.", d.year);
;
gk.fullname.firstname=gkfn ;
gk.fullname.lastname=gkln ;
gk.address=gkadr ;
;
printf(" %s", gk.fullname.firstname);
printf(" %s", gk.fullname.lastname);
printf(" %s", gk.address);
;
*gk.fullname.firstname='g' ;
*gk.fullname.lastname='k' ;
;
printf(" %s", gk.fullname.firstname);
printf(" %s", gk.fullname.lastname);
;
c=*++gk.address ;  // *++(gk.adress) ; 'o'
c=*gk.address-- ;  // *(gk.adress)-- ; 'o'
c=*gk.address ;    //                  't'
;
c=++*gk.address ;    // 'u'
c=(*gk.address)-- ;  // 'u'
c=*gk.address ;      // 't'
;
c=gk.address[5] ;    // 'e'
;
gk.address=(char *)calloc(30,sizeof(char)) ;
// ...
free(gk.address);
```

Uwaga: Zawsze należy pamiętać o właściwym zainicjowaniu składowych wskaźnikowych struktury!

Wskaźnik do struktury

W języku C obowiązuje kilka ograniczeń dotyczących struktur. Jedynymi dozwolonymi operacjami na strukturze są: pobranie jej adresu za pomocą operatora `&` oraz działanie na jednej z jej składowych.

Zatem w standardowych wersjach języka C struktura nie może być ani wynikiem przypisania, ani argumentem oraz wynikiem funkcji (więcej o funkcjach niebawem; dowiemy się wtedy, że nie musi to być prawdą).

Uwaga: Rozszerzenia języka C/C++ (także kompilator `djgpp`***) ograniczeń tych nie mają.***

```
struct pos { float x, y ; } r={2,3}, x ;
;
x=r ;
printf(" %.1f %.1f", x.x, x.y);
```

Ponadto, we wszystkich implementacjach języka C także wskaźnik do struktury powyższym ograniczeniom nie podlega.

Deklaracja wskaźnika do struktury jest dokładnie taka sama, jak i wskaźników do innych typów danych.

```
struct date { int day, month, year ; } ;
struct date *pd ;
```

Powyżej zmienna `pd` jest wskaźnikiem do struktury typu `date`. Do składowej np. `year` takiego obiektu można odwołać się używając znanego już nam operatora `.` następująco:

```
(*pd).year
```

(nawiasy są konieczne!). Zmienna `pd` zawiera adres pewnego kawałka pamięci, w którym znajduje się obiekt będący strukturą typu `date`. Operacja wyłuskania `*pd` zwraca nam właśnie tę strukturę. Dopiero w tym momencie możliwy jest dostęp do poszczególnych składowych struktury, np. `(*pd).day`.

Wskaźniki do struktur są tak często stosowane, że dla wygody wprowadzono jeszcze jeden operator dostępu do składowej, operator `->`. Jeśli ***structure*** jest wskaźnikiem do struktury, a ***component*** składową struktury, to wyrażenie

structure`->`***component***

zwraca *wartość* tej składowej. Dla przykładu powyżej mamy zatem

```
pd->year .
```

Zapisy z `(*).` oraz `->` są sobie całkowicie równoważne. Zauważmy także, że operator `.` wymaga aby jego lewy argument był nazwą struktury, natomiast lewym argumentem dla operatora `->` musi być wskaźnik do struktury.

Oba operatory strukturowe `.` i `->` oraz operatory nawiasowe `()` i `[]` mają najwyższe priorytety podczas wiązania elementów wyrażeń. Ponadto wszystkie są lewostronnie łączne.

```
struct date { int day, month, year ; } ;
struct name { char *firstname ;
              char *lastname ; } ;
struct person { struct name fullname ;
                char *address ;
                struct date birthdate ;
                struct date hiredate ; } ;
;
char *gkfn="Grzegorz", *gkln="Kopacki" ;
char *gkadr="top secret", *text="empty" ;
char *pc, *pct=text ;
;
struct date d={20,6,2001}, *pd ;
struct person *list ;
```

```
int c, np=5, nc=256 ;
;
pd=&d ;
c=pd->month ;
c=(*pd).month ;
;
c=++pd->month ;  // ++(pd->month)
c=(*pd).month++ ;
;
list=(struct person *)
     calloc(np,sizeof(struct person)) ;
for(c=0 ; c<np ; c++)
  {
  list->fullname.firstname=
    (char *)calloc(nc,sizeof(char)) ;
  list->fullname.lastname=
    (char *)calloc(nc,sizeof(char)) ;
  list++->address=
    (char *)calloc(nc,sizeof(char)) ;
  }
list-=np ;
;
pc=list->fullname.firstname ;
while(*list->fullname.firstname++=*gkfn++);
list->fullname.firstname=pc ;
;
pc=list->fullname.lastname ;
while(*list->fullname.lastname++=*gkln++);
list->fullname.lastname=pc ;
;
pc=list->address ;
while(*list->address++=*gkadr++);
list->address=pc ;
;
printf(" %s  ", (*list).fullname.firstname);
printf(" %s  ", (*list).fullname.lastname);
printf(" %s\n", (*list).address);
;
list++ ;
for(c=1 ; c<np ; list++, c++)
  {
  pc=list->fullname.firstname ; pct=text ;
  while(*list->fullname.firstname++=*text++);
  list->fullname.firstname=pc ; text=pct ;
  ;
  pc=list->fullname.lastname ; pct=text ;
  while(*list->fullname.lastname++=*text++);
  list->fullname.lastname=pc ; text=pct ;
  ;
  pc=list->address ; pct=text ;
  while(*list->address++=*text++);
  list->address=pc ; text=pct ;
  }
list-=np ;
;
for(list++, c=1 ; c<np ; list++, c++)
  {
  list->birthdate.day=list->birthdate.month=
    list->birthdate.year=0 ;
  list->hiredate.day=list->hiredate.month=
    list->hiredate.year=0 ;
  }
list-=np ;
;
(*list).birthdate.day=29 ;
(*list).birthdate.month=6 ;
(*list).birthdate.year=1970 ;
(*list).hiredate.day=15 ;
(*list).hiredate.month=12 ;
(*list).hiredate.year=1992 ;
;
for(c=0 ; c<np ; c++)
  {
  printf(" %d ", c);
  printf(" %s ", list[c].fullname.firstname);
  printf(" %s ", list[c].fullname.lastname);
  printf(" %s ", list[c].address);
  printf("%d:",  list[c].birthdate.day);
  printf("%d:",  list[c].birthdate.month);
  printf("%d  ", list[c].birthdate.year);
  printf("%d:",  list[c].hiredate.day);
  printf("%d:",  list[c].hiredate.month);
  printf("%d\n", list[c].hiredate.year);
  }
;
c=++*list->address ;         // or
c=++(*(list->address)) ;     // or
c=++(*((*list).address)) ;
printf(" %c %c\n", c, *list->address);
                             // 'u' 'u'
;
c=*++list->address ;         // or
c=*++(list->address) ;       // or
c=*++((*list).address) ;
printf(" %c\n", c);         // 'o'
;
c=*list->address++ ;         // or
c=*(list->address)++ ;       // or
c=*((*list).address)++ ;
printf(" %c\n", c);         // 'o'
c=*list->address ;
printf(" %c\n", c);         // 'p'
;
c=(*list->address)++ ;
printf(" %c %c\n", c, *list->address);
                             // 'p' 'q'
;
c=*(++list)->address ;       // or
c=*((*++list).address) ;
printf(" %c\n", c);         // 'e'
;
c=*list++->address ;         // or
c=*((list++)->address) ;     // or
c=*(*list++).address ;
printf(" %c\n", c);         // 'e'
;
list->address+=4 ;
printf(" %c\n", c);         // 'y'
c=++*list->address ;
printf(" %c\n", c);         // 'z'
```

```
;
list-=2 ;
;
for(c=0 ; c<np ; c++)
  printf(" %d %s\n", c, list[c].address);
;
list->address-=2 ;
list[2].address-=4 ;
;
for(c=0 ; c<np ; c++)
  printf(" %d %s\n", c, list[c].address);
```

Powyższy przykład zawiera wiele odwołań zarówno do wskaźnika do struktury (np. `list`), jak i do wskaźnika-składowej struktury (np. `list->address`). Proszę zwrócić uwagę na priorytety operatorów `*`, `[]`, `.`, `->` oraz `++`, a więc także na konieczność używania w niektórych przypadkach nawiasów. Wzmianki wymaga także fakt, że cała pamięć dla struktur przydzielana jest dynamicznie.

Przypominam także, że wszelkie operacje na wskaźniku do struktury (np. `list`) wykonuje się z uwzględnieniem rozmiaru struktury. Zatem `list++` zwiększa `list` o wiadomą wielkość tak, aby otrzymać adres następnego bloku pamięci o rozmiarze odpowiadającym rozmiarowi struktury.

Uwaga: Rozmiar obszaru pamięci zajmowanego przez daną strukturę nie zawsze jest równy sumie rozmiarów jej składowych! Ze względu na pewne wymagania stawiane przez różne obiekty na położenie w pamięci, może okazać się, że struktura ma 'dziury'. Nie ma to oczywiście wpływu na poprawność dostępu do elementu pamięci poprzez wskaźnik do struktury.

Zmienna `list` jest wskaźnikiem, więc stosuje się do niej zasada równoważności wyrażeń `list[i]` oraz `*(list+i)`. Można ich używać zamiennie. Zatem pobranie `j`-tego znaku składowej `address` `i`-tej struktury można wykonać następująco (wypisujemy wszystkie możliwości, także z użyciem operatora `->`):

```
list[i].address[j]
*((*(list+i)).address+j)
;
*(list[i].address+j)
(*(list+i)).address[j]
;
(list+i)->address[j]
*((list+i)->address+j)
```

Operator wyłuskania `*` ma niższy priorytet od operatora strukturowego `.`. Z tego powodu w wyrażeniach typu `(*list).address` nawiasy `()` są niezbędne do wymuszenia właściwej (i jedynej poprawnej) kolejności wiązania argumentów. A jak zachowałby się kompilator w przypadku braku tych nawiasów?

Niektóre z powyższych wyrażeń są trochę zbyt złożone. Rozpiszmy zatem szczegółowo kolejność wiązania argumentów dla kilku z nich:

```
list[i].address[j] :
  list                 // struct person *
  list[i]              // struct person
  list[i].address      // char *
  list[i].address[j]   // char
;
*((*(list+i)).address+j) :
  list+i                     // struct person *
  *(list+i)                  // struct person
  (*(list+i)).address        // char *
  *((*(list+i)).address+j)   // char
```

Tablice struktur poznamy za chwilę, ale już teraz podkreślam, że gdyby w powyższych przykładach zmienna `list` była tablicą struktur, to nadal stosowałyby się do niej takie same postaci wyrażeń.

Rozpatrzmy jeszcze przypadek wskaźnika do struktury, której jedna ze składowych także jest wskaźnikiem do pewnej struktury.

```
struct date { int day, month, year ; } ;
struct name { char *fname, *lname ; } ;
struct person { struct name *names ;
                struct date *dates ; } ;
;
struct person *list ;
int np=3 ;  // max no of persons
int nc=5 ;  // max no of chars
int nn=3 ;  // max no of names
int nd=3 ;  // max no of dates
int i, j, k, c ;
;
list=(struct person *)
     calloc(np,sizeof(struct person)) ;
for(i=0 ; i<np ; i++)
  {
  list->names=(struct name *)
    calloc(nn,sizeof(struct name)) ;
  for(j=0 ; j<nn ; j++)
    {
    list->names->fname=
      (char *)calloc(nc,sizeof(char)) ;
    list->names++->lname=
      (char *)calloc(nc,sizeof(char)) ;
    }
  list->names-=nn ;
  list++->dates=(struct date *)
    calloc(nd,sizeof(struct date)) ;
  }
list-=np ;
;
c='a' ;
for(i=0 ; i<np ; i++, c+=2)
  {
  for(j=0 ; j<nn ; j++, c++)
    {
    for(k=0 ; k<nc-1 ; k++)
      {
      list[i].names[j].fname[k]=c+k ;
      list[i].names[j].lname[k]=c+2+k ;
      }
    list[i].names[j].fname[k]=0 ;
    list[i].names[j].lname[k]=0 ;
```

```
    }
  for(j=0 ; j<nd ; j++)
    {
    list[i].dates[j].day=c%31 ;
    list[i].dates[j].month=c%12 ;
    list[i].dates[j].year=2000+c%5 ;
    }
  }
;
for(i=0 ; i<np ; i++)
  {
  printf(" %d\n", i);
  c=(nn>nd)?nn:nd ;
  for(j=0 ; j<c ; j++)
    {
    if(j<nn)
      printf(" %s %s ",
        list[i].names[j].fname,
        list[i].names[j].lname);
    if(j<nd)
      printf(" %02d:%02d:%d\n",
        list[i].dates[j].day,
        list[i].dates[j].month,
        list[i].dates[j].year);
    }
  }
 /*
 0
 abcd cdef  07:04:2000
 bcde defg  07:04:2000
 cdef efgh  07:04:2000
 1
 fghi hijk  12:09:2000
 ghij ijkl  12:09:2000
 hijk jklm  12:09:2000
 2
 klmn mnop  17:02:2000
 lmno nopq  17:02:2000
 mnop opqr  17:02:2000
 */
;
c=++*list->names->fname ;       // or
c=++*((list->names)->fname) ;
c=++*(*(*list).names).fname ;
printf(" %c %c\n", c, *list->names->fname);
                      // 'b', 'b'
;
c=*++list->names->fname ;       // or
c=*++(list->names->fname) ;
c=*++((*(*list).names).fname) ;
printf(" %c\n", c);  // 'b'
;
c=*list->names->fname++ ;       // or
c=*(list->names->fname)++ ;
printf(" %c\n", c);  // 'b'
c=*list->names->fname ;
printf(" %c\n", c);  // 'c'
;
c=(*list->names->fname)++ ;
printf(" %c %c\n", c, *list->names->fname);
                      // 'c', 'd'
;
c=*(++list)->names->fname ;     // or
c=*((*(*++list).names).fname) ;
printf(" %c\n", c);  // 'f'
;
c=*list++->names->fname ;       // or
c=*((list++)->names->fname) ;
printf(" %c\n", c);  // 'f'
c=*list->names->fname ;
printf(" %c\n", c);  // 'k'
;
c=*(++list->names)->fname ;     // or
c=*((*++(*list).names).fname) ;
printf(" %c\n", c);  // 'l'
;
c=*list->names++->fname ;       // or
c=*((list->names++)->fname) ;
printf(" %c\n", c);  // 'l'
c=*list->names->fname ;
printf(" %c\n", c);  // 'm'
```

Podobnie jak poprzednio rozpiszemy kilka wyrażeń uwzględniając kolejność wiązania argumentów. Wszystkie nawiasy okrągłe () poniżej są niezbędne (dlaczego?).

```
list[i].names[j].fname[k] :
  list                        struct person *
  list[i]                     struct person
  list[i].names               struct name *
  list[i].names[j]            struct name
  list[i].names[j].fanme      char *
  list[i].names[j].fname[k]   char
  ;
  list
  *(list+i)
  (*(list+i)).names
  *((*(list+i)).names+j)
  (*((*(list+i)).names+j)).fname
  *((*((*(list+i)).names+j)).fname+k)
;
*list->names->fname :
  list                  struct person *
  list->names           struct name *
  list->names->fname    char *
  *list->names->fname   char
  ;
  list
  (*list).names
  (*(*list).names).fname
  *(*(*list).names).fname
```

Poniżej rozbierzemy na szczegóły także trochę wyrażeń zawierających operator zwiększania ++. Zaznaczone zostaną typy tylko tych wskaźników, które ulegają zmianie. Niekiedy operator ++ wiąże silnej od operatora ->, pomimo że ten drugi ma większy priorytet od pierwszego. Dlaczego?

```
*list++->names->fname :
  list++                struct person *
  list++->names
  list++->names->fname
```

```
  *list++->names->fname
;
*++list->names->fname :
  list->names
  list->names-fname
  ++list->names->fname   char *
  *++list->names->fname
;
*list->names++->fname :
  list->names
  list->names++          struct name *
  list->names++->fname
  *list->names++->fname
;
*list->names->fname++ :
  list->names
  list->names->fname
  list->names->fname++   char *
  *list->names->fname++
```

Zadanie: Zapisać powyższe wyrażenia najpierw za pomocą operatora `[]`, a następnie za pomocą operatora `*`.

Zauważmy jeszcze jedną ważną rzecz. Otóż w przykładach powyższych zarówno struktura `list` jak i niektóre jej składowe (struktury oraz dane znakowe), czyli

```
list
list->names
list->names->fname
list->names->lname
list->dates
```

są wskaźnikami. Zatem wszystkie te obiekty mogą być poddawane takim samym operacjom jak dowolny wskaźnik, nawet składowe 'pośrednie', jaką w naszym przypadku jest element `names`. Gdyby składowa `names` wskaźnikiem nie była działań takich podejmować nie można byłoby. A więc np. wyrażenie `list->names++` byłoby niepoprawne, natomiast `list->names.fname++` poprawnym by było.

Zadanie: Niech `a->b.c->d->e.f` jest składową pewnej struktury i niech `f` jest wskaźnikiem. Wskazać wszystkie inne wskaźniki. Wypisać wszystkie możliwe poprawne wyrażenia zawierające operator zmniejszania `--`. Następnie przepisać te wyrażenia tak, aby nie zawierały operatora `->`. Zakładamy, że wszystkie wskaźniki wskazują sensownie.

Zadanie: Zdefiniować odpowiednie struktury z poprzedniego zadania zakładając, że składowa `f` jest wskaźnikiem znakowym. Dokonać najpierw 'statycznego', a następnie dynamicznego przydziału pamięci dla jednego obiektu.

Jeszcze słów kilka o adresach do składowych struktur.

```
struct date { int day, month, year ; } ;
struct name { char *fname, *lname ; } ;
struct person { struct name fullname ;
                struct date *birthdate ;
                float *age ; } ;
;
struct person *pp, gk ;
struct name *np ;
struct date *dp ;
float *fp ;
int *ip ;
char *cp ;
;
pp=&gk ;
np=&gk.fullname ;
dp=gk.birthdate ;
ip=&gk.birthdate->day ;
cp=gk.fullname.fname ;
fp=gk.age ;
;
np=pp->fullname ;
dp=pp->birthdate ;
ip=&pp->birthdate->day ;
cp=pp->fullname.fname ;
fp=pp->age ;
```

Jak widzimy, za pomocą operatora `&` można swobodnie pobrać adres dowolnego elementu składowego struktury. Wykorzystamy to później podczas omawiania funkcji.

Przykład: obsługa nagłówka obrazu FITS (c.d.)

Poniżej przytaczam definicję struktury `FTSHeader`, która grupuje wszystkie istotne parametry obrazu w formacie FITS.

```
struct Time { int hour, min, sec ; } ;
struct Date { int day, mon, year ; } ;
struct Box
 { int xorig, xend, yorig, yend ; } ;
;
struct FTSHeader
 {
 int type, filter, binning, bitpix,
     nofaxis, nofframes ;
 double exptime, gain, temp ;
 double bscale, bzero ;
 char *object, *observer, *origin ;
 struct Date date ;
 struct Time time ;
 struct Box box, frame, prescan ;
 } ;
```

Tablice struktur

Skoro znamy już doskonale wskaźniki do struktur, pojęcie tablic struktur to tylko (zbędny?) dodatek.

Tak jak już była mowa wcześniej etykieta struktury określa nowy typ danych. Możliwe jest zatem utworzenie zmiennej będącej tablicą struktur. Deklaracja takiego obiektu wygląda następująco:

```
const int ND=33 ;
struct date { int day, month, year ; } ;
struct date birthdate[ND] ;
```

Rozmiar tablicy można pominąć, ale wtedy tablicę taką należy zainicjować. Jak zwykle w takich przypadkach, rozmiar zostanie wyznaczony przez kompilator.

Inicjowanie tablicy struktur przeprowadza się podobnie jak i dla innych znanych już nam typów obiektów, mianowicie poprzez podanie listy wartości początkowych, ujętą w nawiasy klamrowe `{}`.

```
struct day { char *name ; int count } ;
struct day daytab[]={ "Montag",     0,
                      "Dienstag",   0,
                      "Mittwoch",   0,
                      "Donnerstag", 0,
                      "Freitag",    0,
                      "Samstag",    0,
                      "Sonntag",    0 } ;
int nd=sizeof(daytab)/sizeof(struct day) ;
```

Wartości początkowe są podawane struktura po strukturze, z których każda jest elementem tablicy. Wartości początkowe poszczególnych struktur (czyli elementów tablicy) powinno się w zasadzie także ująć w nawiasy klamrowe, ale gdy są one prostymi stałymi lub tekstami i gdy są podane wszystkie, wówczas wewnętrzne klamry można pominąć.

Oczywiście tablica może być także składową struktury.

```
const int nch=9, np=5 ;
struct name { char firstname[nch] ;
              char lastname[nch] ; } ;
struct person { struct name fullname ;
                char address[nch] ;
                float age ; } ;
;
struct name myfriends[np], myenemies[np] ;
struct person myfamily[np], *pp ;
int i, j, c='a' ;
;
pp=myfamily ;
;
for(i=0 ; i<np ; i++, pp++)
  {
  pp->age=0 ;
  for(j=0 ; j<nch-1 ; j++, c++)
    {
    pp->fullname.firstname[j]=c ;
    *(pp->fullname.lastname+j)=c+2 ;
    *(pp->address+j)=c+4 ;
    }
  pp->fullname.firstname[j]=0 ;
  *(pp->fullname.lastname+j)=0 ;
  *(pp->address+j)=0 ;
  }
pp-=np ;
;
for(i=0 ; i<np ; i++)
  printf(" %s %s %s %.1f\n",
    myfamily[i].fullname.firstname,
    myfamily[i].fullname.lastname,
    myfamily[i].address,
    myfamily[i].age);
       // abcd cdef efgh 0.0
       // efgh ghij ijkl 0.0
       // ijkl klmn mnop 0.0
       // mnop opqr qrst 0.0
       // qrst stuv uvwx 0.0
;
c=myfamily->fullname.firstname[3] ;
printf(" %c\n", c);  // 'd'
;
c=(myfamily+1)->fullname.firstname[2] ; // or
c=(*(myfamily+1)).fullname.firstname[2] ;
printf(" %c\n", c);  // 'g'
;
c=myfamily[2].fullname.firstname[3] ;   // or
c=*((*(myfamily+2)).fullname.firstname+3) ;
printf(" %c\n", c);  // 'l'
```

Obsługa tablic struktur jest prawie dokładnie taka sama jak wskaźników do struktur. Z tym wyjątkiem, że nazwa tablicy struktur jest stałą, a nie zmienną. Zatem nie można zastosować do niej np. operatora zwiększania `++`. Nadal oczywiście nazwa tablicy struktur oznacza adres jej pierwszego elementu.

Rozłóżmy na czynniki pierwsze niektóre wyrażenia z przykładu powyżej zwracając szczególną uwagę na kolejność wiązania argumentów (pierwszeństwo i łączność operatorów).

```
myfamily->fullname.firstname[3] :
  myfamily->fullname
  myfamily->fullname.firstname
  myfamily->fullname.firstname[3]
;
myfamily[2].fullname.firstname[3] :
  myfamily[2]
  myfamily[2].fullname
  myfamily[2].fullname.firstname
  myfamily[2].fullname.firstname[3]
```

Zauważmy, że wszystkie użyte powyżej operatory (`->`, `.`, `[]`) mają taki sam priorytet i tym co określa kolejność wiązania argumentów jest lewostronna łączność tych operatorów.

Struktury odwołujące się do samych siebie

Ciekawe, ciekawe, ale pomijam!

Ważne jest jedynie to, że w języku C nie dopuszcza się, aby struktura zawierała w sobie swoje własne wcielenie. Możliwe jest jednak odwołanie się do tej samej struktury poprzez zmienną wskaźnikową.

```
struct tnode { char *word ;
               int count ;
               struct tnode *leftnode ;
               struct tnode *rightnode ; } ;
```

Struktura powyższa jest przykładem tzw. ***drzewa binarnego*** i stanowi definicję węzła takiego drzewa.

Pola

Gdy trzeba oszczędzać pamięć, umieszczanie różnych obiektów w jednym słowie maszynowym (zmiennej typu (`unsigned`) `int`) może okazać się koniecznością. Jest tak zwłaszcza z jednobitowymi znacznikami np. symboli języka programowania w kompilatorze. Wymagany wtedy jest dostęp do kawałków słów.

Rozpatrzmy jako przykład fragment kodu kompilatora operujący na tablicy elementów języka. Każdemu takiemu elementowi przypisane są pewne informacje: że jest słowem kluczowym lub nie, jest operatorem lub nie, zmienną itd. Zazwyczaj definuje się wtedy zestaw 'masek' bitowych odpowiadających właściwym pozycjom bitów, np.

```
#define KEYWORD  01
#define EXTERNAL 02
#define STATIC   04
```

Obsługa takich znaczników bitowych sprowadza się wtedy do użycia odpowiednich operatorów bitowych.

```
unsigned int flags=0 ;
;
flags|=EXTERNAL|STATIC ;
flags&=~(EXTERNAL|STATIC) ;
;
if((flags&(EXTERNAL|STATIC))==0) ...
```

Powyżej instrukcja `flags|=EXTERNAL|STATIC;` ustawia (nadaje im wartość `1`) bity `EXTERNAL` i `STATIC` w zmiennej `flags`. Instrukcja następna natomiast oba te bity kasuje (nadaje im wartość `0`). Warunek w instrukcji `if` jest prawdziwy, gdy oba bity są skasowane.

Język C oferuje wygodniejszy sposób definiowania i operowania polami wewnątrz słowa bez konieczności korzystania z operatorów bitowych.

Pole stanowi zestaw przylegających do siebie zbiorów bitów znajdujących się w jednym słowie. Składnia deklaracji pól i dostępu do nich wzorowana jest na strukturach. Definicja pola wygląda następująco:

```
struct { unsigned int is_keyword:1 ;
         unsigned int is_extern:1 ;
         unsigned int is_static:1 ; } flags ;
```

Zmienna `flags` powyżej zawiera trzy jednobitowe pola. Liczba występująca po dwukropku oznacza rozmiar pola w bitach.

Odwołania do poszczególnych pól mają dokładnie taką samą postać jak odwołania do składowych struktur.

Pola traktowane są w wyrażeniach arytmetycznych jak liczby całkowite.

Poprzednie przykłady można teraz zapisać następująco:

```
flags.is_extern=flags.is_static=1 ;
flags.is_extern=flags.is_static=0 ;
if(!flags.is_extern && !flags.is_static) ...
```

Pole nie może przekroczyć granicy słowa; jeżeli rozmiar pola sprawia, że tak się dzieje, to pole zostaje przeniesione do następnego słowa. Pole nie musi mieć nazwy. Nienazwane pola (tylko dwukropek i rozmiar) są używane do zapychania dziur (niewykorzystanych bitów między polami). Stosując rozmiar `0` można także wymusić przesunięcie pola w granice następnego słowa.

Pola są liczbami całkowitymi bez znaku. Mogą zatem być umieszczane wyłącznie w słowach typu `unsigned int`. Pola nie są adresowalne; dlatego nie można stosować do nich operatora `&`.

```
struct FLAG
  { unsigned int a:1, b:3, c:5, :0, d:7 ; } ;
struct FLAG flag ;
;
flag.a=flag.b=flag.c=flag.d=1 ;
printf(" %3d %3d %3d %3d\n",
  flag.a, flag.b, flag.c, flag.d);
;
flag.a=flag.b=flag.c=flag.d=7 ;
printf(" %3d %3d %3d %3d\n",
  flag.a, flag.b, flag.c, flag.d);
;
flag.a=flag.b=flag.c=flag.d=17 ;
printf(" %3d %3d %3d %3d\n",
  flag.a, flag.b, flag.c, flag.d);
;
flag.a=flag.b=flag.c=flag.d=121 ;
printf(" %3d %3d %3d %3d\n",
  flag.a, flag.b, flag.c, flag.d);
;
flag.a=flag.b=flag.c=flag.d=333 ;
printf(" %3d %3d %3d %3d\n",
  flag.a, flag.b, flag.c, flag.d);
;
 /*
  1   1   1   1
  1   7   7   7
  1   1  17  17
  1   1  25 121
  1   5  13  77
  */
```

Uwaga: Podczas przypisywania pól 'szerszych' do pól 'węższych' (tak jak w przykładzie powyżej) następuje obcięcie wystających bitów. Ma to, jak widać, wpływ na wartość pola 'węższego'.

Unie

Unia jest zmienną, która może zawierać obiekty różnych typów i rozmiarów. Istotne jest to, że fizycznie (w pamięci) unia zawiera zawsze tylko *jeden* z tych obiektów.

Zatem unia jest zmienną mogącą przechować wartość *jednego* obiektu, ale obiekt ten może być dowolnego, określonego w deklaracji unii, typu danych.

Unię deklaruje się dokładnie tak samo jak strukturę, używa się tylko innego słowa kluczowego, słowa `union`.

```
union tag { int ival ;
            float fval ;
            char *pchar ; } ;
union tag uval ;
```

Zmienna `uval` zajmie w pamięci obszar wystarczająco duży, aby pomieścić w nim wartość największego z trzech typów danych, występujących w definicji unii `tag`.

Do zmiennej `uval` można wstawić wartość dowolnego z typów `int`, `float` i `char *`, a następnie używać jej dopóty, dopóki użycie to jest poprawne, tzn. typ pobierany jest zgodny z typem napóźniej wstawionym. Do programisty należy kontrola typu obiektu przechowywanego w unii. Jeśli np. do zmiennej `uval` wstawimy obiekt typu `int`, ale pobieżemy go traktując jako wskaźnik do zmiennej znakowej (`char *`), to oczywiście pojawią się kłopoty.

Odwołań do 'składowych' unii dokonuje się za pomocą operatorów strukturowych `.` oraz `->`.

```
struct tagtype ;
```

```
  { int is_int:1, is_float:1, is_pchar:1 ; } ;
struct tagtype utype ;
;
union tag { int ival ;
            float fval ;
            char *pchar ; } ;
union tag uval ;
;
int xint=4, i ;
float xfloat=5.5 ;
char *text=
  "What life is worth without love?" ;
;
uval.ival=xint ;
utype.is_int=1 ;
utype.is_float=utype.is_pchar=0 ;
if(utype.is_int)
  printf(" %d\n", uval.ival);
;
uval.fval=xfloat ;
utype.is_float=1 ;
utype.is_int=utype.is_pchar=0 ;
if(utype.is_float)
  printf(" %.3f\n", uval.fval);
;
uval.pchar=text ;
utype.is_pchar=1 ;
utype.is_int=utype.is_float=0 ;
if(utype.is_pchar)
  printf(" %s\n", uval.pchar);
```

Unie mogą występować w strukturach, a struktury w uniach. Notacja dla odwołania do 'składowej' unii w strukturze (i odwrotnie) jest taka sama, jak dla struktur zagnieżdżonych.

Zatem unia jest w pewnym sensie strukturą, w której wszystkie składowe są umieszczane bez przesunięcia względem jej początku, i która jest dostatecznie duża, aby pomieścić najobszerniejszą za składowych.

Do unii stosują się te same ograniczenia, co i do struktur. Ponadto, wskaźników do unii używa się w taki sam sposób, jak wskaźników do struktur.

Deklaracja `typedef`

Słowo kluczowe `typedef` służy do tworzenia nowych nazw typów danych. Składnia deklaracji `typedef` jest następująca:

typedef ***type-specifier name*** ;

type-specifier jest nazwą dowolnego typu danych, zarówno predefiniowanego (np. `int`) jak i nowo-utworzonego (struktury, pola, unii itp.), a ***name*** nową nazwą (synonimem) dla typu ***type-specifier***.

```
typedef int Length ;
typedef struct { int y, m, d ; } Date ;
typedef char *String ;
typedef long int *LIPTable[20] ;
;
Length i, *lengths[20] ;
Date d ;
String text="I travel without any reason." ;
LIPTable liptab ;
```

Warto zwrócić uwagę na to, że nowe nazwy typów w deklaracajach `typedef` pojawiają się dokładnie w miejscu nazwy zmiennej.

Deklaracja `typedef` w żadnym razie nie tworzy nowego typu; tworzy tylko dodatkową nazwę istniejącego już typu danych.

Funkcje

Funkcje pomagają podzielić duży problem obliczeniowy na mniejsze, samowystarczalne jednostki. Dzięki nim można korzystać z tego, co już sami zrobiliśmy wcześniej, a także z pracy innych, bez konieczności ponownego wnikania w definicje funkcji. Istnieje także całe mnóstwo zestawów 'bibliotecznych' funkcji dostarczanych razem z kompilatorami języka C, np. funkcje wejścia-wyjścia (`fopen` itd.) lub funkcje matematyczne (`sin`, `exp` itd.). Teraz skupimy się na tworzeniu nowych funkcji.

Można powiedzieć, że funkcja jest operacją zdefiniowaną przez programistę. W programie reprezentuje ją jednak nie operator, lecz nazwa.

Funkcję definiuje się następująco:

type-specifier name(***arguments***)
declarations
{
statements
}

Najpierw podaje się nazwę typu wartości przekazywanej przez funkcję, a więc część ***type-specifier***. Następnie określa się nazwę własną ***name*** funkcji. Za nią umieszcza się ujętą w nawiasy okrągłe `()` listę argumentów funkcji ***arguments***. Lista ta składa się z nazw poszczególnych argumentów oddzielonych przecinkami. Część następna, ***declarations***, zawiera deklaracje typów danych wszystkich argumentów funkcji. Główna część definicji funkcji to jej treść, a wiec blok instrukcji ***instructions***, określający czynności wykonywane przez funkcję. Treść funkcji ujmuje się w parę nawiasów klamrowych `{}`.

```
int ipower(x, n)
 int x, n ;
 {
 int power=1 ;
 while(n) { power*=x ; n-- ; }
 return(power);
 }
```

Podany wyżej sposób definiowania funkcji jest standardem w pierwotnej wersji języka C i tylko tam jest sztywno przestrzegany. Wersje rozszerzone języka C pozwalają dodatkowo na umieszczenie deklaracji agrumentów ***declarations*** bezpośrdnio po nazwie funkcji ***name***. Jednak w takim przypadku każdemu argumentowi należy przypisać typ danych, nawet jeśli niektóre z nich są takiego samego typu. Składnia takiej definicji wygląda zatem tak:

type-specifier name(***declarations***)
{
statements

```
}
```

Definicję funkcji ipower można zatem przepisać następująco:

```
int ipower(int x, int n)
 {
 int power=1 ;
 while(n) { power*=x ; n-- ; }
 return(power);
 }
```

Różne części definicji funkcji można pominąć. Najprostszą definicją funkcji jest definicja funkcji 'pustej' (atrapy):

```
dummy() {}
```

Funkcja dummy nie wykonuje czegokolwiek. Ale może być przydatna do zarezerwowania pewnej nazwy dla funkcji, którą mamy zamiar zdefiniować później.

Jeśli nazwa funkcji nie jest poprzedzona typem danych, to domyślnym typem zwracanej przez nią wartości będzie typ całkowity int.

Funkcja nie musi także posiadać listy argumentów (ale para nawiasów () jest konieczna). W wersjach rozszerzonych języka C taką 'pustą' listę argumentów oznacza się zwykle słowem void.

```
dummy(void) {}
```

W języku C nie dopuszcza się definiowania funkcji wewnątrz innych funkcji.

Instrukcja return

Narzędziem, dzięki któremu funkcja może przekazywać (poprzez nazwę) do miejsca swego wywołania określoną wartość jest instrukcja return. Składnia wywołania tej instrukcji jest następująca:

return(***expression***);

lub

return ***expression*** ;

gdzie ***expression*** jest dowolnym wyrażeniem, którego wartość zgodna jest (uwzględniając niejawne rzutowanie) z typem danej zwracanej przez funkcję.

Wartość wyrażenia ***expression*** jest zawsze przed wykonaniem instrukcji return przekształcana do typu zwracanego przez funkcję, o ile jest to możliwe.

W wywołaniu instrukcji return można pominąć wyrażenie expression (a także nawiasy okrągłe). Wówczas nie dochodzi do przekazania jakiejkolwiek wartości poprzez nazwę funkcji. Instrukcję return można także w ogóle pominąć. Gdy jej brak, sterowanie przekazywane jest również bez wartości, a wykonanie funkcji kończy się po osiągnięciu zamykającego nawiasu klamrowego. Nie jest to błędem, jednakże zwykle wtedy kompilator zgłasza pewne ostrzeżenia w związku z ewentualną niezgodnością typów: zadeklarowanego oraz zwracanego.

```
void dummy(void) {}
```

Funkcja nie musi zwracać poprzez swoją nazwę jakiejkolwiek wartości. Jest tak wtedy, gdy typ zwracanej przez funkcję 'wartości' określimy jako void. I choć tak naprawdę w tym przypadku funkcja nie zwraca czegokolwiek poprzez swą nazwę, zwykle mówi się, że funkcja przekazuje wartość typu void.

Wywołanie funkcji i przekazywanie argumentów

Program źródłowy jest właściwie zbiorem odrębnych funkcji (oraz innych obiektów zewnętrznych, o których niebawem). Jak pamiętamy wykonanie programu zaczyna się od funkcji o zarezerwowanej nazwie main. Funkcja ta musi gdzieś w tekście programu się pojawić.

Komunikacja między funkcjami odbywa się za pośrednictwem argumentów i wartości zwracanej przez funkcję (oraz poprzez zmienne zewnętrzne, o których będzie mowa wkrótce). W pliku źródłowym funkcje mogą występować w dowolnej kolejności (obiekty wykorzystywane aktualnie ale definiowane później należy oczywiście zadeklarować wraz ze słowem extern). Ponadto, program źródłowy można rozdzielić na kilka plików pod warunkiem, że żadna funkcja nie zostanie podzielona.

Definicja funkcji określa sposób jej użycia dalej w programie. Dowolne wywołanie funkcji musi być zgodne z jej definicją (pomijam możliwość przeciążania nazwy funkcji). Oznacza to taką samą ilość argumentów oraz zgodność typów wszystkich argumentów wywołania z typami odpowiednich agrumentów formalnych w definicji funkcji (ale uwaga na niejawne rzutowanie typów).

Określona funkcja bedzie wykonywana za każdym razem, gdy po jej nazwie umieści się ***operator wywołania funkcji***, czyli parę nawiasów okrągłych (), z ewentualną listą argumentów wewnątrz. Wywołanie funkcji powoduje przekazanie do niej sterowania, a wykonywanie bieżąco aktywnej funkcji zostanie zatrzymane. Po zakończeniu wykonywania wywołanej funkcji, wznawia się wykonywanie funkcji macierzystej od instrukcji następującej bezpośrednio po wywołaniu.

Jeżeli funkcja wymaga argumentów, to argumenty wywołania, czyli ***argumenty aktualne***, umieszcza się w parę nawiasów okrągłych stanowiących operator wywołania funkcji. Podobnie jak w definicji funkcji, argumenty oddziela się przecinkami. Cały ten proces nazywa się ***przekazywaniem argumentów*** do funkcji.

```
int ipower(int x, int n)
 {
 int power=1 ;
 while(n) { power*=x ; n-- ; }
 return(power);
 }
;
void main(void)
 {
 int x, y, n=3 ;
 x=ipower(2,3) ;
 x=ipower(2,n) ;
 x=2 ;
 x=ipower(x,n) ;
 printf(" %d ", ipower(x,n));
 x=ipower(x+y,2*n);
 }
```

Argumenty aktualne mogą być wyrażeniami arytmetycznymi. W takich przypadkach najpierw obliczna jest wartość wyrażenia i dopiero ona przekazywana jest funkcji jako wartość argumentu.

Uwaga: Argumenty przekazywane są wyłącznie przez wartość, nigdy przez adres.

Argumenty funkcji przekazywane są przez wartość. Oznacza to, że wywoływana funkcja otrzymuje 'prywatne', tymczasowe kopie każdego argumentu. Co za tym idzie funkcja nie ma dostępu do rzeczywistego obiektu (zmiennej), przekazywanego jej jako argument. Wewnątrz funkcji każdy argument jest zatem zmienną lokalną o wartości początkowej równej wartości, z którą funkcja została wywołana.

Wynika z tego jeden ważny wniosek. Wszelkie operacje wewnątrz funkcji (pomijając zmienne zewnętrzne) odbywają się na zmiennych lokalnych, tworzonych wraz z wywołaniem funkcji i istniejących fizycznie tylko w czasie jej wykonywania. Przed przekazaniem sterowania do funkcji macierzystej wszystkie zmienne lokalne są z pamięci kasowane.

Zmienne lokalne deklarowane wewnątrz funkcji zwykle zwie się zmiennymi ***automatycznymi***. Zaczynają istnieć w chwili wywołania funkcji i nikną po jej zakończeniu.

```
char func(char arg) { return(arg+=2); }
;
void main(void)
 {
 char x='c', y, func() ;
 ;
 y=func(x) ;
 printf(" %c %c\n", x, y);
 ;
 x=func(x) ;  // x=x+2 ;
 printf(" %c\n", x);
 ;
 return ;
 }
```

W przykładzie powyżej w definicji funkcji `func` nie pojawia się ani jedna zmienna automatyczna: wszelkie operacje wykonywane są jedynie na argumencie formalnym `arg`. Rozpatrzmy wyrażenie `y=func(x)`. W instrukcji przypisania pojawia się po prawej stronie wywołanie funkcji `func` z argumentem aktualnym (wywołania) `x`. Zatem tylko na potrzeby funkcji `func` zostanie utworzona w pamięci kopia tego argumentu, i to ona będzie przetwarzana wewnątrz funkcji jako argument formalny `arg`. Zmiennej `arg` nadawana zostaje więc wartość zmiennej `x`, a następnie `arg` jest zwiększany o 2. Zauważmy, że wartość `x` nie ulega zmianie. Wartość `arg+2` jest przekazywana poprzez nazwę funkcji `func` do funkcji głównej `main`, i to ona zostanie podstawiona pod wartość zmiennej `y`. Tak więc wartość zmiennej `x` pozostaje taka sama i wynosi `'c'`, natomiast zmiennej `y` przypisana zostaje wartość `'e'`.

Wyrażenie `x=func(x)` zawiera takie samo wywołanie funkcji `func`. Jednakże po wykonaniu tej instrukcji zmienna `x` ulegnie zmianie i przyjmie wartość `'e'`. Dlaczego?

Definicja i deklaracja funkcji

Jak pamiętamy, definicja zmiennej powoduje jej fizyczne utworzenie, a więc wydzielenie dla niej pamięci. Deklaracja natomiast sygnalizuje jedynie o istnieniu zmiennej. W deklaracji takiej używa się słowa `extern`.

Podobnie jest z funkcjami. Definicja funkcji polega na określeniu działania funkcji, a więc zawiera ***treść*** funkcji. Natomiast deklaracja funkcji zapowiada jedynie o jej istnieniu. Składa się ona w języku C z typu wyniku funkcji, jej nazwy oraz operatora wywołania funkcji, czyli pary nawiasów `()`. W wersjach rozszerzonych języka można ją także uzupełnić listą typów argumentów. W języku C++ te trzy elementy nazywa się ***prototypem*** funkcji. Jak widzimy są one wystarczające do określenia sposobu używania funkcji.

```
int ipower();          // C
int ipower(int, int);  // C++
```

Zatem: Definicja określa działanie funkcji. Natomiast deklaracja potwierdza jedynie istnienie funkcji oraz przypomina jaki jest typ wartości zwracanej przez tę funkcję. Ponadto, ale tylko w C++, deklaracja funkcji przypomina o sposobie wywołania funkcji.

Przed użyciem funkcji należy ją przynajmniej zadeklarować. Jeśli się tego nie zrobi zostanie zgłoszony błąd podczas kompilacji.

Oczywiście definicja funkcji jest jej deklaracją. Jednakże daną funkcję można w programie zdefiniować tylko raz. Deklaracja (prototyp) funkcji natomiast może pojawiać się wielokrotnie, zarówno w pilku z definicją funkcji, jak i w innych plikach, gdzie funkcja jest jedynie wywoływana.

W języku C pierwsze wystąpienie identyfikatora, po którym bezpośrednio następuje półokrągły nawias otwierajćy ((), jest traktowane jako kontekstowa deklaracja funkcji zwracającej wartość całkowitą typu `int`. Zatem w większości przypadków funkcje całkowite nie muszą być deklarowane.

Sprawdzanie zgodności typów

W wersjach rozszerzonych języka C/C++ rygorystycznie przestrzega się zgodności typów, także w przypadku funkcji.

Podczas kompilacji programu sprawdzane są zarówno typy w liście argumentów, jak i typy przekazywanych wartości. W przypadku niezgodności między typem argumentu aktualnego a typem argumentu formalnego (lub deklarowanego w prototypie funkcji), dokonuje się niejawnego przekształcenia typu, o ile jest to możliwe. Jeśli nie można wykonać przekształcenia niejawnego lub nie zgadza się ilość argumentów, to kompilator zgłasza błąd.

Prototyp funkcji dostarcza kompilatorowi informacji o typach danych, aby możliwe było sprawdzenie poprawności wywołań funkcji. Z tego też powodu nie można wywoływać funkcji bez uprzedniego jej zadeklarowania. (Zagadnienie to okazało się tak ważne, że komitet ANSI do spraw standardu języka C przyjął opisane wyżej zasady i sposób określania prototypu funkcji takie, jak w C++.)

Rozpatrzmy to zagadnienie na konkretnych przykładach. Zdefinujmy funkcję `sum` następująco:

```
int sum(int x, int y) { return(x+y); }
```

i umieśćmy ją w jakimś pliku.

Spróbujmy teraz przetłumaczyć program podany niżej. Reakcja kompilatora zależy od sposobu deklarowania funkcji.

```
void main(void)
 {
 int s, sum() ;  // C
 s=sum(s) ;
 }
```

Definicja funkcji `sum` ustala, że potrzebuje ona dwóch argumentów. W przykładzie wyżej, deklaracja tej funkcji nie zawiera listy typów argumentów. Zatem kompilator powiadamiany jest jedynie o tym, że funkcja o nazwie `sum` istnieje, i że zwraca wartość całkowitą `int`. Wywołanie `s=sum(s)` zgodne jest z tymi informacjami, więc podczas kompilacji nie pojawi się żaden błąd. Zaprotestuje dopiero linker.

Jeśli podamy jakiś prototyp funkcji `sum` (niekoniecznie zgodny z definicją funkcji) a resztę zostawimy nie zmienioną, tak jak poniżej:

```
void main(void)
 {
 int s, sum(int, int, char) ;  // C++
 s=sum(s) ;
 }
```

to kompilator zgłosi błąd o niewłaściwej ilości argumentów w wywołaniu funkcji.

Jeśli podamy właściwy prototyp funkcji `sum`, to cały przykład, aby nie zawierał błędów, należy zapisać następująco:

```
void main(void)
 {
 int s, sum(int, int) ;  // C++
 float f ;
 s=sum(s,s) ;
 f=sum(f,f) ;  // casts
 }
```

Powyżej pojawia się wywołanie funkcji `sum`, które wymaga niejawnej konwersji zarówno argumentów (z typu `float` do `int`), jak i wartości zwracanej przez funkcję (z typu `int` do `float`).

Jeszcze słów kilka o domyślnym typie wartości zwracanej przez funkcję. Jak pamiętamy jest nim typ `int`. Rozpatrzmy program, w którym pojawia się wywołanie funkcji `func`, ale nie ma podanego jej prototypu, wyglądającego tak: `double func(void)`.

```
void main(void) { double x=func() ; }
```

Program nie zawiera jawnej deklaracji funkcji `func`, zatem kompilator zakłada, że zwraca ona domyślną wartość całkowitą. Prowadzi to oczywiście do niepoprawnych wyników, ponieważ wartość zwracana przez funkcję `func` (typu `double`) traktowana jest jak wartość typu `int`.

Gdyby typy funkcji `func` i jej wywołania w `main` były sprzeczne, a oba wystąpienia `func` (prototyp i wywołanie) znajdowały się w tym samym pliku źródłowym, to kompilator zgłosiłby błąd. Natomiast gdyby funkcja `func` tłumaczona była oddzielnie (co jest bardziej prawdopodobne), to niezgodność ta nie zostałaby wykryta: funkcja przekazywałaby wartości typu `double`, które funkcja `main` traktowałaby jako typu `int`; otrzymywane wyniki nie miałyby sensu.

Zatem: Wskazane jest podawanie prototypu funkcji przed jej użyciem.

Prototypy funkcji umieszcza się często w tzw. zbiorach nagłówkowych (ich nazwy kończą się zwykle na `.h`, np. `stdio.h`.). Prototypy funkcji umieszczone w plikach nagłówkowych dołącza się do programu za pomocą dyrektywy `include` preprocesora języka C (więcej o tym dalej).

Argumenty i wartość zwracana przez funkcję

W wersjach rozszerzonych języka C typ wartości przekazywanej przez funkcję może być: zdefiniowany pierwotnie (np. `int`, `double`), zdefiniowany przez programistę (struktury, unie, pola, typ wyliczeniowy), wskaźnikowy (nie ma tutaj jakichkolwiek ograniczeń; może to być wskaźnik do czegokolwiek), albo też określony jako typ `void` (funkcja nie przekazuje wartości).

Nie można specyfikować tablicy ani funkcji jako wartości zwracanej przez funkcję. Może nią jednak być wskaźnik do tablicy albo do funkcji.

Funkcja może zwracać tylko jedną wartość.

W przypadku dostępnych typów argumentów funkcji jest podobnie jak dla wartości zwracanej przez funkcję.

Wprawdzie w języku C nie ma wyraźnych ograniczeń na ilość argumentów funkcji, należy jednak starać się aby nie była ona zbyt duża (powiedzmy najwyżej 8). Jeśli jednak chcemy przekazać funkcji wiele parametrów, to zaleca się przekazywanie ich w postaci struktur.

W języku C++ listę (typów) argumentów funkcji nazywa się ***sygnaturą*** funkcji, ponieważ często służy do odróżniania poszczególnych wersji funkcji.

Jak pamiętamy, argumenty w sygnaturze funkcji oddziela się przecinkami. Ten sam symbol może oznaczać także operator przecinkowy. Domyślnie, przecinek w liście argumentów wywołania funkcji oznacza separator argumentów. Dowolne wyrażenie przecinkowe w sygnaturze funkcji musi być zatem ujęte w nawiasy okrągłe `()`.

```
void func(int, int, int) ;
int a=0, b, c=0 ;
;
func(a,(b=3,b+2),c);
;
func(a,b=3,b+2,c);  // error
```

Powyżej drugi argument pierwszego wywołania funkcji `func` ma wartość 5.

Wskaźnikowe argumenty funkcji

W języku C argumenty są przekazywane zawsze przez wartość. Nie ma zatem bezpośredniego sposobu na to, aby wywoływana funkcja mogła zmienić wartości zmiennych przekazywanych jej jako argumenty aktualne (jak pamiętamy funkcja operuje na kopiach tych zmiennych).

Aby jednak zmiana wartości zmiennych przekazywanych do funkcji była możliwa, należy użyć wskaźników.

```
char func(char *arg) { return(*arg+=2); }
;
void main(void)
 {
 char x='c', y='h', *p=&y, func(char *) ;
 ;
 printf(" %c %c %c\n", x, y, *p);
 ;
 y=func(&x) ;
 printf(" %c %c %c\n", x, y, *p); // e
 ;
 x=func(p) ;
 printf(" %c %c %c\n", x, y, *p); // g
 }
```

Powyżej w definicji funkcji `func` występuje tylko jeden argument, który zadeklarowany został jako wskaźnik do znaku (`char *`). Wywołanie tej funkcji wymaga podania jako argumentu aktualnego adresu do zmiennej znakowej, tak jak to ma miejsce w wyrażeniach `y=func(&x)` oraz `x=func(p)`.

Proszę zwrócić uwagę na to, że zasada przekazywania argumentu do funkcji przez wartość jest w tym przypadku także spełniona. Funkcja `func` otrzymuje kopię wartości argumentu, który jednakże jest wskaźnikiem, a więc otrzymuje wartość, która jest adresem zmiennej. Zatem pośrednio poprzez działania na kopi zmiennej wskaźnikowej (adresie) możliwe jest zmienianie wartości obiektu, na który ta kopia-adres wskazuje.

Rozpatrzmy dokładnie wyrażenie `y=func(&x)`. W instrukcji przypisania pojawia się po prawej stronie wywołanie funkcji `func` z argumentem aktualnym (wywołania) `&x`. Tylko na potrzeby funkcji `func` zostanie utworzona w pamięci zmienna automatyczna `arg`, która zgodnie z definicją funkcji będzie wskaźnikiem znakowym, a więc obiektem typu `char *`. Zmienna `arg` zostanie zainicjowana wartością argumentu aktualnego, a więc adresem zmiennej `x`, czyli `&x`. Działanie funkcji `func` polega na tym, że zwiększa wartość na jaką wskazuje `arg` o 2. Ta sama (zwiększona) wartość jest także zwracana przez nazwę funkcji. Wartość `*arg+2` jest przekazywana poprzez nazwę funkcji `func` do funkcji głównej `main`, i to ona zostanie podstawiona pod wartość zmiennej `y`. Zatem ostatecznie zmianie ulegnie zarówno wartość zmiennej `x` (pośrednio, poprzez jej adres), jak i zmiennej `y` (jawnie, poprzez podstawienie). Oczywiście tak jak zwykle argument wywołania (`&x`) nie ulegnie jakiejkolwiek zmianie. Wykorzystuje się to głównie przy operacjach na napisach i ogólnie tablicach.

```
void strcopy(char *str, char *copy)
 { while(*str++=*copy++); }
;
void stradd(char *str, char *add)
 {
 while(*str) str++ ;
 while(*str++=*add++);
 }
;
void main(void)
 {
 char *you="you", love[]="maybe I love " ;
 char text[256] ;
 ;
 strcopy(text,love); stradd(text,you);
 printf(" %s\n", text);
 }
```

Przykład powyższy zawiera definicje dwóch funkcji: `strcopy` przenosi jeden napis do drugiego, a `stradd` doczepia napis do innego napisu. W funkcji głównej `main` pojawiają się wywołania obu tych funkcji. Przekazywane im są (poprzez wartość) adresy do początków napisów. Wszelkie operacje wewnątrz funkcji odbywają się jednak na kopiach tych adresów. Zatem wywołania funkcji w żaden sposób nie zmieniają wartości przekazywanych im zmiennych (w zasadzie tylko jednej, zmiennej wskaźnikowej `you`, bo pozostałe dwie są nazwami tablic, a więc stałymi). Możemy więc mieć całkowitą pewność, że po wywołaniu `strcopy(text,love)` wskaźnik `love` będzie wskazywał na ten sam znak.

Zadanie: Nazwa tablicy jest jak wiemy stałą. W wyrażeniu `stradd(text,you)` oba argumenty są takimi właśnie stałymi obiektami. Z drugiej strony funkcja `stradd` zmienia wartości agrumentów-wskaźników za pomocą operatora zwiększania `++`. Jak pamiętamy operacje takie są niedozwolone na nazwach tablic. Dlaczego zatem kompilator nie zgłasza błędu?

Funkcja może także zwracać wskaźnik. Przykład takiej funkcji pokazany jest poniżej.

```
char *fun(char *str, char c)
 {
 char *p=str ;
 while(*str)
   if(*str++==c) break ;
 if(*str) return(str-p?--str:str);
 return(p);
 }
;
void main(void)
 {
 char *name="Grzegorz", *pc ;
 ;
 pc=fun(name,'K') ;          // Grzegorz
 ;
 pc=fun(name,'e') ;          // egorz
 ;
 *(pc=fun(name,'g'))='G' ;  // Gorz
 ;
 pc=fun(name,'g') ;          // GrzeGorz
 ++*fun(name,'g') ;          // HrzeGorz
 ;
 // *fun(name,'g')++ ;       // error
 }
```

Tablicowe argumenty funkcji

Jak pamiętamy, tablica oznacza zawsze (pomijając przypadek, gdy jest ona argumentem operatora `sizeof`) wskazanie na jej pierwszy element. Zasada ta jest prawdziwa również dla argumentów funkcji.

Każdy argument formalny funkcji zadeklarowany jako typu tablicowego jest zawsze niejawnie zamieniany na typ wskaźnikowy (do elementu tablicy). Rozmiar tablicy nie ma tu żadnego znaczenia i jest pomijany. Zatem poniższe trzy deklaracje funkcji `funt` są sobie równoważne, przy czym dwie pierwsze rozumiane są tak jak ostatnia.

```
void funt(float arg[55]) ;
void funt(float arg[]) ;
void funt(float *arg) ;
;
float f, *pf, tab[33] ;
;
funt(&f) ;
funt(pf) ;
funt(tab) ;
```

Parametr `arg` został określony jako wskaźnik do obiektu typu `float`, a więc funkcji `funt` można przekazać albo tablicę obiektów typu `float`, albo wskaźnik do obiektów typu `float`.

Zadanie: Powyżej wywołanie `funt(&f);` jest jak najbardziej poprawne pod względem zgodności typów. Czy byłoby jednak całkowicie bezpieczne, gdyby funkcja `funt` zdefiniowna była następująco?

```
void funt(float *arg) { arg+=3 ; *arg+=3 ; }
```

Zadanie: Jeśli funkcja `funt` ma poprawnie obsługiwać tablice elementów typu `float`, to trzeba jej przekazać jeszcze jeden argument. Jaki? Proszę poprawić definicję funkcji `funt` z poprzedniego zadania.

Podobnie jest w przypadku tablic dwuwymiarowych. Nazwa takiej tablicy jest wskazaniem na pierwszy element, czyli na tablicę jednowymiarową. Proszę zauważyć, że rozmiar tej tablicy jednowymiarowej jest istotny i nie można go pominąć. Zatem trzy równoważne deklaracje funkcji z parametrem będącym tablicą dwuwymiarową wyglądają następująco:

```
const int NT=30 ;
;
void funtt(float arg[44][NT]) ;
void funtt(float arg[][NT]) ;
void funtt(float (*arg)[NT]) ;
;
void funtt(float arg[][]) ;  // error
;
float **ppf, tab[33][NT], tabe[2][4] ;
float (*pt)[NT], (*pte)[5] ;
float t[NT], te[5] ;
;
funtt(tab) ;
funtt(pt) ;
funtt(&t) ;
;
// warning: incompatible pointer type:
funtt(ppf) ;
funtt(tabe) ;
funtt(pte) ;
funtt(&te) ;
```

Parametr `arg` funkcji `funtt` został zadeklarowany jako wskaźnik do tablicy `NT` obiektów typu `float`, a więc funkcji `funtt` można przekazać albo tablicę tablic `NT` obiektów typu `float`, albo wskaźnik do tablicy `NT` obiektów typu `float`, albo adres tablicy `NT` obiektów typu `float`. Wszystkie inne podstawienia wymagają jawnego przekształcenia typu wskaźnika i zwykle prowadzą do nieoczekiwanego działania programu.

Zadanie: Dlaczego deklaracje

```
void funtt(float *arg[NT]) ;
void funtt(float **arg) ;
```

nie są równoważne deklaracji

```
void funtt(float (*arg)[NT]) ;
```

Dla tablic o większej ilości wymiarów wygląda to bardzo podobnie. Dla tablic trójwymiarowych należałoby napisać:

```
const int NR=30, NC=10 ;
;
void funttt(float arg[22][NR][NC]) ;
void funttt(float arg[][NR][NC]) ;
void funttt(float (*arg)[NR][NC]) ;
```

Proszę znowu zwrócić uwagę na to, że można pominąć jedynie rozmiar najbardziej zewnętrznej tablicy (jednowymiarowej). W tym wypadku `arg` jest wskaźnikiem do tablic dwuwymiarowych `NR` na `NC` elementów typu `float`.

Zadanie: Jakich wyrażeń można użyć jako argumentów wywołania funkcji `funttt`?

```
#define NR 4
#define NC 5
;
void init2tab(tab, nx, ny)
 float tab[][NC] ; int nx, ny ;
 {
 int i, j, n=1 ;
 for(i=0 ; i<nx ; i++)
   for(j=0 ; j<ny ; tab[i][j++]=n++) ;
 }
;
void show2tab(tab, nx, ny)
 float tab[][NC] ; int nx, ny ;
 {
 int i, j ;
 for(i=0 ; i<nx ; i++, printf("\n"))
   for(j=0 ; j<ny ; j++)
     printf("    printf("\n");
 }
;
void main()
 {
 float t[NC], t2[NC+5], tab[NR][NC] ;
 float tab2[NR][NC+2], tab3[NR-1][NC-1] ;
 float (*pt)[NC] ;
 int n=4 ;
 ;
```

```
init2tab(tab,NR,NC); // ok
show2tab(tab,NR,NC);
;
pt=(float (* )[NC])
   calloc(n,sizeof(float [NC])) ;
init2tab(tab,n,NC);  // ok
show2tab(tab,n,NC);
;
init2tab(&t,1,NC);   // ok
show2tab(&t,1,NC);
;
init2tab((float (* )[NC])&t2,1,NC+5);
show2tab((float (* )[NC])&t2,1,NC+5);
;
init2tab((float (* )[NC])tab2,NR,NC+2);
show2tab((float (* )[NC])tab2,NR,NC+2);
;
init2tab((float (* )[NC])tab3,NR-1,NC-1);
show2tab((float (* )[NC])tab3,NR-1,NC-1);
}
```

Program powyżej zawiera przykłady definicji funkcji, których jednym z argumentów jest wskaźnik do tablicy.

Zadanie: Wyjaśnić dokładnie jak działają funkcje `init2tab` i `show2tab`, gdy przekazuje się im wskaźnik do tablicy o rozmiarze innym niż zadeklarowany w definicji funkcji. Czy może to prowadzić do kłopotów podczas uruchamiania programu?

```
const NR=4, NC=5 ;
// ;
float tab[NR][NC] ;
float (*pts)[NC-2], (*ptl)[NC+2] ;
;
pts=(float (* )[NC-2])tab ;
;
pts[0][0]=5 ;  // tab[0][0]
pts[1][2]=5 ;  // tab[1][0]
pts[2][2]=5 ;  // tab[1][3]
**(pts+=5)=5 ; // tab[3][0]
;
ptl=(float (* )[NC+2])tab ;
;
ptl[0][0]=5 ;       // tab[0][0]
ptl[0][6]=5 ;       // tab[1][1]
(ptl+=2)[0][5]=5 ; // tab[3][4]
```

Przykład powyżej można traktować jako wskazówkę do ostatniego zadania.

Zadanie: Niech wskaźniki `pts` i `ptl` z ostatniego przykładu wskazują na początek tablicy `tab`. Jakie są zakresy dozwolonych indeksów `i` i `j`, dla których wyrażenia `pts[i][j]` lub `ptl[i][j]` odnoszą się do elementów tablicy `tab`?

Funkcje i struktury

W wersjach rozszerzonych języka C struktura (a nie tylko wskaźnik do struktury w C) może być zarówno argumentem, jak i wartością zwracaną przez funkcję.

```
#include<stdio.h>
#include<math.h>
;
struct vec { double x, y, z ; } ;
;
double scalprod(struct vec v1, struct vec v2)
 {
 return(v1.x*v2.x+v1.y*v2.y+v1.z*v2.z);
 }
;
double norm(struct vec x)
 {
 return(sqrt(scalprod(x,x)));
 }
;
struct vec vecprod(v1, v2)
 struct vec v1, v2 ;
 {
 struct vec result ;
 result.x=v1.y*v2.z-v1.z-v2.y ;
 result.y=v1.z*v2.x-v1.x-v2.z ;
 result.z=v1.x*v2.y-v1.y-v2.x ;
 return(result);
 }
;
void main(void)
 {
 double sp, n ;
 struct vec x={3,3,3}, y={2,2,2}, v ;
 ;
 sp=scalprod(x,y) ;
 n=norm(x) ;
 printf(" %.1f %.1f\n", sp, n);
 ;
 v=vecprod(x,y) ;
 printf(" %.1f %.1f %.1f\n", v.x, v.y, v.z);
 }
```

Zalecam jednak przekazywanie wartości i argumentów funkcji za pomocą wskaźników do struktur. Przepisanie przykładu powyżej tak, aby występowały w nim tylko wskaźniki do struktur jest już proste, dlatego nie przytaczam go tutaj.

Proszę zwrócić uwagę na to, że w powyższym programie defincja struktury `vec` umieszczona jest na zewnątrz wszystkich funkcji (także `main`). Dzięki temu obiekt ten rozumiany jest właściwie przez wszystkie funkcje w programie.

Funkcja `sqrt` występująca w definicji funkcji `norm` pochodzi ze standardowej biblioteki funkcji matematycznych. Jej prototyp znajduje się w pliku nagłówkowym `math.h`.

Adres zmiennej automatycznej

Jak pamiętamy wartości przekazywane funkcji przechowywane są w tymczasowych zmiennych. Zmienne te istnieją tylko w czasie wykonywania funkcji i nikną wraz z jej zakończeniem. Podobnie jest ze zmiennymi automatycznymi (zresztą w pewnym sensie argument formalny funkcji także jest zmienną automatyczną). Z tego powodu przekazywanie na zewnątrz funkcji adresów zmiennych automatycznych prowadzi zawsze do kłopotów podczas uruchamiania programu (kompilator niczego nie wykryje).

```
struct pos { float x, y ; } ;
;
struct pos funsvv(struct pos arg)
 {
 arg.x+=2 ; arg.y+=2 ;
 return(arg);
 }
;
struct pos funsvp(struct pos *arg)
 {
 arg->x+=2 ; arg->y+=2 ;
 return(*arg);
 }
;
struct pos *funspv(struct pos arg)
 {
 arg.x+=2 ; arg.y+=2 ;
 return(&arg);          // problems
 }
;
struct pos *funspp(struct pos *arg)
 {
 arg->x+=2 ; arg->y+=2 ;
 return(arg);
 }
;
void main(void)
 {
 struct pos r={2.2,2.2}, x, y, *p ;
 ;
 y=funsvv(x=r) ;
 printf(" %.1f %.1f\n", x.x, x.y);
 printf(" %.1f %.1f\n", y.x, y.y);
 ;
 x=r ;            // superfluous
 y=funsvp(&x) ;   // ok
 printf(" %.1f %.1f\n", x.x, x.y);
 printf(" %.1f %.1f\n", y.x, y.y);
 ;
 p=funspv(x=r) ;  // problems: look above
                  // p is not pointing at x
                  // it points to random
 printf(" %.1f %.1f\n", x.x, x.y);
 printf(" %.1f %.1f\n", p->x, p->y);
 ;
 x=r ;            // superfluous
 p=funspp(&x) ;   // ok: p is pointing at x
 printf(" %.1f %.1f\n", x.x, x.y);
 printf(" %.1f %.1f\n", p->x, p->y);
 }
```

Zarówno argumenty jak i wartość funkcji mogą być albo obiektem, albo wskaźnikiem do obiektu. Zatem istnieją cztery możliwe sposoby zdefiniowania danej funkcji. Powyżej przedstawione zostały te cztery możliwości.

Funkcja `funspv` zwraca adres argumentu formalnego, co prowadzi do nieprzewidywalnego działania programu podczas wywoływania tej funkcji.

Typ referencyjny

Typ referencyjny występuje jedynie w C++.

W definicji obiektu referencyjnego należy przed jego nazwą umieścić operator adresu `&`. Zasada jest dokładnie taka sama, jak i dla wskaźników.

Każdej zmiennej referencyjnej, podobnie jak każdej stałej, trzeba nadać wartość początkową.

```
int val=10 ;
int &refval1=val, &refval2=val ;
int &refval ;  // error: not initialized
```

Typ referencyjny, określany czasem jako ***odniesienie*** (nie mylić ze wskazaniem), służy do wprowadzenia dodatkowej nazwy dla obiektu użytego w definicji jako wartość początkowa. Wszystkie operacje dokonywane na zmiennej referencyjnej dotyczą także tego obiektu, do którego *odnosi się* zmienna referencyjna. Zatem po wykonaniu instrukcji

```
refval1+=3 ;   lub
refval2+=3 ;
```

wartość zmiennej `val` zwiększy się o `3` i wyniesie `13`. W instrukcji

```
int i=refval1 ;
```

zmiennej `i` zostaje przypisana bieżąca wartość zmiennej `val`, natomiast instrukcja

```
int *pi=&refval2 ;
```

inicjuje wskaźnik `p` adresem zmiennej `val`.

Nic nie stoi na przeszkodzie zdefiniowania odniesienia do wskaźnika, tak jak na przykład poniżej.

```
char *pc="first" ;
char *&refpc=pc ;
;
printf(" %s", refpc);
;
refpc="second" ;
printf(" %s", pc);
;
refpc+=2 ;
printf(" %s", pc);
```

Jak widać, typ referencyjny wykazuje pewne podobieństwo do typu wskaźnikowego. Niekiedy uważa się nawet typ referencyjny za szczególny rodzaj wskaźnika, mianowicie taki, dla którego powinno się stosować ten sam sposób zapisu wyrażen, co dla wskazywanego obiektu. Na przykład wartością wyrażenia

```
(*pval==refval && pval==&refval)
```

jest zawsze prawda, jeżeli wskaźnik `pval` wskazuje na ten sam obiekt, do którego odnosi się zmienna referencyjna `refval`. Jednak w przeciwieństwie do wskaźnika, obiekt referencyjny musi być zainicjowany, i nie może odnosić się do innego obiektu niż ten, który został określony jako wartość początkowa. Zbędne zatem byłoby deklarowanie obiektu referencyjnego w sposób jawny jako obiektu stałego, tak jak poniżej

```
int &const refval=val ;
```

ponieważ każda definicja obiektu referencyjnego określa go (niejawnie) jako obiekt stały.

```
int val=10, *pval ;
int &refval=val ;
;
refval+=2 ;     // 12
val=refval+2 ;  // 14
refval=val+2 ;  // 16
;
pval=&refval ;  // pval=&val ;
*pval+=3 ;      // 19
;
if(*pval==refval && pval==&refval)
  printf("true");
```

Wartością początkową obiektu referencyjnego zadeklarowanego jako odniesienie do obiektu nie będącego stałą, może być tylko l-wartość odpowiedniego typu. (W `djgpp` istnieją pewne odstępstwa od tej zasady.)

```
int val=4, idx=2 ;
int *pval=&val, &ref=*pval ;
;
ref+=2 ;
printf(" %d %d %d \n", val, ref, *pval);
;
pval=&idx ;
printf(" %d %d %d \n", val, ref, *pval);
ref+=2 ;
printf(" %d %d %d \n", val, ref, *pval);
```

Powyżej zmienną referencyjną `ref` inicjujemy obiektem, na który wskazuje zmienna wskaźnikowa `pval`. Uprzednio wskaźnikowi `pval` nadana została wartość adresu zmiennej `val`. Zatem `ref` odnosi się do zmiennej `val`. Wyrażenie `pval=&idx` sprawia, że `pval` zaczyna wskazywać na inny obiekt, na zmienną `idx`. Ale `ref` nadal odnosi się do zmiennej `val`!

Zadanie: Czy poniższy program będzie działał poprawnie?

```
int val=4, *pval, &ref=*pval ;
;
pval=&val ;
printf(" %d %d %d \n", val, ref, *pval);
ref+=2 ;
printf(" %d %d %d \n", val, ref, *pval);
```

Kilka niepoprawnych instrukcji inicjujących podajemy w przykładzie poniżej. (Kompilator C++ pakietu `djgpp` nie traktuje ich jednak jako błędne.)

```
int i=2, *pi ;
double d=4 ;
;                   // warning:
int &refi=1234 ;    // init from rvalue
double &refd=2*d ;  // init from rvalue
float &reff=i ;     // init with 'int'
;
refi=33 ;
refd=33 ;
;
pi=&refi ;
*pi=11 ;
;
reff=44.44 ;  // i==2 ;
```

Powyżej deklaracje zmiennych referencyjnych `refi`, `refd` oraz `reff` są w istocie równoważne definicjom *nowych* zmiennych. Zatem opuszczenie operatora `&` w instrukcjach deklaracyjnych dałoby dokładnie taki sam wynik.

Natomiast wartość początkowa obiektu referencyjnego zadeklarowanego jako odniesienie do obiektu stałego, nie musi być l-wartością. Przykłady poniżej.

```
char i=2, j=5 ;
const int &refci=1024 ;
const char &refcc=i+j ;
const double &refcd=i ;
```

Typów referencyjnych używa się najczęściej jako typów argumentów funkcji (lub wartości przekazywanych przez funkcje). Więcej o tym za chwilę.

Referencyjne argumenty funkcji

W C++ argumenty formalne funkcji mogą być typu referencyjnego.

Przykład: Katalog nazw zbiorów

Wskaźniki do funkcji

Wprawdzie w języku C funkcje nie są zmiennymi, ale istnieje możliwość definiowania wskaźników do funkcji. Na wskaźnikach takich można dokonywać operacji, przekazywać je funkcjom, wstawiać do tablic itd.

W języku C ma sens określenie ***typu funkcji***. Wyznaczają go jednoznacznie typ przekazywanej wartości i sygnatura (lista typów argumentów). Nazwa funkcji jest identyfikatorem i nie ma tu znaczenia. Deklaracja

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

zapowiada funkcję `func`, która jest typu `double ...( int, int *)` (zwraca wartość typu `double` oraz wymaga dwóch argumentów o typach `int` i `int *`). Deklaracja wskaźnika do funkcji tego typu wygląda następująco:

```
double (*pf)(int, int *) ;
```

Zatem obiekt `pf` jest wskaźnikiem do funkcji typu `double ...(int, int *)`. Obiektem tym można posługiwać się dokładnie w taki sam sposób, jak dowolną zmienną. Może on być np. przekazywany funkcji jako argument.

Zauważmy, że pierwsza para nawiasów okrągłych w deklaracji obiektu `pf` jest niezbędna. Bez nich deklaracja

```
double *pf(int, int *) ;
```

mówiłaby, że `pf` jest funkcją (a nie wskaźnikiem do funkcji) zwracającą wskaźnik do zmiennych typu `double`. Konieczność użycia nawiasów jest spowodowana wyższym priorytetem operatora wywołania funkcji `()` od operatora wyłuskania `*`.

Istnieje tyle różnych typów funkcji (więc także wskaźników do funkcji), ile jest różnych kombinacji sygnatur i typów przekazywanej wartości.

Podobnie jak identyfikator tablicy (bez nawiasów `[]`) oznacza wskazanie na pierwszy element tablicy, tak nazwa funkcji danego typu (bez operatora wywołania) ma wartość wskaźnika do funkcji tego typu. Nie ma znaczenia, czy nazwę funkcji poprzedza operator adresu `&`,

czy nie – w obu przypadkach wartością wyrażenia jest wskaźnik do funkcji. A więc wyrażenia `func` oraz `&func` są sobie równoważne.

Wskaźniki do funkcji inicjuje się podobnie jak poznane już przez nas wskaźniki do innych obiektów.

```
int mini(int *, int), maxi(int *, int) ;
double mind(double *, int) ;
double maxd(double *, int) ;
;
void main(void)
 {
 int (*pi)(int *, int)=0 ;
 double (*pd)(double *, int)=maxd ;
 ;
 if(!pi) pi=mini ;
 if(pi==mini) pi=&maxi ;
 ;
 pd=mind ;
 ;
 pi=maxd ;  // warning
 }
```

Jak widać powyżej wskaźnikom do funkcji można przypisywać wartości, także wartość zerową.

Warunkiem poprawności inicjowania lub przypisywania wartości wskaźnikowi do funkcji jest ścisła (tylko w C++) zgodność typu przypisywanego obiektu i typu wskaźnika, o której decydują listy typów argumentów i typy zwracanej wartości.

Jeśli `pf` jest wskaźnikiem do funkcji, to aby wywołać funkcję, na którą wskazuje, należy użyć operatora wyłuskania `*` następująco:

```
(*pf)(...)
```

(kropki `...` oznaczają ewentualną listę argumentów wywołania). Ujęcie wyrażenia `*pf` w nawiasy jest konieczne.

W wersjach rozszerzonych języka C wywołania funkcji określonej przez wskaźnik nie trzeba poprzedzać operatorem wyłuskania. Zapis w postaci uproszczonej `pf(...)` jest równoważny `(*pf)(...)`.

```
int *f(int *x, int n)
 {
 int i, *p=x, max=*x ;
 for(i=0 ; i<n ; i++)
   if(x[i]>max) { max=x[i] ; p=x+i ; }
 return(p);
 }
;
void main(void)
 {
 int tab[]=2,7,4,8,2,3,5 ;
 int ntab=sizeof(tab)/sizeof(int) ;
 int *pmax, max ;
 ;
 int *(*pf)(int *, int)=0 ;
 ;
 pmax=f(tab,ntab) ;
 max=*f(tab,ntab) ;
 printf(" %d %d\n", *pmax, max);
 ;
 pf=f ;
 pmax=(*pf)(tab,ntab) ;
 max=*(*pf)(tab,ntab) ;
 printf(" %d %d\n", *pmax, max);
 ;
 pmax=pf(tab,ntab) ;     // simplified
 max=*pf(tab,ntab) ;     // simplified
 printf(" %d %d\n", *pmax, max);
 ;
 max=**pf(tab,ntab) ;  // error
 }
```

Przykład powyżej pokazuje sposoby wywoływania funkcji wskazywanej przez wskaźnik.

Aby zadeklarować wskaźnik do funkcji, która zwraca wskaźnik do obiektu typu np. `double`, należy napisać

```
double *(*fun)() ;
```

Przykład użycia takiego wskaźnika podajemy poniżej.

```
char *strfun(int pos, char *str)
 {
 if(pos<1) pos=1 ;
 return(str+pos-1);
 }
;
void main(void)
 {
 char *(*pfun)(int, char *), *name ;
 ;
 pfun=&strfun ;
 ;
 name=(*pfun)(-10,"Gil-galad");
 printf(" %s\n", name);
 ;
 name=(*pfun)(5,name);
 printf(" %s\n", name);
 }
```

Tablice wskaźników do funkcji. Wskaźniki do wskaźników do funkcji

Oczywiście można zdefiniować także tablicę wskaźników do funkcji, wskaźnik do wskaźnika do funkcji itd.

Na przykład instrukcja

```
int (*ftab[ntab])(void) ;
```

deklaruje zmienną `ftab` jako tablicę zawierającą `ntab` elementów, z których każdy jest wskaźnikiem do funkcji nie posiadającej argumentów (sygnatura `void`) oraz przekazującej wartość typu `int`.

Uwaga: W języku C nie ma możliwości zdefiniowania tablicy funkcji. Ponadto wynikiem funkcji nie może być tablica albo funkcja oraz funkcja nie może być składową struktury.

Przykłady błędnych deklaracji podajemy poniżej.

```
int funtab()[4] ;
int funfun()() ;
int tabfun[3]() ;
struct { int structfun() ; } var ;
```

Wskaźnik do wskaźnika do funkcji typu np. `void ...(int)` definujemy następująco:

```
void (**ppf)(int) ;
```

Zmienna `ppf` wskazuje na wskaźniki do funkcji typu `void ...(int)`. Wartością wyrażenia `*ppf` jest zatem wskaźnik do funkcji typu `void ...(int)`. Natomiast `**pf` jest funkcją typu `void ...(int)`. Jak wygląda wywołanie tej funkcji łatwo się domyślić.

```
int min(int *x, int n)
 {
 int i, v=*x ;
 for(i=1 ; i<n ; i++) if(x[i]<v) v=x[i] ;
 return(v);
 }
;
int max(int *x, int n)
 {
 int i, v=*x ;
 for(i=1 ; i<n ; i++) if(x[i]>v) v=x[i] ;
 return(v);
 }
;
int sum(int *x, int n)
 {
 int i, v=0 ;
 for(i=0 ; i<n ; v+=x[i++]);
 return(v);
 }
;
void main(void)
 {
 int x[]=2,7,4,8,2,3,5, y, i ;
 int nx=sizeof(x)/sizeof(int) ;
 ;
 const int ntab=3 ;
 int (*ftab[])(int *, int)={min,max,sum} ;
 ;
 int (**ppf)(int *, int)=ftab ;
 ;
 y=sum(x,nx) ;
 ;
 for(i=0 ; i<ntab ; i++)
   {
   y=(*ftab[i])(x,nx) ; // or
   y=ftab[i](x,nx) ;
   printf(" %d ", y);
   }
 ;
 y=(*ppf[2])(x,nx) ;    // or
 y=(**(ppf+2))(x,nx) ;  // or
 y=ppf[2](x,nx) ;
 printf(" %d", y);
 ;
 for(i=0 ; i<ntab ; i++)
   {
   y=(**ppf++)(x,nx) ;
   printf(" %d ", y);
   }
 ppf-=ntab ;
 }
```

Proszę zwrócić uwagę na inicjalizację tablicy wskaźników do funkcji w przykładzie powyżej oraz na operacje na wskaźnikach.

Proszę zwrócić uwagę na różnice w zapisie deklaracji różnych obiektów podanych poniżej.

```
float *f() ;
float (*pf)() ;
float *(*pfp)() ;
float (**ppf)() ;
```

Powyżej `f` jest funkcją zwracającą wskaźnik do obiektów typu `float`, `pf` jest wskaźnikiem do funkcji zwracającej wartości typu `float`, `pfp` jest wskaźnikiem do funkcji, która zwraca wskaźnik do obiektów typu `float`, a `ppf` jest wskaźnikiem do wskaźnika do funkcji zwracających wartości typu `float`.

Deklaratory

Deklaratory służą do deklarowania identyfikatorów obiektów. Zatem pojawiają się one w definicjach i deklaracjach obiektów (zmiennych i funkcji), czyli instrukcjach postaci

class-specifier type-specifier declarators ;

gdzie ***class-specifier*** jest oznaczeniem klasy pamięci a ***type-specifier*** jest oznaczeniem typu danych.

Najprostszy deklarator składa się z samego identyfikatora deklarowanego obiektu.

```
static int n ;
```

Jeśli pewne wyrażenie ***dec*** jest deklaratorem, to deklaratorem jest także wyrażenie `(`***dec***`)` oznaczające to samo co ***dec***, wyrażenie ***dec***`[N]` oznaczające tablicę o `N` elementach, wyrażenie `*`***dec*** oznaczające wskaźnik oraz wyrażenie ***dec***`()` oznaczające funkcję.

Jak widzieliśmy wcześniej, niekiedy konieczne jest użycie nawiasów półokrągłych w celu wymuszenia właściwej interpretacji deklaratora.

```
int *tp[5], (*pt)[5] ;
float *fp(), (*pf)() ;
```

Analizę złożonego deklaratora należy zaczynać od identyfikatora i prowadzić ją (z uwzględnieniem priorytetów) od wnętrza ku zewnętrzu deklaracji.

Więcej przykładów złożonych deklaratorów zamieszczam w następnym rozdziale.

Wskaźnik do funkcji a definicja funkcji

W języku C dozwolone jest deklarowanie argumentów funkcji będących wskaźnikami do funkcji.

```
int sum(int *x, int n)
 {
 int i, v=0 ;
 for(i=0 ; i<n ; v+=x[i++]);
 return(v);
 }
;
int fun(x, n, f)
```

```
int *x, n, (*f)(int *, int) ;
 {
 int i ;
 for(i=0 ; i<n ; x[i++]+=2);
 return((*f)(x,n));
 }
;
void main(void)
 {
 int x[]=2,7,4,8,2,3,5, y ;
 int nx=sizeof(x)/sizeof(int) ;
 ;
 y=sum(x,nx) ;
 printf(" %d ", y);  // 31
 ;
 y=fun(x,nx,sum) ;
 printf(" %d ", y);  // 45
 ;
 y=fun(x,nx,&sum) ;
 printf(" %d ", y);  // 59
 }
```

W wersjach rozszerzonych C również typem wartości przekazywanej przez funkcję może być wskaźnik do funkcji. Na przykład w instrukcji

```
int (*fun(double))(int *, int) ;
```

zadeklarowano `fun` jako funkcję mającą jeden argument typu `double`. Funkcja ta przekazuje wskaźnik do funkcji typu `int ...(int *, int)`.

Ogólnie rzecz biorąc funkcja może zwracać wskaźnik do obiektu dowolnego typu, także do tablic.

```
const int NR=6, NC=9 ;
// ;
float (*funpt())[NC] ;
float (*funptt())[NR][NC] ;
// ;
float (*(*pfunpt)())[NC] ;
float (*(*pfunptt)())[NR][NC] ;
// ;
float (*ptab)[NC], (*pttab)[NR][NC] ;
float (*tabpt[NR])[NC] ;
;
pfunpt=&funpt ;
pfunptt=&funptt ;
;
ptab=funpt() ;
pttab=funptt() ;
;
ptab=(*pfunpt)() ;
pttab=(*pfunptt)() ;
;
*tabpt=funpt() ;
```

Powyżej `funpt` zwraca wskaźnik do tablic typu `float [NC]`, a `funptt` zwraca wskaźnik do tablic typu `float [NR][NC]`. Natomiast `pfunpt` jest wskaźnikiem do funkcji typu `float (*...())[NC]`, a `pfunptt` jest wskaźnikiem do funkcji typu `float (*...())[NR][NC]`. Zauważmy również, że dwie pierwsze deklaracje to deklaracje zapowiadające funkcji, a kolejne dwie deklaracje to definicje zmiennych wskaźnikowych.

```
#define NR 60
#define NC 90
;
void strcopy(char *dest, char *source)
 { while((*dest++=*source++)); }
;
void main(void)
 {
 int n=0 ;
 char (*fun(int, char (* )[NC]))[NC] ;
 char arr[NR][NC], (*pos)[NC] ;
 ;
 strcopy(arr[n++],"Grzegorz");
 strcopy(arr[n++],"Piotr");
 strcopy(arr[n++],"Jan");
 strcopy(arr[n++],"Agata");
 strcopy(arr[n++],"Kazimierz");
 ;
 pos=fun(2,arr) ;
 ;
 printf(" %c\n", **pos);           // J
 printf(" %c\n", **fun(2,arr));    // J
 printf(" %s\n", fun(2,arr)[0]);  // Jan
 ;
 **fun(0,arr)='X' ;
 printf(" %s\n", *fun(0,arr)); // Xrzegorz
 ;
 ++**fun(0,arr) ;
 printf(" %s\n", *fun(0,arr)); // Yrzegorz
 ;
 fun(0,arr)[4][2]='S' ;
 printf(" %s\n", *fun(4,arr)); // KaSimierz
 ;
 strcopy(*fun(0,arr),"Tomasz");
 printf(" %s\n", *fun(0,arr)); // Tomasz
 }
;
char (*fun(int n, char (*tab)[NC]))[NC]
 { return(tab+=n); }
```

W przykładzie powyżej zdefiniowana została funkcja `fun`, która zwraca wskaźnik do `NC`-elementowej tablicy znaków. Proszę zwrócić szczególną uwagę na instrukcje przypisania, w których wywołanie funkcji `fun` następuje z lewej (a nie jak zwykle prawej!) strony operatora `=`.

```
void funsum(int *a, int *b, int n)
 {
 int i ;
 for(i=0 ; i<n ; i++) a[i]+=b[i] ;
 }
;
void funsub(int *a, int *b, int n)
 {
 int i ;
 for(i=0 ; i<n ; i++) a[i]-=b[i] ;
 }
;
void funmul(int *a, int *b, int n)
 {
```

```
  int i ;
  for(i=0 ; i<n ; i++) a[i]*=b[i] ;
  }
 ;
 void fundiv(int *a, int *b, int n)
  {
  int i ;
  for(i=0 ; i<n ; i++)
    if(b[i]) a[i]/=b[i] ;
    else a[i]=0 ;
  }
 ;
 void (*fun(int idx))(int *, int *, int)
  {
  switch(idx)
    {
    case(1): return(&funsum);
    case(2): return(&funsub);
    case(3): return(&funmul);
    case(4): return(&fundiv);
    default: return(0);
    }
  }
 ;
 void showtabs(int *a, int *b, int n)
  {
  int i ;
  for(i=0 ; i<n ; i++)
    printf(" %3d %3d\n", a[i], b[i]);
  printf("\n");
  }
 ;
 void main(void)
  {
  void (*sub)(int *, int *, int) ;
  // ;
  int A[]={1,2,3,4,5,6,7,5,3} ;
  int B[]={2,3,6,7,9,0} ;
  int N, i ;
  ;
  N=sizeof(A) ;
  if(sizeof(B)<N) N=sizeof(B) ;
  N/=sizeof(int) ;
  ;
  sub=fun(1) ;
  (*sub)(A,B,N);
  showtabs(A,B,N);
  ;
  (*fun(4))(A,B,N);
  showtabs(A,B,N);
  ;
  i=8 ;
  if((sub=fun(i))) (*fun(i))(A,B,N);
  showtabs(A,B,N);
  ;
  // (*fun(8))(A,B,N); // dangerous!
  }
```

W przykładzie powyżej zdefiniowano funkcję `fun`, która zwraca wskaźnik do funkcji typu `void ...(int *, int *, int)`. Dlaczego ostatnia instrukcja (wykomentowana) jest niebezpieczna?

Zadanie: Zadeklarować wskaźnik do wskaźnika do funkcji, która zwraca wskaźnik do funkcji wymagającej argumentów: `int` i `float *` oraz zwracającej wskaźnik do obiektów typu `double`.

Odpowiedź:

```
double (*(**var))(int, float *) ;
```

Zadanie: Zadeklarować wskaźnik do tablicy `NR` wskaźników do funkcji zwracających wskaźnik do danych typu `short int`.

Odpowiedź:

```
short int *(*(*var)[NR])() ;
```

Zadanie: Jakie obiekty zostały zadeklarowane poniżej.

```
float (*(*VarA)(float))(void) ;
float *(*VarB())(int) ;
float (*(*VarC())[NR])(char) ;
```

Odpowiedź: `VarA` jest wskaźnikiem do funkcji, która wymaga argumentu typu `float` oraz zwraca wskaźnik do funkcji typu `float ...(void)`. `VarB` jest funkcją, która zwraca wskaźnik do funkcji typu `float *...(int)`. `VarC` jest funkcją zwracającą wskaźnik do tablicy `NR` wskaźników do funkcji typu `float ...(char)`.

Funkcje o zmiennej ilości argumentów

Na razie pomijam, ale wrócę kiedyś do tego.

Przeciążanie nazw funkcji

Występuje tylko w C++. Pomijam.

Rekurencja

Argumenty wywołania programu

W języku C możliwe jest przekazywanie argumentów wywołania wykonywanemu programowi. Działanie programu zaczyna się wtedy od funkcji `main` z dwoma argumentami. Pierwszy określa ilość argumentów wywołania programu (łącznie z nazwą programu), a drugi jest listą argumentów w postaci wskaźnika do tablicy tekstów znakowych (każdy argument traktowany jest jako osobny tekst).

```
void main(int narg, char **arg)
 {
 int i ;
 for(i=0 ; i<narg ; i++)
   printf(" %s\n", arg[i]);
 }
```

Przyjmuje się, że `arg[0]` jest nazwą, z jaką program został wywołany, a więc `narg` jest równy co najmniej `1`. Jeżeli powyższy program (nazwijmy go `echo`) uruchomimy następująco:

```
echo fine girl
```

to `narg` będzie równy `3`, a `arg[0]`, `arg[1]` i `arg[2]` będą wskazywać odpowiednio napisy `"echo"`, `"fine"` oraz `"girl"`. Pierwszym 'prawdziwym' argumentem jest `arg[1]`, a ostatnim `arg[narg-1]`. Oczywiście, wartość `narg` równa `1` oznacza, że w wierszu komend po nazwie programu nie było argumentów.

Napiszmy prosty program `stat.c`, który będzie wyznaczał podstawowe parametry statystyczne zestawu liczb, podawanych programowi jako argumenty jego uruchomienia.

```
#include<stdio.h>
#include<stdlib.h>
;
void main(int narg, char **arg)
 {
 int i ;
 double sum, avr, min, max, v ;
 if(narg==1)
   { printf(" usage: stat <n> ...\n");
   return ; }
 for(sum=0, i=1 ; i<narg ; i++)
   {
   v=atof(*++arg) ;
   if(i==1) { min=max=v ; }
   if(v>max) max=v ;
   if(v<min) min=v ;
   sum+=v ;
   }
 avr=sum/(narg-1) ;
 printf(" min=%lf max=%lf\n", min, max);
 printf(" sum=%lf avr=%lf\n", sum, avr);
 }
```

Po skompilowaniu program `stat` możemy uruchomić na przykład tak:

```
stat 2 3 4 5 6 7 8 9
```

W powyższym przykładzie pojawia się funkcja `atof` z biblioteki standardowej `stdlib.h`, która przekształca tekst reprezentujący liczbę na liczbę zmiennoprzecinkową.

Zmienne zewnętrzne

Zmienne w dowolnej funkcji są dla niej prywatne, co oznacza, że żadna inna funkcja nie ma do nich jakiegokolwiek dostępu. Są to tzw. zmienne ***automatyczne*** (była o nich mowa wcześniej). Każda taka zmienna lokalna pojawia się w momencie wywołania funkcji i znika po zakończeniu jej działania. Zatem zmienne automatyczne (pomijając te zadeklarowane jako `static`) nie zachowują swoich wartości pomiędzy kolejnymi wywołaniami funkcji i muszą być jawnie inicjowane w definicji funkcji. Jeśli ich wartości początkowe nie zostaną określone, to zmienne będą zawierać śmiecie.

Zmienne lokalne, czyli argumenty funkcji i zmienne automatyczne moglibyśmy określić jako 'wewnętrzne', ponieważ definiowane są wewnątrz funkcji.

Przeciwieństwem zmiennych automatycznych są ***zmienne zewnętrzne***. Ten typ zmiennych swą nazwę zawdzięcza temu, że znajdują się one poza wszystkimi funkcjami w programie. Są to przez domniemanie zmienne globalne, dostępne przez nazwę w dowolnej funkcji, w której są potrzebne (także w funkcjach tłumaczonych oddzielnie). Później zobaczymy jak się tworzy zmienne zewnętrzne (i funkcje) niedostępne globalnie: widziane jedynie wewnątrz jednego pliku źródłowego.

Ze względu na tę ogólną dostępność, zmienne zewnętrzne mogą być stosowane zamiast list argumentów przy przekazywaniu danych między funkcjami. Jest to szczególnie wygodne dla funkcji wymagających dostępu do dużej liczby wspólnych danych.

Ponadto, ponieważ zmienne zewnętrzne istnieją stale (pojawiają się w momencie definicji i trwają do końca programu, a nie do końca pewnej funkcji), więc zachowują swoje wartości nawet po zakończeniu działania funkcji, które im te wartości nadały.

Zauważmy, że funkcje w języku C są zawsze zewnętrzne.

Zmienna zewnętrzna musi być zdefiniowana na zewnątrz wszystkich funkcji. Zmienna taka musi być także zadeklarowana w każdej funkcji, która ma mieć do niej dostęp. Można to zrobić poprzez jawną deklarację `extern` albo niejawnie przez kontekst.

Przykład zarówno jawnej, jak i niejawnej deklaracji zmiennej zewnętrznej w funkcji podany jest poniżej.

```
int func(int x)
 {
 extern int cst ;
 return(x+cst);
 }
;
int cst=3 ;
;
void main(void)
 {
 int x=5 ;
 ;
 x=func(x) ;
 ;
 cst=-3 ;
 x=func(x) ;
 }
```

Jak widać, jawna deklaracja jest konieczna wtedy, gdy definicja zmiennej znajduje się za definicją funkcji. Jest tak dla funkcji `fun`. Natomiast w funkcji `main` deklaracja `extern` została pominięta, ponieważ definicja (a więc także deklaracja) zmiennej `cst` pojawia się przed jej użyciem w funkcji `main`. Związane jest to z ***zasięgiem*** zmiennej zewnętrznej (więcej o zasięgu dalej).

Dostęp do zmiennej zewnętrznej

Jeżeli w programie zdefiniowane zostały dwie zmienne o takiej samej nazwie (powiedzmy `var`), jedna zewnętrzna, a druga automatyczna (lokalna dla pewnej funkcji, np. `fun`), to dostęp do zmiennej zewnętrznej z wnętrza funkcji `fun` możliwy jest jedynie w oddzielnym bloku (ograniczonym nawiasami `{}`), tak jak w przykładzie poniżej.

```
float var=3.0 ;
;
void fun(void)
 {
 int var=2 ;
 {
 extern float var ;
 var+=5.7 ;          // var=3.0+5.7 ;
```

```
    }
    var+=6 ;              // var=2+6 ;
    }
```

Powyżej wewnętrzna para nawiasów klamrowych wyznacza zakres treści funkcji `fun`, w którym nazwa `var` oznacza zmienną zewnętrzną. Poza tym zakresem `var` odnosi się do zmiennej automatycznej.

Zadanie: Co by się stało, gdyby w przykładzie powyżej pominięte zostało słowo `extern`?

```
char letter='c' ;
;
void main(void)
 {
 char letter='s', idx ;
 ;
 {
 extern char letter ;
 printf(" %c\n", letter++);  // 'c'
 }
 ;
 printf(" %c\n", letter);    // 's'
 ;
 {
 extern char letter ;
 printf(" %c\n", letter);    // 'd'
 }
 ;
 letter=2 ;
 { extern char letter ; letter=4 ; }
 ;
 idx=letter ;
 { extern char letter ; idx+=letter ; }
 printf(" %d\n", idx);  // 6
 ;
 idx=letter ;
 idx+=letter ;
 printf(" %d\n", idx);  // 4
 }
```

Reguły dostępu i zasięg

Funkcje i zmienne zewnętrzne tworzące dany program nie muszą znajdować się w jednym pliku (a więc nie muszą być tłumaczone razem). Tekst źródłowy programu może być podzielony na kilka części zawartych w odrębnych plikach. Dodatkowo niektóre funkcje mogą być dołączane z bibliotek standardowych.

Zasięgiem nazwy jest ta część programu, dla której zmienna o podanej nazwie została zdefiniowana.

Dla zmiennej automatycznej zadeklarowanej na początku treści funkcji zasięgiem jest cała definicja tej funkcji. Zatem w różnych funkcjach zmienne o takiej samej nazwie nie są ze sobą związane w jakikolwiek sposób. To samo dotyczy argumentów formalnych funkcji.

Zasięg zmiennej zewnętrznej rozciąga się od miejsca, w którym została zadeklarowana w pliku źródłowym do końca pliku. Na przykład, jeśli `nx`, `x`, `avr`, `min` i `max` są zdefiniowane w jednym pliku w następującej kolejności:

```
int nx ;
double *x ;
;
double min(void) { ... }
double max(void) { ... }
double avr(void) { ... }
```

to funkcje `avr`, `min` i `max` mogą odwoływać się do zmiennych `nx` i `x` po prostu przez nazwy. Żadne dodatkowe deklaracje nie są w tym przypadku potrzebne.

Z drugiej strony, jeśli odwołanie do zmiennej zewnętrznej wystąpi przed jej definicją, bądź jeśli jej definicja znajduje się w innym pliku źródłowm, to wymagana jest deklaracja `extern`. Zatem jeśli definicje zmiennych zewnętrznych znajdują się za funkcją, która z tych zmiennych korzysta, to należy zapowiedzieć te zmienne deklaracjami `extern`, tak jak w przykładach poniżej:

```
extern int nx ;
extern double *x ;
;
double avr(void) { ... }
;
int nx ;
double *x ;
;
double min(void) { ... }
double max(void) { ... }
```

lub

```
double avr(void)
 {
 extern int nx ;
 extern double *x ;
 ...
 }
;
int nx ;
double *x ;
;
double min(void) { ... }
double max(void) { ... }
```

We wszystkich plikach źródłowych składających się na jeden program może wystąpić tylko jedna definicja zmiennej zewnętrznej. Wszystkie pliki (także ten z definicją) mogą zawierać deklaracje `extern` zapewniające dostęp do zmiennej.

Nadanie zmiennej zewnętrznej dowolnej wartości początkowej jest możliwe jedynie podczas jej definiowania. Definicje tablic muszą określać ich rozmiar, natomiast w deklaracjach `extern` rozmiary można pominąć. Ponadto można inicjować tablice zewnętrzne, natomiast nie można inicjować tablic automatycznych.

Gdyby zmienne `nx` i `x` były zdefiniowane (i zainicjowane) w jednym pliku źródłowym, a funkcje `avr`, `min` oraz `max` w innym, wówczas aby powiązać wszystko poprawnie w całość, należałoby podać następujące definicje i deklaracje:

w pliku pierwszym (nazwijmy go np. `vars.c`):

```
int nx=0 ;
double *x ;
```

w pliku drugim, funcs.c:

```
extern int nx ;
extern double *x ;
;
double avr(void) { ... }
double min(void) { ... }
double max(void) { ... }
```

W pliku funcs.c deklaracje extern umieszczone zostały przed definicjami funkcji, dotyczą zatem wszystkich trzech funkcji. Jak widzimy jeden zestaw deklaracji zmiennych zewnętrznych wystarcza na całą zawartość pliku funcs.c.

Pliki vars.c i funcs.c można przetłumaczyć oddzielnie, a następnie połączyć za pomocą linkera.

Istnieje także możliwość wstawienia zawartości jakiegoś pliku za pomocą dyrektywy include preprocesora języka C. W takim przypadku plik funcs.c mógłby wyglądać następująco:

```
#include"vars.c"
;
double avr(void) { ... }
double min(void) { ... }
double max(void) { ... }
```

Zauważmy, że nie ma teraz potrzeby wstawiania deklaracji extern.

Zmienne i funkcje statyczne

Zmienne statyczne tworzą trzecią klasę pamięci, uzupełniającą klasy zmiennych automatycznych i zewnętrznych.

Zmienną statyczną deklaruje się za pomocą słowa kluczowego static.

```
static int n ;
static double x[333] ;
```

Zmienne statyczne mogą być zarówno wewnętrzne, jak i zewnętrzne. Zmienna jest wewnętrzna, gdy została zdefiniowana wewnątrz pewnej funkcji, jest zewnętrzna, jeśli jej definicja wystąpiła poza wszystkimi funkcjami.

Wewnętrzne zmienne static są lokalne dla funkcji tak samo, jak zwykłe zmienne automatyczne. Jednak w przeciwieństwie do zmiennych automatycznych nie pojawiają się wraz z wywołaniem funkcji i nie znikają po zakończeniu funkcji, lecz istnieją między jej wywołaniami. Oznacza to, że wewnętrzne zmienne statyczne stanowią prywatną, stałą pamięć funkcji.

```
void fun(int arg)
 {
 static int x ;
 if(arg>0) x++ ;
 else x=0 ;
 printf(" %d ", x);
 }
;
void main(void)
 {
 fun(-1);  // 0
 fun(2);   // 1
 fun(2);   // 2
 fun(-2);  // 0
 fun(5);   // 1
 fun(5);   // 2
 }
```

Zewnętrzna zmienna statyczna znana jest w pliku źródłowym od miejsca, w którym ją zadeklarowano, natomiast nie jest znana w żadnym innym pliku. Deklarowanie zewnętrznych obiektów statycznych jest zatem sposobem na ukrywanie ich nazw.

Funkcje zwykle są obiektami globalnymi. Istnieje jednak możliwość zadeklarowania funkcji jako static. Wówczas jej nazwa nie będzie znana na zewnątrz pliku zawierającego jej definicję.

Niech plik static.c zawiera:

```
static int param=4 ;
static void sfun(int *arg) { (*arg)++ ; }
;
int efun(int *arg)
 { sfun(arg); return(++param); }
```

a plik main.c:

```
int param=0 ;
void sfun(int *arg) { (*arg)-- ; }
;
void main(void)
 {
 int x=2, y, efun(int *) ;
 y=efun(&x);
 printf(" %d %d %d\n", param, x, y);
 param++ ;
 sfun(&x);
 printf(" %d %d %d\n", param, x, y);
 }
```

Wtedy oba pliki można bez przeszkód przetłumaczyć, a następnie połączyć. Funkcja main z pliku main.c może mieć dostęp jedynie do funkcji efun. Pozostałe obiekty z pliku static.c, a więc param oraz sfun, są statyczne, zatem zdefiniowane jedynie wewnątrz pliku static.c i nie są poza nim widoczne. Zauważmy, że globalna funkcja efun może swobodnie korzystać z obiektów statycznych, zarówno zmiennej param, jak i funkcji sfun.

W pliku main.c możliwe jest zdefiniowanie obiektów o takich samych nazwach jak obiekty statyczne z pliku static.c. Nie prowadzi to do jakiegokolwiek konfliktu.

W języku C słowo static oznacza nie tylko trwałość, lecz także stopień 'prywatności'. Wewnętrzne obiekty statyczne znane są tylko jednej funkcji. Zewnętrzne obiekty statyczne są znane jedynie wewnątrz pliku, w którym są definiowane: ich nazwy nie mają związku z funkcjami i zmiennymi o takich samych nazwach w innych plikach.

Zewnętrzne zmienne i funkcje deklarowane jako static pozwalają ukryć pewne obiekty oraz wszystkie mechanizmy ich obsługi, by uniemożliwić nawet przypadkowy konflikt nazw z innymi funkcjami i danymi.

Klasy pamięci: podsumowanie

Oznaczenia klasy pamięci mogą dotyczyć zarówno identyfikatorów zmiennych, jak i funkcji. Określają one zakres dostępności obiektu oznaczonego danym identyfikatorem, a w przypadku zmiennych określają ponadto warunki utworzenia i sposób traktowania obiektu.

Zaleca się aby w deklaracjach słowa kluczowe oznaczające klasę obiektu (`auto`, `static`, `extern`, `register`) umieszczane były przed oznaczeniami typu danych, np.

```
auto int x, y, z ;
static long int index, fun() ;
extern double **ppd ;
```

Domyślnym typem obiektu dowolnej klasy jest typ `int`.

```
auto x, y, z ;
static index, fun() ;
extern **ppd ;
```

Klasą domyślną dla zmiennych wewnętrznych (zdefiniowanych w obrębie treści pewnej funkcji) jest klasa `auto` (zmienne automatyczne), dlatego słowo `auto` zwykle się pomija.

Obiekty klasy `auto` są tworzone oraz inicjowane podczas każdego wykonania początku instrukcji grupującej zawierającej deklarację identyfikatora i są usuwane podczas wykonywania zakończenia tej instrukcji.

```
int count=-1 ;
;
while(count++<3)
  {
  auto char idx ;
  if(!count) { idx='G' ; continue ; }
  putchar(idx++);
  }
```

Mimo że w większości implementacji języka C wykonanie programu powyżej powoduje wyprowadzenie napisu `GHI`, zgodnie z tym co powiedziano wcześniej jest on programem błędnym, ponieważ przy wykonywaniu drugiego i trzeciego kroku pętli `while` zmienna `idx` ma wartość nieokreśloną. Natomiast wykonanie programu następującego

```
int count=-1 ;
;
while(count++<3)
  {
  auto char idx='G' ;
  putchar(idx++);
  }
```

spowoduje wyprowadzenie napisu `GGG`. Dlaczego?

Obiekty klasy `static` są tworzone i inicjowane podczas wykonywania prologu programu i są usuwane podczas wykonywania jego epilogu. Zatem w przeciwieństwie do zmiennych automatycznych są one inicjowane tylko raz. Ponadto zmienne statyczne można inicjować jedynie obiektami stałymi (nie mogą to być np. zmienne).

```
char *name="Grzegorz" ;
;
while(*name)
  {
  static char letter=0 ;
  if(!letter) letter=*name ;
  putchar(letter);
  letter=*++name ;
  }
```

Wykonanie przykładu powyżej prowadzi do wypisania na standardowym urządzeniu wyjścia teksu `Grzegorz`.

Zadanie: Co by się stało, gdyby zmienna `letter` w przykładzie wyżej była klasy `auto`?

Obiekty klasy `extern` definiuje się na zewnątrz definicji wszystkich funkcji. Jeśli pewien obiekt zostanie zdefiniowany bez atrybutu klasy oraz na poziomie definicji funkcji, to domyślnie zostanie mu przydzielona klasa `extern`. Definicja obiektu zewnętrznego o danym identyfikatorze może pojawić się w tekście programu tylko raz. Wszystkie deklaracje tego samego identyfikatora z atrybutem `extern` mają jedynie charakter odwoławczy i mogą występować wielokrotnie.

Podobnie jak obiekty klasy `static`, obiekty klasy `extern` są tworzone jednorazowo, podczas wykonywania prologu programu i są usuwane podczas wykonywania jego epilogu.

Obiekty klasy `register` umieszczane są, o ile jest to możliwe, w szybkich rejestrach procesora. Obiekty te różnią się od obiektów klasy `auto` jedynie tym, że nie można pobierać ich adresów. Użycie zmiennych rejestrowych może znaczanie przyspieszyć wykonanie programu, jeżeli ze zmiennych tych często się korzysta. Dlatego zmiennymi rejestrowymi powinny być zwłaszcza indeksy tablic i wskaźniki wewnątrz pętli.

```
const int NR=300, NR=700 ;
;
register int i, j ;
int tab[NR][NC] ;
;
for(i=0 ; i<NR ; i++)
  for(j=0 ; j<NC ; j++) tab[i][j]=i+j ;
```

Argument formalny funkcji również może zostać zadeklarowany jako zmienna rejestrowa.

```
void fun(register char *str)
 { while(*str) *str+++=2 ; }
```

Jeśli danego obiektu nie można umieścić w rejestrze, to traktowany jest jak obiekt klasy `auto`.

Struktura blokowa

Język C nie ma struktury blokowej, ponieważ nie można definiować w nim funkcji wewnątrz funkcji. Jednakże zmienne mogą być definiowane według reguł struktury blokowej. Deklaracje zmiennych można umieszczać zaraz po otwierającym nawiasie klamrowym dowolnej instrukcji złożonej (grupującej), a nie tylko po tym, który rozpoczyna treść funkcji. Tak zadeklarowane zmienne zasłaniają identycznie nazwane zmienne z bloków zewnętrznych i istnieją do napotkania odpowiedniego zamykającego nawiasu klamrowego.

```
int i=5 ;
;
if(1)
```

```
    {
    int i=2, n ;
    for(n=0 ; n<5 ; n++) i++ ;
    printf(" %d ", i); // 7
    }
  ;
  i-=3 ;
  printf(" %d ", i);   // 2
  ;
  // n++ ; // error: undeclared
```

W przykładzie wyżej zasięgiem zmiennej `n` jest gałąź prawdy instrukcji `if`. Zmienna `i` zadeklarowana wewnątrz tego bloku zasłania zmienną `i` zadeklarowaną wcześniej; obie zmienne pomimo takiej samej nazwy nie są ze sobą zupełnie związane.

Te same reguły struktury blokowej dotyczą także zmiennych zewnętrznych oraz argumentów formalnych funkcji.

```
float x ;
;
void fa(void)
 {
 auto char x ;
 // ...
 }
;
void fb(register int x)
 { /*...*/ }
```

Powyżej, zmienna `x` wewnątrz funkcji `fa` jest odwołaniem do zmiennej automatycznej typu `char`. Zmienna `x` z definicji funkcji `fb` jest odwołaniem do parametru formalnego, a nie do zmiennej zewnętrznej.

Pliki nagłówkowe

Pliki nagłówkowe służą przede wszystkim do przechowywania deklaracji zapowiadających funkcji oraz definicji zmiennych globalnych.

Preprocesor języka C

Biblioteka funkcji wejścia-wyjścia

Biblioteka funkcji matematycznych

Inne pożyteczne funkcje