Materiały

Waldemar Hebisch, Adam Szustalewicz
Krzysztof Tabisz

Instytut Matematyczny
Uniwersytet Wrocławski

1999

Spis treści

1 Rzeczy podstawowe – pierwszy program

Program w języku C jest zbudowany wyłącznie z *funkcji*. Jednej funkcji lub kilku, które realizują poszczególne fragmenty procesu obliczeniowego. Każdy program musi zawierać funkcję o nazwie `main` (główna). Działanie programu rozpoczyna się od początku funkcji `main`, a kończy się po wykonaniu ostatniej jej *instrukcji*.

Treść funkcji `main` stanowi więc jakby główną treść programu i na ogół w niej są wywoływane różne inne funkcje wykonujące określone zadania, a które zostały opracowane w celu zwiększenia przejrzystości programu. Jedne funkcje mogą pochodzić z tego samego programu, inne z bibliotek funkcji napisanych wcześniej.

Listę argumentów funkcji ujmuje się w parę nawiasów okrągłych.

Dla wygody użytkownika w programie można umieszczać *komentarze*, których treść jest ignorowana przez kompilator:

DEF. Komentarz rozpoczyna para następujących bezpośrednio po sobie znaków `/*`, a kończy najbliższa para znaków `*/`. (Takie wyróżnione sekwencje znaków nazywamy **ogranicznikami**.) Treść takiego komentarza może mieścić się w kilku wierszach programu.

Jest jeszcze druga, prostsza postać komentarza, pochodząca z języka C++, którego długość nie może przekraczać jednego wiersza programu: początkowym ogranicznikiem takiego komentarza są dwa znaki dzielenia: `//`, a końcowym – najbliższe przejście do nowej linii. Dla wygody będziemy stosować i tę postać komentarza. □

1.1 Przykład programu

```
/* prog1.c */                                  /*  1 */
#include <stdio.h>                             /*  2 */
main()                                         /*  3 */
{                                              /*  4 */
printf("   *************************\n");      /*  5 */
printf("   *                       *\n");      /*  6 */
printf("   *  Witaj przy monitorze  *\n");      /*  7 */
printf("   *                       *\n");;;    //  8
printf("   *************************\n");      //  9
return 0;                                      // 10
}                                              // 11
```

1.1.1 Uwagi do programu

1. Poszczególne wiersze programu zostały ponumerowane kolejnymi liczbami umieszczonymi w komentarzach. Jak widać – wykorzystane zostały komentarze obydwu rodzajów.

2. Treść funkcji składa się z ciągu instrukcji umieszczonych pomiędzy nawiasami klamrowymi `{}` (wiersze 4 – 11). Kolejne instrukcje zakończone są średnikami. W wierszu 8, średniki następujące bezpośrednio po sobie, stanowią **instrukcje puste**.

3. Funkcja printf jest funkcją standardową języka C. W tym programie wyprowadza ona jedynie **łańcuch**, czyli tekst umieszczony pomiędzy znakami cudzysłowu, na standardowe wyjście komputera, którym jest (ekran monitora). Uwaga: cudzysłów nie jest częścią łańcucha! Znak specjalny `\n`, występujący w łańcuchu, jest w języku C poleceniem przejścia do początku nowego wiersza. Kilka innych znaków specjalnych w C podajemy dalej w rozdz. 3.1, przy opisie funkcji printf.

4. *dyrektywa* `#include <stdio.h>` w wierszu 2 informuje kompilator, że w programie mogą znajdować się funkcje opisane w pliku stdio.h (w tym wypadku jest to funkcja printf). Dodatkowo, nawiasy <> informują o tym, że pliku stdio.h należy szukać w kartotece innej aniżeli kartoteka bieżąca (np. w kartotece zawierającej deklaracje funkcji standardowych). Nazwa pliku umieszczona w cudzysłowie "" informuje, że pliku należy szukać w kartotece bieżącej (na przykład gdy jest to plik własny programisty).

   Takie pliki, dołączane do programu na jego początku, za pomocą dyrektywy `#include`, nazywane są *plikami nagłówkowymi* (*header files*) i deklarują przydatne nam funkcje.

5. Instrukcja `return 0` kończy działanie funkcji i służy do przekazywania wyznaczonej wartości funkcji do programu. W tym wypadku jednak sam program nie korzysta z wynikowej wartości funkcji, a wartość 0 przekazywana jest przez funkcję main do systemu i informuje go o pomyślnym zakończeniu jej działania, a więc i działania całego programu. Możliwe są również inne postacie tej instrukcji: `return(0)` lub tylko `return`.

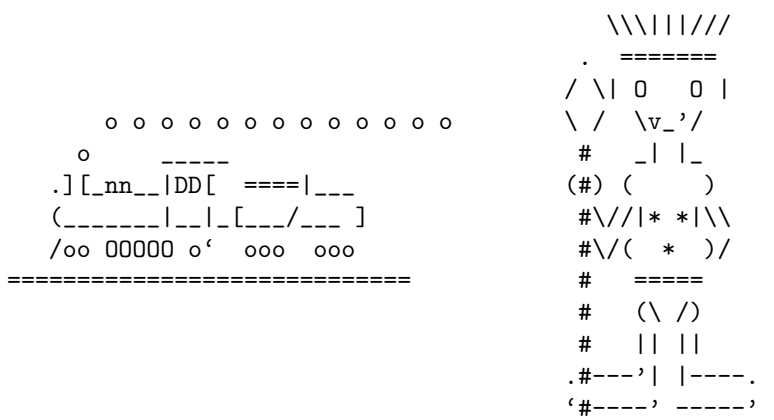
6. Ten program *nie* jest porządnie napisany.

   Zamiast instrukcji return można przekazać systemowi informację o zakończeniu działania funkcji main za pomocą standardowej funkcji exit() wywołaniem `exit(0)`. Funkcja ta jest zadeklarowana w pliku stdlib.h i trzeba poinformować o tym kompilator dyrektywą `#include <stdlib.h>`.

1.2 Zadania

1. Modyfikując powyższy program narysować na ekranie swoją wizytówkę.

2. Wykorzystując dostępne znaki narysować na ekranie w trybie tekstowym jakiś obrazek, np. lokomotywę, wojownika, choinkę, ...



2 Zmienne, typy i deklaracje

Jak wynika z samej nazwy – *zmienna* (*variable*) może reprezentować różne wartości: $x = 5$, $x = 3.14159265368$.

DEF. Zmienna jest pojęciem fizycznym. Zajmuje ona, albo jest utożsamiana z pewnym obszarem w pamięci operacyjnej komputera, a rozmiar tego obszaru i sposób interpretacji jego zawartości są określone przez *typ* zmiennej, definiujący rodzaj wartości jakie przyjmuje zmienna (np. liczby całkowite, rzeczywiste, znaki, ...).
Każda zmienna jest identyfikowana przez przypisaną jej *nazwę*. □

DEF. Nazwa jest ciągiem liter lub cyfr zaczynającym się od litery. Na prawach litery traktowany jest również znak podkreślenia _. Kompilatory języka C rozróżniają litery duże i małe. □

Warto wiedzieć, że nazwy zaczynające się od dwu znaków podkreślenia lub znaku podkreślenia i dużej litery są zarezerwowane do użycia w bibliotece języka C i nie należy ich używać we własnych programach. Niektóre (stare) kompilatory biorą pod uwagę jedynie 32 pierwsze znaki nazwy.

Wszystkie zmienne (tzn. ich nazwy) muszą być *zadeklarowane* przed użyciem, a więc najwygodniej jest zrobić to na początku funkcji czy programu.

DEF. Język C zawiera pewne zarezerwowane nazwy, których znaczenie i efekt użycia zostały ściśle określone. Nazywamy je **słowami kluczowymi** i piszemy wyłącznie małymi literami. □

Dla wygody teraz podajemy listę słów kluczowych języka C. Korzystać z nich będziemy w miarę potrzeby:
auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while.
Słów kluczowych nie można używać jako nazw zmiennych. Ponadto, nie należy jako nazw zmiennych używać słów kluczowych języka C++:
asm, bool, catch, class, `const_cast`, delete, `dynamic_cast`, explicit, inline, false, friend, mutable, namespace, new, operator, private, protected, public,
`reinterpret_cast`, `static_cast`, template, this, throw, true, try, typeid, typename, using, virtual, `wchar_t`,

2.1 Typy całkowite

Słowa kluczowe określające typy całkowite:

char, int, long, short, signed, unsigned.

Z ich pomocą możemy zadeklarować *zmienne typu całkowitego*. Rozmiar, a więc i zakres zmiennych zależy od komputera i kompilatora języka. U nas:

Deklaracja	Rozmiar w bajtach	Min	Max
signed char znak;	1	−128	127
unsigned char c;	1	0	255
short int krotka;	2	−32 768	32 767
int m;	4	−2 147 483 648	2 147 483 647
long int dluga;	4	−2 147 483 648	2 147 483 647
unsigned short int krotka;	2	0	65 535
unsigned long int dluga;	4	0	4 294 967 295
long long `b_dluga`;	8	-2^{63}	$2^{63}-1$
unsigned long long `b_dluga`;	8	0	2^{64}

Uwagi:

1. W języku C jest jeszcze typ char, identyczny z jednym z dwóch wymienionych typów: signed char albo unsigned char i najczęściej identyfikowany z tym pierwszym. Jeżeli więc to może być istotne, to lepiej deklarować zmienne używając jednoznacznych nazw.
2. Rozmiar, a więc i zakres zmiennych typu int zależy od kompilatora. Mogą to być 2, 4 albo 8 bajtów (u nas 4).
3. Podobnie jest z typem long. Wiadomo, że długość jego nie powinna być mniejsza od 32 bitów.

2.2 Typy zmiennopozycyjne

Zmienne typów zmiennopozycyjnych (albo zmiennoprzecinkowych) służą do przechowywania wartości niecałkowitych, w języku C definiujemy je za pomocą słów kluczowych:

float, double i long.

W poniższej tabeli podajemy zakresy dodatnich wartości takich zmiennych:

Deklaracja	Rozmiar w bajtach	Min	Max
float x;	4	3.4E−38	3.4E38
double x;	8	1.7E−308	1.7E308
long double x;	10	3.4E−4932	1.1E4932

Na co dzień wymienione typy zmiennoprzecinkowe nazywa się typami odpowiednio pojedynczej, podwójnej oraz wysokiej precyzji.

2.3 Deklaracje zmiennych

DEF. Deklaracja zmiennej określonego typu, czyli polecenie zarezerwowania fragmentu pamięci na przechowywanie wartości tej zmiennej, w najprostszej postaci ma postać:

$$< typ > \quad < nazwa\_zmiennej >;$$

Kolejne deklaracje zakończone są średnikiem. □

Możemy zadeklarować kilka zmiennych tego samego typu rozdzielając je przecinkami:

```
int a1,b,maly;
char c1,c2,cc;
```

W momencie deklarowania można dokonać *inicjacji zmiennej*, tzn. nadać jej wartość początkową:

```
double szerokosc = 1.5E12;
int b = −15835;
char cc = 'A';
```

2.4 Stałe

Stałe, podobnie jak zmienne mogą być tylko stałymi określonych typów:

typ	składnia stałej	przykład
char	ujęta w apostrofy	'a'
ciąg znaków	ujęta w cudzysłowy	"abc"
int	nie może mieć 0 na początku	123
liczba ósemkowa	na początku musi mieć 0	0123
liczba szesnastkowa	na początku musi mieć 0x	0x1af
long int	kończy się literą l lub L	123L
double	zawiera kropkę lub jest	12.123
	pisana z literą e lub E	1.1e-123

Czasem, gdy ta sama stała, np. będąca wartością jakiegoś parametru, występuje w kilku miejscach programu (funkcji), każda zmiana jej wartości wymaga wprowadzenia takich samych poprawek we wszystkich miejscach występowania tej stałej. Stwarza to niebezpieczeństwo popełnienia błędu i jest mało wygodne. Dlatego warto zadeklarować nową zmienną oraz nadać jej wymaganą wartość na początku programu i stosować dalej w treści. Nawet bezpieczniej – można takiej zmiennej *przypisać atrybut stałości* (zob. [5, str. 111]). Wartość takiej zmiennej nie będzie mogła ulec zmianie w trakcie działania programu:

```
const double szerokosc = 1.5E12;
const b = −15835;
const char cc = 'A';
```

2.4.1 Stałe symboliczne

Stałą symboliczną definiuje się za pomocą dyrektywy `#define` preprocesora. Przyjęło się pisać nazwy stałych symbolicznych dużymi literami. Stałych symbolicznych można używać przy definiowaniu tablicy:

```
#define N 8
int tab[N];
```

Do dyspozycji użytkownika w języku C został przewidziany zestaw standardowych stałych:

2.4.2 Standardowe stałe matematyczne

Większość stałych matematycznych z podanych w [5, str. 441] jest również dostępna pod Linuxem. Trzeba tylko dołączyć do programu plik nagłówkowy math.h . Nazwy standardowych stałych są pisane wyłącznie dużymi literami. Między innymi są to:

nazwa stałej	znaczenie	nazwa stałej	znaczenie
M_E	e	M_PI	π
M_LOG2E	$\log_2 e$	M_PI_2	$\pi/2$
M_LOG10E	$\log_{10} e$	M_PI_4	$\pi/4$
M_LN2	$\ln 2$	M_2_SQRTPI	$2/\sqrt{\pi}$
M_LN10	$\ln 10$	M_SQRT2	$\sqrt{2}$

3 Obsługa standardowego wejścia i wyjścia

W języku C do przechowywania i przesyłania informacji służą **pliki** (*file*), inaczej – *ciągi bajtów*. Mogą one być przechowywane na dyskach, wprowadzane z klawiatury do pamięci, przesyłane przez sieć, wyświetlane na monitorze, ...

Takie przesyłane pliki zwane są też *strumieniami znaków*. C zawiera obszerny zestaw funkcji do obsługi plików.

Język C rozróżnia dwa rodzaje plików — pliki binarne i pliki tekstowe. Pliki binarne traktowane są jako ciągi bajtów, a interpretacja znaczenia zawartości pliku jest zadaniem programu. Pliki tekstowe składają się z wierszy (linii). Zewnętrzna postać pliku tekstowego zależy od systemu operacyjnego — jest wiele różnych sposobów zaznaczania granic wierszy. Przy czytaniu pliku tekstowego funkcje biblioteczne C wczytują zawartość wiersza i kończą go znakiem nowej linii `\n`. Przy zapisie dokonuje się konwersji w przeciwną stronę. W systemie Linux, tak jak i w innych wersjach UNIX-a, w rzeczywistości nie trzeba dokonywać żadnych konwersji i nie ma praktycznych różnic między plikami binarnymi i tekstowymi. W systemie DOS wiersze kończą sie parą znaków `\r` i `\n`. Na komputerach Macintosh wiersze kończą się znakiem `\r`. Niekiedy używa się jednego znaku do zaznaczania początku i innego do zaznaczania końca wiersza. Można też umieszczać przed wierszem licznik podający ilość znaków w wierszu. Komputery mainframe często ograniczały długość wiersza do 80 znaków a krótsze wiersze uzupełniały spacjami. Przy wczytywaniu danych z klawiatury końcem wiersza jest <*Enter*>.

Pliki mają zwykle skończoną długość, tak że po przeczytaniu wszystkich danych dochodzimy do końca pliku. Jeśli wczytujemy plik znak po znaku, to otrzymamy wtedy specjalną wartość `EOF`. Należy tu podkreślić że `EOF` nie jest znakiem, jest liczbą całkowitą (zwykle -1), różną od wszystkich wartości odpowiadających wczytywanym znakom. W przypadku wprowadzania pliku z klawiatury koniec pliku sygnalizujemy znakiem ^d (*Ctrl–d*) w nowym wierszu.

3.1 Drukowanie na ekranie – funkcja printf

W rozdziale 1.1 zamieściliśmy program drukujący na ekranie łańcuchy za pomocą standardowej funkcji printf. W ogólnej swojej postaci funkcja printf służy do drukowania wartości wyrażeń. Jej pierwszym argumentem jest **łańcuch formatujący**, a następnymi wyrażenia, których wartości zamierzamy drukować.

Otóż, jeśli chcemy wydrukować na ekranie zmienną jakiegokolwiek typu (wyrażenie), musimy najpierw określić **specyfikator formatu** odpowiadający tej zmiennej (wyrażeniu) i umieścić go na odpowiednim miejscu w *łańcuchu formatującym*.

Przykład. Po uruchomieniu programu

```
#include<stdio.h>
#include<math.h>
main()
{
  double x;
  int k;
  k=5;  x=k+M_PI;
  printf("k = %d, ale x = %10.8f jest większe.\n",k,x);
}
```

otrzymujemy na ekranie tekst:

```
k = 5, ale x = 8.14159265 jest większe.
```

a kursor znalazł się w następnym wierszu.
Łańcuch formatujący w instrukcji printf przewidywał wydrukowanie dwóch liczb całkowitej i rzeczywistej, wplecionych w podany tekst. □

Specyfikator formatu jest sekwencją znaków składającą się co najmniej ze znaku % i **znaku przekształcenia** polecającego wyprowadzić odpowiednio:

znak przekształcenia	printf wyprowadza
d,**i**	liczba typu int
u	l. całkowita typu unsigned
o	l. całk. bez znaku w postaci ósemkowej
x,**X**	l. całk. bez znaku w postaci szesnastkowej
f	l. zmiennopozycyjna
e,**E**	l. typu double w postaci wykładniczej
g,**G**	krótsza ze specyfikacji: **%f**, **%e**, **%E**
c	argument typu char
s	łańcuch
p	argument typu wskaźnik
%	znak %

Mały albo duży znak przekształcenia wybiera odpowiednio małe albo duże litery w formacie liczby. Pomiędzy tymi dwoma znakami mogą się jeszcze znaleźć kolejno:

element	znaczenie
–	wyrównanie do lewej strony pola
<liczba całkowita>	minimalna szerokość pola
.<liczba całkowita>	ilość cyfr dziesiętnych po kropce dla liczby zmiennopozycyjnej
litera h albo l	dla typów całkowitych short albo long
litera L	dla zmiennopozycyjnego typu long double

W łańcuchu formatującym mogą jeszcze wystąpić *dwuznakowe sekwencje sterujące*. Składają się one z *backslash*'a i znaku dodatkowego:

sekwencja	znaczenie
\ a	sygnał dźwiękowy
\ n	przejście do nowej linii
\ r	powrót karetki
\ t	tabulacja
\ b	cofnięcie o jeden znak
\ “	znak “
\ ’	znak ’
\\	znak \

Zadanie. Napisać program ilustrujący rezultaty drukowania

1. liczb całkowitych formatami: %o, %x, %5d.

2. liczb zmiennoprzecinkowych za pomocą formatów: %f, %10f, %.2f, %6.2f, %e, %10e, %.2e, %6.2e, %0f, %0e.
   Uwaga: Specyfikacja %0f zapobiega wypisywaniu kropki dziesiętnej i zerowej części ułamkowej.

3.2 Wprowadzanie danych znakowych

4 Wyrażenie arytmetyczne

4.1 Instrukcja przypisania

Jest to podstawowa instrukcja, która pozwala na nadawanie zmiennym konkretnych wartości.
DEF. Instrukcja przypisania ma postać:

$$<nazwa\_zmiennej> \quad = \quad <wyrażenie>;$$ □

Na razie mamy na myśli intuicyjnie zrozumiałe *wyrażenie arytmetyczne*, np:

$$x = 3.14; \qquad alfa = x + 5;$$

4.2 Operatory arytmetyczne, kolejność działań

Do dyspozycji mamy przede wszystkim **operatory arytmetyczne** i nawiasy okrągłe ().

DEF. Operatory typu dodawania: $+$, $-$ należą do operatorów jedno– lub dwuargumentowych. □

DEF. Operatory typu mnożenia: $*$, $/$, $\%$ są dwuargumentowymi.
Operator $\%$ określony jest wyłącznie dla argumentów całkowitych i oznacza *dzielenie modulo* czyli wyznaczenie reszty z dzielenia liczb całkowitych ze znakiem dzielnej.

Gdy obydwa argumenty operatora dzielenia / są typu całkowitego – rezultatem jest liczba typu całkowitego, równa wynikowi dzielenia całkowitego. Jest to maksymalna liczba całkowita niewiększa od ilorazu modułów argumentów ze znakiem ilorazu argumentów. Jeżeli co najmniej jeden z argumentów jest typu rzeczywistego, to wynik jest również typu rzeczywistego (*liczbą zmiennoprzecinkową*). □

W języku C obowiązują następujące zasady obliczania wartości wyrażeń arytmetycznych:

1. Wyrażenia arytmetyczne piszemy na jednym poziomie.
2. Operatory typu mnożenia mają wyższy priorytet od operatorów typu dodawania.
3. Operatory dwuargumentowe o tym samym priorytecie wykonuje się zaczynając od lewej strony.
4. Operatory jednoargumentowe o tym samym priorytecie wykonujemy zaczynając od strony prawej.
5. O ile nie spowoduje to wprowadzenia niedozwolonej interpretacji (zob. dalej np. operatory inkrementacji) operatory jednoargumentowe mogą sąsiadować ze sobą lub z wcześniejszymi operatorami dwuargumentowymi.
6. Kolejność wykonywania operatorów arytmetycznych możemy zmieniać wykorzystując nawiasy okrągłe (), ale nie mamy już wpływu na kolejność wyznaczania wartosci argumentów poszczególnych operatorów występujących w wyrażeniu. Zatem, jeżeli kolejność ta ma znaczenie, to należy obliczać wartości pośrednie otrzymując w rezultacie jednoznacznie określoną wartość dłuższego wyrażenia.

Przykłady dla int y;

1. y = 71/ − + − +−3; ⟶ y= −23 ,
2. y = 71% − + − +−3; ⟶ y=2 ,
3. y = −15/+2/ + −2; ⟶ y=3 .

W trakcie obliczania wartości wyrażenia arytmetycznego dokonywane są *niejawne, automatyczne przekształcenia arytmetyczne argumentów operatorów arytmetycznych* (zob. [2, str. 55]). Ogólnie – *jeśli argumenty operatora dwuargumentowego są różnego typu, to przed wykonaniem operacji typ mniejszy jest zamieniany na typ większy i typem wyniku jest typ większy.* Cała arytmetyka zmiennopozycyjna języka C jest oparta na podwójnej precyzji double. Dokładniej:

1. Typy char i short są zamieniane na int, a float na double.
2. Jeśli którykolwiek z argumentów jest typu double, to drugi też zostaje zamieniony na double i wynik również jest typu double.
3. W przeciwnym przypadku, jeśli typem jednego z argumentów jest long, to drugi argument też zostaje zamieniony na long i wynik również jest typu long.
4. W przeciwnym przypadku, jeśli typem jednego z argumentów jest unsigned, to drugi argument też zostaje zamieniony na unsigned i wynik jest typu unsigned.
5. W przeciwnym przypadku obydwa argumenty muszą być typu int, a więc i wynik również jest typu int.

Przykład. Pisząc:

```
int i,j;  float a;
   i=23;   j=3;   a=i/j;
```

otrzymamy wynik `a=7`, a więc wynik z dzielenia całkowitego!

Przy programowaniu wyrażeń arytmetycznych trzeba więc czasem np. przepisać wartość do zmiennej zmiennoprzecinkowej, albo dopisać kropkę z zerem do stałej całkowitej. Można też wykorzystać *operatory rzutowania*:

DEF. Operatory rzutowania służą do wykonywania poprawnej zmiany aktualnego typu wyrażenia arytmetycznego na żądany i przyjmują formę:

$$(<nazwa\_typu>) \quad <wyrażenie\_arytmetyczne>;$$

Rzutowanie działa tak, jak gdyby wartość wyrażenia arytmetycznego przypisano zmiennej wskazanego typu, a następnie tę zmienną użyto dalej. □

A więc jeśli chcemy dzielenia zmiennoprzecinkowego, to zamiast `a=i/j;` piszemy: `a=((double)i)/j;` lub `a=i/(double)j;` otrzymując wartość `a=7.6666667`.

4.3 Standardowe funkcje matematyczne

Do dyspozycji mamy zestaw standardowych funkcji matematycznych. Aby korzystać z wielu z nich trzeba dołączyć do programu plik math.h. Podstawowym typem zmiennopozycyjnym języka C jest double, najczęściej też argumenty i wartości funkcji arytmetycznych zostały przewidziane w tym właśnie typie. Do dyspozycji użytkownika pozostają również standardowe funkcje matematyczne o argumentach i wartościach typu long double. Nazwy tych funkcji są dłuższe o przyrostek l (np. cosl(x).

1. Funkcje trygonometryczne – argumenty w radianach:
   cos(x), sin(x), tan(x).
2. Funkcje cyklometryczne – wartość w radianach:
   acos(x), asin(x), atan(x), atan2(x,y) – *arcus tangens*(x/y).
3. Funkcje hiperboliczne:
   cosh(x), sinh(x), tanh(x).
4. Funkcje wykładnicza i logarytmiczna:
   exp(x),
   log(x) – logarytm naturalny,
   log10(x) – logarytm dziesiętny,
   pow(x,y) – potęga: x do y.
5. Funkcje pozostałe:
   sqrt(x),
   abs() – wartość bezwzględna: argument i wartość typu int
   fabs() – wartość bezwzględna: argument i wartość typu double
   ceil(x) – najmniejsza l. całkowita niemniejsza od x,
   floor(x) – największa l. całkowita niewiększa od x,
   fmod(x,y) – x *modulo* y.

4.4 Funkcje

Większy program, ku wygodzie użytkownika może być podzielony na mniejsze fragmenty, popularnie zwane podprogramami, procedurami, funkcjami, ... W języku C nie ma takiego rozróżnienia – są tylko *funkcje*.

Zatem funkcje mogą zwracać do programu obliczoną wartość i wtedy trzeba mówić o typie tej wartości, a mogą też żadnej wartości nie zwracać. W ostatnim przypadku mówimy o *pustym typie wartości* void.

Jeżeli funkcja zwraca wynik innego typu aniżeli int, to musi być zadeklarowaną przed pierwszym wywołaniem.

Definicja funkcji ma następującą postac:

<*typ_wartości_funkcji*> <*nazwa_funkcji*> (<*lista_argumentów_funkcji*>)
{ <*instrukcje_czyli_treść_funkcji*> }

gdzie <*lista_argumentów_funkcji*> jest ciągiem oddzielonych przecinkami kolejnych *deklaracji argumentów funkcji*. □

Przykład. Definicja delty przy rozwiązywaniu równania kwadratowego może mieć postać:

```
double delta(double a, double b, double c)
  { return b*b-4*a*c; }
```

□

Obowiązują następujące zasady:

1. Wszystkie funkcje muszą występować na tym samym poziomie leksykalnym – nie jest możliwe zadeklarowanie jednej funkcji wewnątrz drugiej.
2. Argumenty funkcji są *przekazywane przez wartość*.
3. Działanie funkcji przerywa i kończy instrukcja return lub dojście do końca tekstu funkcji.
   - Jeżeli funkcja przekazuje na zewnątrz obliczoną wartość, to powinna być wykorzystana instrukcja:

     return <*wyrażenie*>;

   - Jeżeli funkcja nie przekazuje żadnej wartości na zewnątrz, to powinna być to po prostu instrukcja:

     return;

     lub dojście do końca treści funkcji.
4. Przekazywana przez funkcję wartość może byc ignorowana w programie.

5 Podstawy arytmetyki komputerowej

Jak dotąd najoptymalniejszym (najtańszym i najmniej zawodnym) okazał się system dwójkowy elektronicznej reprezentacji liczb. Najmniejszą jednostką pamięci komputera jest bit mogący pamiętać jedynie dwa różne stany, a więc właśnie cyfry dwójkowe: 0 i 1. Bity zostały pogrupowane w wygodniejsze większe, ośmiobitowe jednostki zwane bajtami, a kilka takich bajtów, jako ustalonych długości dłuższe słowa maszynowe są przydzielane zmiennym dla przechowywania przyjmowanych przez nie wartości liczbowych.

5.1 Arytmetyka liczb całkowitych

Liczby typów całkowitych są pamiętane w konwencji **stałopozycyjnej** (albo stałoprzecinkowej), co z grubsza możemy przedstawić tak:

1. W typach bez znaku (unsigned) wszystkie bity są przeznaczone na pamiętanie cyfr i na przykład w jednym bajcie (8 cyfr dwójkowych) możemy pamiętać liczby od $00000000_{(2)} = 0_{(10)}$ do $11111111_{(2)} = 255_{(10)}$.

2. W typach ze znakiem trzeba przeznaczyć jeden bit na pamiętanie znaku, zatem w jednym bajcie (stosowany jest *kod uzupełnieniowy*) możemy pamiętać liczby od $10000000_{(2)} = -128_{(10)}$ do $01111111_{(2)} = +127_{(10)}$.

3. Dodawanie, odejmowanie i mnożenie przebiegają niezależnie od znaku liczby! W każdym przypadku jest to działanie modulo 2^n odpowiednio dla $n = 8, 16, 32$ w zależności od typu.

4. Liczby całkowite są pamiętane dokładnie. Zadaniem typów całkowitych w obliczeniach komputerowych jest przede wszystkim zapewnienie obsługi indeksów tablic, adresów (*wskaźników*), pętli. Operacje na takich wielkościach powinny być dokładne i maksymalnie szybkie. Dokładne, jak widać są, a szybkość zapewniono nie troszcząc się o ewentualne przekroczenie dopuszczalnego zakresu wartości poszczególnych zmiennych całkowitych.

5.2 Arytmetyka liczb rzeczywistych

5.2.1 Reprezentacja liczb rzeczywistych w komputerze

Liczby rzeczywiste są pamiętane w konwencji **zmiennopozycyjnej** (albo zmiennoprzecinkowej, ang. *floating point*): słowo maszynowe, długości kilku bajtów jest podzielone na dwie główne części. W jednej pamiętana jest ułamkowa **mantysa liczby**: m ze znakiem, w drugiej całkowita **cecha liczby**: d.

Liczba ma postać:

$$x = m * 2^d\,,$$

gdzie:

$$\begin{array}{ll} \frac{1}{2} \le |m| < 1 & \text{dla} \quad x \neq 0\,, \\ m = 0 \;\text{ i }\; d = 0 & \text{dla} \quad x = 0\,. \end{array}$$

W układzie dziesiętnym warunki byłyby następujące:

$$\begin{array}{c} x = m * 10^d\,, \\ 0.1 \le |m| < 1 \quad \text{dla} \quad x \neq 0\,. \end{array}$$

Liczby różne od zera należy najpierw sprowadzić do wymaganej postaci. Na przykład początkowe zera liczby nie mają znaczenia. To nie są *cyfry znaczące* (istotne) tej liczby.

Dla prostoty przyjmijmy model arytmetyki zmiennopozycyjnej dziesiętnej z czterema cyframi mantysy i dwiema cyframi cechy. Dla wygody będziemy zaznaczać równą zeru część całkowitą mantysy i 10 – podstawę systemu liczbowego:

$$\pm 0\,.\,\square\square\square\square_{\mathbf{10}} \pm \square\square$$

Nasze słowo maszynowe zawiera więc osiem pól:
- pięć na przechowywanie mantysy m (znak i 4 cyfry znaczące),

- trzy na przechowywanie cechy d (znak i 2 cyfry).

Przyjrzyjmy się teraz podstawowym operacjom arytmetycznym. Wykonajmy instrukcję

$$x = 5/3;$$

Wstawiana liczba ma nieskończone rozwinięcie dziesiętne $1.666666\ldots$. W mantysie wystarcza miejsca jedynie na 4 cyfry dziesiętne, a więc w najlepszym razie zapamiętane zostanie

$$\widetilde{x} = 0.1667_{10}01\,.$$

DEF. Wielkość $\delta = |x - \widetilde{x}|$ nazywamy **błędem bezwzględnym reprezentacji** $\widetilde{x}$ liczby zmiennopozycyjnej x. □

Prawdziwe są twierdzenia:

TW. 1 *Jeżeli algorytm zapamiętujący będzie poprawnie zaokrąglać umieszczane w pamięci liczby do liczb z mantysą o wymaganej długości, to błąd bezwzględny reprezentacji* $\delta = |x - \widetilde{x}|$ *nie przekroczy* 5 *na pierwszym opuszczonym miejscu mantysy, i dla różnej od zera*

$$\widetilde{x} = m * 10^d$$

prawdziwą będzie nierówność:

$$\delta \leq 5 * 10^{d-5}\,.$$ □

TW. 2 *Jeżeli algorytm zapamiętujący będzie jedynie obcinać zapamiętywaną mantysę do wymaganej długości, to możemy tylko stwierdzić, że dla różnej od zera liczby*

$$x = m * 10^d$$

jej bezwzględny błąd reprezentacji δ *spełnia nierówność*

$$\delta < 10^{d-4}\,.$$ □

Bardzo wygodnym w użyciu jest *błąd względny* reprezentacji zapamiętanej i różnej od zera liczby:

DEF. Błędem względnym reprezentacji $\widetilde{x} \neq 0$ nazywamy

$$\epsilon = \frac{\delta}{|\widetilde{x}|}$$

gdzie δ jest błędem względnym reprezentacji. □

Oszacujmy błąd względny reprezentacji. W wypadku zaokrąglania otrzymujemy:

$$\epsilon = \frac{\delta}{|m|10^d} \leq \frac{5 * 10^{d-5}}{0.1 * 10^d} = 5_{10} - 4\,,$$

a w wypadku obcinania:

$$\epsilon < \frac{10^{d-4}}{0.1 * 10^d} = {}_{10} - 3\,.$$

5.2.2 Dodawanie i odejmowanie liczb zmiennopozycyjnych

1. Dodajmy dwie liczby pamiętane już w komputerze, a więc traktowane jako reprezentowane dokładnie:

   $$0.1234_{10}3 + 0.1234_{10}5$$

   Wyobraźmy sobie, że arytmometr procesora wykorzystuje dłuższą mantysę i wykonuje te obliczenia poprawnie. Przed dodawaniem obydwu liczb trzeba wyłączyć czynnik z większą cechą:

   $$\begin{array}{rll} & 0.1234 & *\,10^5 \\ + & 0.001234 & *\,10^5 \\ \hline & 0.124634 & *\,10^5 \end{array}$$

   a teraz należy zaokrąglić wynik do czterech cyfr mantysy. Otrzymujemy

   $$0.1246_{10}5\,.$$

   Błąd względny sumy nie przekracza $5_{10} - 4\,.$

2. Przy większej różnicy dodawanych wielkości jeden ze składników może zostać nie zauważony:

   $$0.1234_{10}0 + 0.1234_{10}5 = 0.1234_{10}5$$

   ponieważ:

   $$\begin{array}{rll} & 0.1234 & *\,10^5 \\ + & 0.000001234 & *\,10^5 \\ \hline & 0.123401234 & *\,10^5 \end{array}$$

   i po zaokrągleniu otrzymujemy:

   $$0.1234_{10}5\,.$$

3. O tym, iż *dodawanie w komputerze nie jest działaniem łącznym*(!) najłatwiej jest się przekonać przy założeniu, że procesor modelowego komputera dysponuje również tylko 4–cyfrową mantysą:

   $$(\ 0.1234_{10}0\ +\ 0.5000_{10} - 4\ )\ +\ 0.5000_{10} - 4\ =\ 0.1234_{10}0\,,$$

   a

   $$0.1234_{10}0\ +\ (\ 0.5000_{10} - 4\ +\ 0.5000_{10} - 4\ )\ =\ 0.1235_{10}0\,.$$

5.2.3 Zadania

1. Ile wynosi najmniejsza zmiennopozycyjna liczba dodatnia, dokładnie reprezentowana w modelowym komputerze? A największa dodatnia?
2. Zaznaczyć na osi liczbowej R liczby zmiennopozycyjne, reprezentowane dokładnie w modelowym komputerze. Przyjrzec się rozkładowi tych liczb.
3. Oszacować błędy reprezentacji zmiennopozycyjnej (bezwzględny i względny) dla arytmetyki dwójkowej z t–cyfrową mantysą.
4. Na przykład metodą prób i błędów dobrać liczby a, b, c tak, aby na komputerze w pracowni różnica między wartościami $x = (a + b) + c$; i $y = a + (b + c)$; była jak największa.
5. Zasymulować przykładowy komputer używając arytmetyki na liczbach całkowitych.

5.2.4 Mnożenie i dzielenie liczb zmiennopozycyjnych

Mantysa iloczynu lub ilorazu dwóch liczb zmiennoprzecinkowych z czterocyfrowymi mantysami jest najczęściej znacznie dłuższa i wymaga zaokrąglenia albo obcięcia do wymaganej długości:

$$(0.1234_{10}3 \ * \ 0.1234_{10}2) \ = \ 0.1522756_{10}5 .$$

Możemy na przykład otrzymać $0.1523_{10}5$, a oszacowanie błędu względnego iloczynu lub ilorazu w naszym modelowym komputerze spełnia nierówność

$$\epsilon \leq 5_{10} - 4 .$$

5.2.5 Utrata dokładnych cyfr znaczących

Rozpatrzmy odejmowanie dwóch bliskich sobie liczb w naszym modelu czterocyfrowej dziesiętnej mantysy. Policzmy $\frac{8}{9} - \frac{6}{7}$ instrukcjami:

$$x = 6/7;$$

$$y = 8/9;$$

$$z = y - x;$$

Realizacja obliczeń:

$$\begin{array}{rll} & 0.8889 & * 10^0 \\ - & 0.8571 & * 10^0 \\ \hline & 0.0318 & * 10^0 \end{array}$$

Otrzymaliśmy $0.3180_{10} - 1$, podczas gdy reprezentacja komputerowa wyniku jest równa $0.3175_{10} - 1$. Błąd względny obliczonej różnicy jest czterokrotnie większy od względnego błędu reprezentacji. Takie przypadki są groźne.

Zadanie. ([1, str. 67])

Niech $x = 0.3721448693$ i $y = 0.3720214371$. Jaki jest względny błąd różnicy $x - y$ obliczonej w komputerze z 5–cyfrową dziesiętną mantysą? □

Typowym, omawianym na *metodach numerycznych* przykładem jest rozwiązywanie równania kwadratowego:

5.2.6 Przykład z algorytmów – równanie kwadratowe

Rozwiązujemy równanie

$$ax^2 + bx + c = 0$$

dla $a > 0$. W szkole nauczyliśmy się robić to tak:
Wyznaczamy deltę (załóżmy, że jest ona dodatnia):

$$\Delta = b^2 - 4ac ,$$

a następnie liczymy:

$$x_1 = \frac{-b+\sqrt{\Delta}}{2a}\,, \quad x_2 = \frac{-b-\sqrt{\Delta}}{2a}\,.$$

Raz dodajemy, a raz odejmujemy w liczniku pierwiastek kwadratowy z tego samego wyrażenia. Jeżeli $4ac$ jest "niewielkie", to w jednym z liczników występuje odejmowanie bliskich sobie wartości, a więc następuje utrata dokładnych cyfr znaczących wyniku. **Taki algorytm jest numerycznie złym algorytmem.** Lepiej więc najpierw wykorzystać ten ze wzorów, w którym obydwa składniki licznika mają taki sam znak, a potem, do wyznaczenia drugiego z pierwiastków skorzystać ze wzoru Viet'a: $c = ax_1x_2$.

Takie postępowanie pozwoli uniknąć odejmowania dwóch bliskich sobie wartości, a drobne przekształcenia zmniejszą liczbę wykonywanych działań. Przekształćmy bowiem wyrażenie:

$$x_1 \;=\; \frac{-b+\sqrt{b^2-4ac}}{2a} \;=\; \left(\frac{-b}{2a}\right) + \sqrt{\left(\frac{-b}{2a}\right)^2 - \left(\frac{c}{a}\right)}\,.$$

Jeżeli $b > 0\,,$ to w tym wyrażeniu następuje redukcja cyfr znaczących. Przekształcajmy więc dalej:

$$\begin{aligned} x_1 \;&=\; \frac{\left(\left(\frac{-b}{2a}\right) + \sqrt{\left(\frac{-b}{2a}\right)^2 - \left(\frac{c}{a}\right)}\right)\left(\left(\frac{-b}{2a}\right) - \sqrt{\left(\frac{-b}{2a}\right)^2 - \left(\frac{c}{a}\right)}\right)}{\left(\left(\frac{-b}{2a}\right) - \sqrt{\left(\frac{-b}{2a}\right)^2 - \left(\frac{c}{a}\right)}\right)} \;= \\ &=\; \frac{\left(\frac{c}{a}\right)}{\left(\frac{-b}{2a}\right) - \sqrt{\left(\frac{-b}{2a}\right)^2 - \left(\frac{c}{a}\right)}}\,, \end{aligned}$$

a w tym wyrażeniu obydwa składniki są już tego samego znaku i redukcji dokładnych cyfr znaczących nie będzie.

Podobnie postępujemy z wyrażeniem na x_2:

$$x_2 \;=\; \frac{-b-\sqrt{b^2-4ac}}{2a} \;=\; \left(\frac{-b}{2a}\right) - \sqrt{\left(\frac{-b}{2a}\right)^2 - \left(\frac{c}{a}\right)}\,.$$

Jeżeli grozi redukcja cyfr, to przekształćmy proponowane wyrażenie:

$$\begin{aligned} x_2 \;&=\; \frac{\left(\left(\frac{-b}{2a}\right) - \sqrt{\left(\frac{-b}{2a}\right)^2 - \left(\frac{c}{a}\right)}\right)\left(\left(\frac{-b}{2a}\right) + \sqrt{\left(\frac{-b}{2a}\right)^2 - \left(\frac{c}{a}\right)}\right)}{\left(\left(\frac{-b}{2a}\right) + \sqrt{\left(\frac{-b}{2a}\right)^2 - \left(\frac{c}{a}\right)}\right)} \;= \\ &=\; \frac{\left(\frac{c}{a}\right)}{\left(\frac{-b}{2a}\right) + \sqrt{\left(\frac{-b}{2a}\right)^2 - \left(\frac{c}{a}\right)}}\,. \end{aligned}$$

Jak widać, pierwszy z pierwiastków wyznaczamy ze wzoru dokładniejszego numerycznie, a drugi ze wzoru Viet'a i obydwa składniki w mianowniku mają ten sam znak.

Algorytm numeryczny rozwiązywania równania kwadratowego powinien wyglądać następująco:

Dane:	$a,\ b,\ c$ – rzeczywiste $(a > 0,\ \Delta > 0)$.
Szukane:	$x_1\,,\ x_2$ – wzory znane.
Wielkości pomocnicze:	$b1,\ c1,\ d1$ – rzeczywiste.
Obliczamy kolejno:	$b1 = -b/(2a)\,,$
	$c1 = c/a\,,$
	$d1 = \sqrt{b1^2 - c1}\,.$
Jeśli	$b1 > 0$
to:	$x_1 = b1 + d1$ i $x_2 = c1/x_1\,,$
w przeciwnym razie:	$x_2 = b1 - d1$ i $x_1 = c1/x_2\,.$

Zadanie

Napisać program rozwiązujący równanie kwadratowe

$$ax^2 + bx + c = 0$$

dla danych wartości współczynników $a,\ b$ i c.

5.2.7 Podsumowanie arytmetyki zmiennoprzecinkowej

Jak widzieliśmy, działania zmiennoprzecinkowe w komputerze są operacjami skomplikowanymi. Zwykłe dodawanie wymaga wstępnego dopasowania cech obu liczb, wykonania operacji na mantysach oraz ewentualnego doprowadzenia otrzymanego wyniku do wymaganej postaci. We współczesnych komputerach dodawanie trwa tyle co mnożenie, a czas dzielenia może być wielokrotnie dłuższy. Coraz częściej producenci arytmometrów zapewniają, że *wszystkie cyfry znaczące wyniku pojedynczej operacji arytmetycznej są cyframi dokładnymi*. Tak, ale nawet poprawne zaokrąglanie nie pozwala zapomnieć o tym, że nieomal każde dwuargumentowe działanie daje wynik obarczony błędem, i w *numerycznie złych algorytmach*, zwłaszcza po dłuższych obliczeniach, błędy te mogą się rozprzestrzenić na tyle, że ostateczny rezultat będzie bezwartościowy.

Dlatego cenną jest informacja z jaką dokładnością reprezentowane są liczby zmiennopozycyjne w komputerze. Bardzo popularną wielkością, podawaną przy charakteryzacji komputera jest *epsilon maszynowe*:

DEF. Epsilonem maszynowym (*machine epsilon*) nazywamy najmniejszą liczbę $\epsilon > 0$ taką, że $x = 1.0 + \epsilon;$ jest większe od $1.0\,.$ □

W praktyce jest to oszacowanie błędu względnego reprezentacji komputerowej liczb pamiętanych w typie zmiennej x. Jest to też informacja z której możemy się dowiedzieć *ile cyfr znaczących liczby jest przechowywanych w pamięci komputera* (dla typu zmiennej x).

Zadanie

Kiedy poznamy już *pętle* (rozdz. 6.12), wyznaczyć *epsilony maszynowe* dla poszczególnych typów zmiennopozycyjnych dostępnych w języku C.

6 Operatory i instrukcje

6.1 Lista operatorów języka C

W języku C jest bardzo dużo operatorów. Poznaliśmy już operatory arytmetyczne i operator przypisania. Tak, znak równości w instrukcji przypisania również

należy do operatorów i ma swój priorytet.

Wygodnie zatem będzie wypisać teraz pełną listę operatorów języka C z podaniem ich priorytetów (za [4, str. 109–110]). Dokładniejsze omówienie poszczególnych operatorów nastąpi dalej, w miarę potrzeby.

```
. -> [] sizeof
++ -- ~ ! + - & *                    operatory jednoargumentowe
* / %
+ -
<< >>
< <= > >=
== !=                                 porównania
&
^
|
&&
||
? :                                   op. warunkowy (trójargumentowy)
= *= /= %= += -= <<= >>= &= |= ^=     przypisania
,
```

Ogólnie możemy powiedzieć, że:

1. Operatory jednoargumentowe i operatory przypisania są prawostronnie łączne; wszystkie inne są lewostronnie łączne.
   Tak więc `a=b=c` oznacza `a=(b=c)`, `a+b+c` oznacza `(a+b)+c`, a `*p++` oznacza `*(p++)`, nie `(*p)++`.

2. Operatory zawarte w tym samym wierszu tabeli mają ten sam priorytet.

3. Operator umieszczony w tabeli wyżej ma wyższy priorytet aniżeli operatory umieszczone niżej.
   Na przykład `a+b*c` oznacza `a+(b*c)`, a `a+b-c` oznacza `(a+b)-c`, bo `+` i `-` mają ten sam priorytet oraz `+` i `-` są lewostronnie łączne.

6.2 Operatory przypisania arytmetycznego

Znamy już operator przypisania (podstawienia) $=$. W języku C istnieją *dwuznakowe operatory przypisania*, otrzymane przez dopisanie przed znakiem równości operatora arytmetycznego. Zwane są one **arytmetycznymi operatorami przypisania**:

Instrukcja	Wykonanie
$x \mathrel{+}= y;$	$x = x + y;$
$x \mathrel{-}= y;$	$x = x - y;$
$x \mathrel{*}= y;$	$x = x * y;$
$x \mathrel{/}= y;$	$x = x/y;$
$x \mathrel{\%}= y;$	$x = x\%y;$

6.2.1 Operatory inkrementacji i dekrementacji

Do jednoargumentowych operatorów należą dwuznakowe operatory pozwalające w zwarty sposób programować bardzo często wykonywane instrukcje powiększania albo zmniejszania wartości zmiennej o 1 :

Instrukcja	Wykonanie	
	wcześniej	później
$x = y{+}{+};$	$x = y;$	$y = y+1;$
$x = {+}{+}y;$	$y = y+1;$	$x = y;$
$x = y{-}{-};$	$x = y;$	$y = y-1;$
$x = {-}{-}y;$	$y = y-1;$	$x = y;$

Uwagi:

1. Zauważmy, że usytuowanie operatora względem zmiennej y należy interpretować jako ustalenie wzajemnego położenia w czasie dwóch operacji: *pobrania aktualnej wartości zmiennej* oraz *zmiany wartości tej zmiennej.*
2. Podobnie można traktować postać arytmetycznych operatorów przypisania, np: $+=$ – *najpierw dodaj aktualne wartości zmiennych, a dopiero potem zapisz wynik.*

6.3 Relacje i operatory logiczne

DEF. W języku C dysponujemy sześcioma **operatorami relacji** między wyrażeniami. Są to:

Operator	Znaczenie
==	równe
!=	różne od
<	mniejsze od
>	większe od
<=	mniejsze lub równe od
>=	większe lub równe od

□

Uwagi:

1. Priorytet operatora relacji jest niższy od priorytetu operatorów arytmetycznych.
2. Priorytet operatorów $<$, $>$, $<=$, $>=$ jest wyższy od priorytetu operatorów $==$ i $!=$.
3. Każdy operator relacji generuje wartość całkowitą:
   1 – jeżeli relacja jest spełniona,
   0 – w przeciwnym razie.

Przykład.
W języku C relacja $1/2 + 1/2 == 1$ jest fałszywa! □
Zadanie: Dlaczego? □

6.3.1 Operatory logiczne

W języku C nie istnieje oddzielny typ logiczny. Różne od zera wartości wyrażeń traktowane są jako *prawda*, a równe zeru – jako *fałsz*.
DEF. Do wykonywania operacji opartych na wartościach logicznych służą przede wszystkim znane **operatory logiczne** podane w kolejności malejącego priorytetu:

Operator	Znaczenie
!	negacja
&&	koniunkcja
||	alternatywa

□

Operatory logiczne dają jako wyniki 1 lub 0, a więc `!!x` daje 1 dla `x` różnego od zera i 0 dla `x` równego 0.

Trzeba pamiętać, że w języku C obowiązuje tzw. *krótkie obliczanie wartości logicznych*: jeżeli obliczony fragment wyrażenia logicznego już wyznacza wartość wyniku, to pozostała część wyrażenia jest pomijana.

6.4 Operatory bitowe

Operator	Znaczenie
~	negacja bitowa
<<	przesuwanie w lewo
>>	przesuwanie w prawo
&	iloczyn bitowy
^	bitowa różnica symetryczna
|	bitowa suma logiczna

Operatory bitowe mogą być stosowane jedynie do argumentów całkowitych (to znaczy char, short, int, long, ze znakiem lub bez). Traktują one liczby całkowite jako ciągi bitów: *Operatory przesuwania przesuwają bity lewego argumentu o ilość pozycji podaną w prawym argumencie.* Bity które "wysuną" się poza zakres liczby są pomijane. Przy przesuwaniu w lewo wolne pozycje wypełnia się zerami. Podobnie przy przesunięciu w prawo jeśli lewy argument ma typ bez znaku to wolne pozycje wypełnia się zerami. W przypadku przesunięcia w prawo jeśli lewy argument ma typ ze znakiem to na wolnych pozycjach powiela się bit znaku. Np. `3<<2` daje 12 zaś `3>>2` daje 0. Przesuwanie daje efekt równoważny mnożeniu lub dzieleniu przez odpowiednią potegę 2 (dlatego trzeba powielać bit znaku dla liczb ujemnych). Np. `3<<2` daje 12, `3>>2` daje 0 zaś `-5>>1` daje -2. Ujemna wartość licznika przesunieć lub wartość większa niż ilość bitów lewego argumentu jest błędem — efekt jest niezdefiniowany tzn. zależy od komputera, na komputerach PC *nie* uzyskuje się oczekiwanego wyniku.

Ilustracja.

Przesunięcie w lewo:

```
     10100111   Przed przesunięciem
101  00111      przesunięte o trzy w lewo
     00111000   Ostateczny wynik, z pominiętymi górnymi bitami
                i uzupełniony zerami
```

Przesunięcie w prawo:

```
10100111       Przed przesunięciem
  101001   11  przesunięte o dwa w prawo
00101001       Wynik po uzupełnieniu zerami
11101001       Wynik po powieleniu znaku
```

□

Pozostałe operatory bitowe wykonują operacje logiczne na wszystkich bitach osobno. Np. `4|1` daje 5, `5^3` daje 6, zaś `6&5` daje 4.

Operatory bitowe pozwalają upakować wiele wartości logicznych w pojedynczej zmiennej.

Przykład.

```
/* Definiujemy sobie pomocnicze stałe */
#define BIT0 1
#define BIT1 (1<<1)
...
#define BIT18 (1<<18)
...
int warunki;
...
    warunki|=BIT1; /* Ustawiamy bit1 */
    warunki&=~BIT18; /* Zerujemy bit18 */
...
/* Testujemy bit1 */
    if(warunki & BIT1)
...
    if(warunki & BIT18)
...
```

6.5 Wartość wyrażenia i efekty uboczne

W języku C wyrażenia mają wartość, a obliczanie tej wartości może powodować *efekty uboczne*. Zwykłe operatory arytmetyczne, logiczne, bitowe i porównania nie mają efektów ubocznych, tylko wartość. Jednakże operator przypisania stosujemy by zmienić wartość zmiennej. **Ta zmiana wartości jest *efektem ubocznym* instrukcji przypisania**. Niech nas nie zmyli to sformułowanie, ostateczną wartość wyrażenia w C najczęściej się pomija, a jedynym trwałym skutkiem są *efekty uboczne*. Np. funkcja `printf` produkuje wartość całkowitą (ilość znaków wypisanych na ekranie), a samo wypisywanie jest efektem ubocznym.

To co w języku C nazywa się wyrażeniem, w innym języku byłoby instrukcją czy wręcz ciągiem instrukcji — dowolne obliczenie nie zawierające pętli można

zapisać jako pojedyncze wyrażenie. Dzięki temu że wyrażenia takie jak przypisanie mają wartość można często uniknąć stosowania pomocniczych zmiennych. Bogaty wybór operatorów pozwala zapisać obliczenia w bardzo zwięzły sposób.

Nie powinno nas też niepokoić pomijanie wartości wyrażeń: jeśli kompilator zauważy że wartość wyrażenia jest niepotrzebna to jej nie oblicza (pilnuje jednak starannie by zaszły wszystkie *efekty uboczne*). W przypadku funkcji kompilator zwykle nie jest w stanie sprawdzić czy ma ona *efekty uboczne* i zawsze oblicza jej wartość.

Należy tu mocno podkreślić iż to, że możemy budować bardzo skomplikowane wyrażenia nie oznacza że powinniśmy. Wręcz przeciwnie — dowolnie duży program można zbudować z prostych wyrażeń. Głównym kryterium powinna być czytelność programu. Język C nie stawia tu sztucznych ograniczeń, ale programista powinien rozsądnie korzystać ze swojej swobody.

6.6 Operator przecinkowy

Znak przecinka też jest operatorem w C. Działa on ten sposób, że najpierw oblicza się wartość lewego argumentu, ignoruje się ją, następnie oblicza się wartość prawego argumentu i ona staje się wartością całego wyrażenia. Praktyczny efekt jest taki, jak byśmy mieli ciąg instrukcji, ale możemy go umieścić tam gdzie wymagane jest pojedyncze wyrażenie np. w intrukcjach pętli.

6.7 Wyrażenie warunkowe

DEF. Wyrażenie warunkowe ma postać:

$$<\textit{wyrażenie1}>?<\textit{wyrażenie2}> :<\textit{wyrażenie3}>$$ □

Przy wykonaniu najpierw oblicza się wartość <*wyrażenia1*>, jeśli jest ona różna od zera (prawdziwa), to oblicza się <*wyrażenie2*> i jego wartość staje się wartością całego wyrażenia; w przeciwnym razie oblicza się <*wyrażenie3*> i jego wartość staje się wartością całego wyrażenia. Np. `(x>y)?x:y` daje maksymalną z wartości `x` i `y`.

6.8 Instrukcja wyrażenie

Jest to najbardziej podstawowa i najczęściej używana instrukcja.
DEF. Wyrażenie po którym umieścimy średnik staje się instrukcją. □
Wartość wyrażenia oblicza się tak by zaszły efekty uboczne, a następnie pomija. W praktyce oznacza to że wyrażenie powinno zmieniać wartości zmiennych lub być wywołaniem funkcji, gdyż jeśli nie ma efektów ubocznych, to instrukcja jest zbędna.

Przykład

```
i++;
f(i,j);
i=12;
i==6;
j<<(i+3);
```

Wyrażenia w pierwszych trzech instrukcjach mają efekty uboczne (w drugim tylko potencjalnie, zależy to od funkcji `f`). Wyrażenia w dwu ostatnich instrukcjach nie mają efektów ubocznych a więc te dwie instrukcje są zbędne.

6.9 Instrukcja złożona

Język C w wielu miejscach gdzie chcemy umieścić kilka instrukcji pozwala umieścić tylko pojedynczą instrukcję. Nie powoduje to żadnych problemów, dzięki *instrukcji złożonej*.

DEF. Instrukcja złożona wygląda następująco:

{<*deklaracja1*>...<*deklaracjaN*>
<*instrukcja1*>...<*instrukcjaN*>} □

Tak więc ujmując ciąg instrukcji w nawiasy klamrowe tworzymy z nich pojedynczą instrukcję (blok). Dodatkowo, możemy taki ciąg instrukcji poprzedzić deklaracjami zmiennych — zmienne te są dostępne tylko wewnątrz tej instrukcji (bloku).

6.10 Instrukcja "if"

Znamy już postać relacji, podstawowe operatory logiczne, możemy je wykorzystać w programach. W algorytmach często trzeba decydować, które czynności powinny być wykonane. Zależy to od spełnienia takich czy innych warunków. W języku C służą do tego *instrukcje warunkowe*, a jedną z nich jest instrukcja "if".

DEF. Ogólna postać **instrukcji "if"** przedstawia się następująco:

if (<*wyrażenie*>) <*instrukcja1*> **else** <*instrukcja2*>

W wersji skróconej można pominąć jej drugą część zostawiając jedynie:

if (<*wyrażenie*>) <*instrukcja1*> □

Działanie:

1. Obliczana jest wartość <*wyrażenia*>, która musi być skalarem.
2. Jeśli jest ona różna od zera (*prawda*), to wykonywana jest <*instrukcja1*>. W przeciwnym razie (*fałsz*) wykonywana jest <*instrukcja2*>.
3. Zarówno <*instrukcja1*>, jak i <*instrukcja2*> mogą być instrukcjami złożonymi.
4. Instrukcje mogą być zagnieżdżone jedna w drugiej. Obowiązuje reguła: *słowo kluczowe* **else** *jest związane z ostatnim słowem* **if** *je poprzedzającym*. Aby uniknąć wątpliwości co do realizacji zaprogramowanych instrukcji używajmy nawiasów klamrowych { }.

6.11 Instrukcja pusta

DEF Instrukcja pusta jest to intrukcja składająca się z samego średnika. □

Instrukcja pusta nie wykonuje żadnych obliczeń, umieszcza się ją w programie by wypełnić miejsca gdzie zgodnie z regułami języka instrukcja jest obowiązkowa, a nie ma nic do roboty.

Uwaga: Nadmiarowe średniki w programie są traktowane jako instrukcje puste, co może spowodować że program będzie poprawny składniowo, ale będzie wykonywał inne obliczenia niż zamierzono.

6.12 Pętle

Pętle są instrukcjami służącymi do wielokrotnego wykonywania wybranego fragmentu programu.

6.12.1 Instrukcja "for"

DEF. Ogólna postać **instrukcji "for"** wygląda następująco:

for (<*wyrażenie1*> ; <*wyrażenie2*> ;<*wyrażenie3*>)
<*instrukcja*> □

Zawartość pierwszego wiersza nazywa się *nagłówkiem instrukcji*, a <*instrukcja*> może być instrukcją złożoną.

W trakcie działania instrukcja "for" wykorzystuje trzy wyrażenia podane w nagłówku i rozdzielone średnikami. Każde z tych wyrażeń może być puste, ale nie można pominąć średników. Każde z wyrażeń może zawierać *wyrażenia wielokrotne* oddzielone przecinkami. Jest to najpopularniejszy przykład użycia *operatora przecinkowego*.

- <*wyrażenie1*> wykonuje czynności przygotowawcze (inicjujące). Jest wykonywane tylko jeden raz na samym początku wykonywania pętli. Celem jego jest zainicjowanie jednej lub kilku zmiennych wykorzystywanych w pętli.
- Obrotami pętli sterują <*wyrażenie2*> i <*wyrażenie3*>. Przed każdym obrotem wyznaczana jest wartość <*wyrażenia2*>. Jest to *wyrażenie warunkowe*. Jego wartość równa zeru poleca zakończyć działanie instrukcji "for". Wartość różna od zera zezwala na wykonanie instrukcji podanych w nawiasach klamrowych, a następnie <*wyrażenia3*>.
- Jeżeli <*wyrażenie2*> jest puste, to uważa się, że jest zawsze prawdziwe i instrukcja **for** staje się pętlą bez końca. Przerwać jej działanie można tylko za pomocą instrukcji `return` lub `break`.
- <*Wyrażenie3*> jest wyrażeniem sterującym zmianami zmiennej, czy zmiennych wykorzystywanych w pętli i regularnie zmieniających swoje wartości z kolejnymi obrotami pętli.

Przykład. Sumowanie liczb parzystych od 10 do 20.

```
s=0;
for  ( i=10 ; i<=20 ; i+=2 )  s+=i;
```

Jak widać, czynnościami przygotowawczymi są: wyzerowanie sumatora `s`, która to instrukcja została wyciągnięta poza pętlę, i przyjęcie wartości początkowej

10 przez licznik i. Wielokrotnie wykonywaną instrukcję `s+=2;` możemy napisać bez nawiasów klamrowych, ponieważ jest instrukcją pojedynczą.

Można by potrzebny fragment programu napisać również w postaci pętli "for" z pustą instrukcją powtarzaną:

```
for  ( s=0, i=10 ; i<=20 ; s+=i, i+=2 );
```

albo:

```
for  ( s=10, i=10 ; i<20 ; i+=2, s+=i );
```

□

6.12.2 Instrukcja "while"

DEF. Instrukcja "while" ma postać następującą:

while (<*wyrażenie*>) <*instrukcja*>

□

Podobnie jak w instrukcji "for" i tutaj mamy *nagłówek pętli* z wyrażeniem warunkowym oraz instrukcję, która może być instrukcją złożoną i ma być powtarzana w kolejnych obrotach pętli.

Najpierw sprawdzana jest wartość <*wyrażenia*>. Jeśli jest ona równą zeru, to nic nie jest wykonywane i przechodzimy do instrukcji następnej. W przeciwnym razie – wykonywana jest <*instrukcja*>, a po jej wykonaniu ponownie jest sprawdzane <*wyrażenie*>. Pętla jest powtarzana do chwili, aż zerowa wartość <*wyrażenia*> każe przerwać i zakończyć działanie pętli.

Pętla "for" z poprzedniego rozdziału może być zaprogramowaną w równoważny sposób za pomocą instrukcji "while":

```
<wyra■enie1>;
while ( <wyra■enie2> )
{ <instrukcja>;  <wyra■enie3>; }
```

6.12.3 Instrukcja "do–while"

W porównaniu z instrukcją "while" tutaj została jedynie zmieniona kolejność wykonywania wielokrotnie powtarzanej <*instrukcji*> i weryfikacji warunku czy należy wykonać kolejny obrót pętli. Postać pętli:

do <*instrukcja*> **while** (<*wyrażenie*>);

Wyrażenie logiczne jest ujęte w nawiasy okrągłe, a cała instrukcja kończy się średnikiem.

6.12.4 Instrukcje break i continue

Podczas wykonywania pętli, albo funkcji (podprogramu) może zdarzyć się sytuacja, że trzeba instrukcję przerwać. W języku C służy do tego instrukcja **break**.

Przykład. Sumowanie z rozdz. 6.12.1.

```
for  (s=0, i=10 ; ; i+=2) {
  if (i > 20) break;
  s+=i;
 }
```

Dla pętli przydatną jest jeszcze druga instrukcja: **continue**. Może się zdarzyć, że w pewnych przypadkach nie należy wykonywać (wszystkich) instrukcji umieszczonych w ciele pętli. Instrukcja **continue** pozwala wcześniej przejść do kolejnego obrotu pętli.

Przykład. Jeszcze raz to samo sumowanie:

```
for  (s=0, i=10 ; ; i+=1) {
  if (i % 2) continue;
    else if (i>20) break;
  s+=i; }
```

6.13 Instrukcja skoku i etykiety

DEF. Etykieta ma postać:

$$<nazwa>:$$ □

DEF. Instrukcja skoku ("goto") ma postać:

$$\textbf{goto}\ \ <nazwa>;$$ □

Etykiety umieszczamy przed instrukcjami. Nazwa występująca w instrukcji skoku musi być nazwą etykiety. Instrukcja skoku zmienia normalną kolejność wykonania instrukcji, powodując że po wykonaniu instrukcji skoku następną wykonywaną instrukcją będzie instrukcja występująca po etykiecie wymienionej w instrukcji skoku.

Przykład

```
...
for(i=2;i<j;i++) {
     if(j%i == 0) {
             printf("%d jest dzielnikiem %d\n",i,j);
             goto dalej;
     }
 }
 printf("%d jest liczbą pierwszą\n",j);
dalej:
 ....
```

Intrukcją skoku można by zastąpić instrukcje `break` i `continue`, ale ze względu na czytelność programu intrukcję skoku należy stosować bardzo oszczędnie i o ile możliwe — zastępować konstrukcjami równoważnymi.

6.14 Instrukcja wyboru

DEF. Schemat **instrukcji wyboru ("switch")** wygląda następująco:

switch(<*wyrażenie*>)
{
case *stała1*: <*instrukcja1*>;
case *stała2*: <*instrukcja2*>;
...
default: <*instrukcjaN*>
} □

Działanie instrukcji **switch** przebiega następująco:

- oblicza się <*wyrażenie*>,
- porównuje się obliczoną wartość ze stałymi występującymi po słowach kluczowych **case**
- jeśli wartość <*wyrażenia*> jest równa którejś ze stałych, to wykonanie instrukcji **switch** kontynuuje się od instrukcji następującej po tej stałej,
- jeśli wartość wyrażenia jest różna od wszystkich stałych, to wykonanie przechodzi do instrukcji po **default**. Jeśli nie użyto słowa **default**, to wykonanie instrukcji **switch** się kończy, a program przechodzi do instrukcji następującej po instrukcji **switch**.
- wykonanie przechodzi przez kolejne przypadki (**case**), aby temu zapobiec używa się instrukcji **break**, **return** lub **goto**.

Przykład. Klasyfikacja znaków.

```
switch(c){
  case '0':
  case '1':
  case '2':
  case '3':
  case '4':
  case '5':
  case '6':
  case '7':
  case '8':
  case '9': printf("c jest cyfrą\n");
            break;
  case 'Z': printf("Du■a litera Z\n");
            break;
  case '$': printf("Znak dolara\n");
            break;
  default: printf("Coś innego\n");
            break;
}
```

Jak widać, wiele przypadków może odpowiadać tej samej czynności; zwykle też po każdej z czynności umieszcza się instrukcję **break**. **break** po ostatnim przypadku jest niepotrzebne, ale lepiej je dopisać żeby o nim nie zapomnieć przy zmianach w programie.

Instrukcja **switch** działa jak skok, tyle że dający wiele rozgałęzień, zaś przypadki **case** zachowują się tak jak etykiety, np. mogą się znaleźć we wnętrzu innej instrukcji.

Przykład. Rozwijanie pętli.

```
switch(a%4) {
    while(a>0) {
                printf("%d\n",a--);
        case 3: printf("%d\n",a--);
        case 2: printf("%d\n",a--);
        case 1: printf("%d\n",a--);
        case 0:         ;
    }
}
```

7 Wskaźniki i tablice

7.1 Wstęp – wskaźnik do zmiennej

Deklaracja zmiennej, np.

int k;

oznacza, że kawałek aktualnie wolnej pamięci o wymaganej wielkości (u nas 4 bajty) zostaje zarezerwowany dla zmiennej k.

DEF. Numer bajtu od którego ten fragment pamięci się rozpoczyna, nazywamy **adresem zmiennej** k. Czasem używana jest nazwa *lewostronna wartość zmiennej* k. □

DEF. W języku C istnieje jednoargumentowy **operator adresowy** & zwracający adres zmiennej. Może on być stosowany jedynie do zmiennych lub do elementów tablic. □

Przykład. Zadeklarujmy:

int k; unsigned long int a;

Instrukcja

a = &k;

przypisuje zmiennej a adres zmiennej k. □

Adresy, jako numery bajtów są liczbami całkowitymi. Pamięć operacyjna komputera jest duża; powinniśmy więc używać najdłuższego typu całkowitego do zapamiętywania być może dużych numerów.

DEF. Wskaźnik albo *zmienna adresowa* (*pointer = address variable*) jest zmienną używaną specjalnie do wskazywania innej zmiennej. Ogólna deklaracja wskaźnika ma postać:

<typ_danych> **<nazwa_wskaźnika>;*

□

Przykład. Uruchamiając program:

```
/* program wskazn1.c */
#include <stdio.h>
main()
{
int k,*pk;
unsigned long int a;
   k = 55;
   pk = & k;          // zapamiętujemy adres k we wskaźniku pk.
   printf("wart. k:  %d,          szesn.: %X \n",k,k);
   a = & k;              // zapamiętujemy adres k w zmiennej a.
   printf("wart. a: %ld,  szesn.: %p, *pk: %x \n",a,pk,*pk);
   printf("wart. a: %lu,   szesn.:   %x, *pk: %d \n",a,a,*pk);
}
```

otrzymujemy na ekranie:

```
wart. k:  55,          szesn: 37
wart. a: -1073743140,  szesn: 0xbffffadc, *pk: 37
wart. a: 3221224156,   szesn:   bffffadc, *pk: 55
```

□

DEF. Gwiazdka * po lewej stronie <*nazwy_wskaźnika*> jest **operatorem wyłuskania** (*dereference operator*) i takie połączenie jak w wywołaniu printf wyżej, oznacza: *weź wartość przechowywaną pod adresem wskazywanym przez wskaźnik* <*nazwa_wskaźnika*>. □

Uwagi do przykładu:

1. Do wartości zmiennej k możemy dotrzeć za pomocą nazwy tej zmiennej albo za pomocą wskaźnika.
2. W powyższym przykładzie próba użycia gwiazdki do zmiennej a byłaby błędem.
3. Zerowa wartość wskaźnika, np. wstawiona za pomocą instrukcji pk=0; , oznacza, że wskaźnik pk niczego nie wskazuje. Mówimy wtedy, że *wskaźnik jest zerowy.*

7.2 Tablice

Programy często muszą manipulować na dużych ilościach obiektów tego samego typu. Takie obiekty zwykle umieszczamy w tablicach.

DEF. Deklaracja tablicy ma postać:

<*typ*> <*nazwa*>[<*wielkość*>];

Do elementu tablicy odwołujemy się pisząc:

<*nazwa*>[<*numer elementu*>]; □

Znaczenie:

1. <*typ*> oznacza typ elementu tablicy np. `int`
2. <*nazwa*> jest nazwą deklarowanej tablicy
3. <*wielkość*> jest liczbą (stałą), jest to ilość elementów (miejsc) w tablicy
4. na elementach tablicy można operować tak jak na innych zmiennych
5. elementy tablicy numeruje się zaczynając od zera, ostatni element tablicy ma numer <*wielkość*>*-1*
6. pisząc kilkakrotnie po sobie nawiasy klamrowe otaczające liczby możemy zadeklarować tablicę wielowymiarową
7. tablicy można nadać wartość początkową, można wtedy pominąć <*wielkość*> — kompilator wyznaczy ją automatycznie

Przykład.

```
char znaki[50];      /* Tablica 50 znaków */
double liczby[20];   /* Tablica 20 liczb double */
char at[]={'a','l','a'}; /* Wartość początkowa, 3 elementy */
int ad[5]={1,2,3}; /* Wartość początkowa, uzupełniona zerami */
int tablica[50][10]; /* Dwuwymiarowa tablica liczb */
znaki[0]='a'; // Nadajemy wartość pierwszemu elementowi tablicy znaki
znaki[1]='l'; // Teraz drugi element
tablica[1][1]=1; // Nadajemy wartość
tablica[1][1]++; // Powiększamy o 1
```

Do przetwarzania tablic typowo używamy pętle.

Przykład. Obliczamy maksymalny element tablicy `liczby`:

```
maks=liczby[0];
ii=0;
for(i=1;i<20;i++) {
        if(liczby[i]>maks) {
                ii=i;
                maks=liczby[i];
        }
}
printf("Maksymalny element to: liczby[%d]=%f\n",ii,maks);
```

7.3 Tablice a wskaźniki

W języku C wskaźniki są bardzo silnie powiązane z tablicami: jeśli w wyrażeniu pojawi się nazwa tablicy to jest ona zamieniana na wskaźnik do jej pierwszego elementu. Dodanie do takiego wskaźnika liczby n powoduje że wskazuje on na element numer n tablicy tzn. wyrażenia `a[n]` i `*(a+n)` są równoważne. Warto tu podkreślić że dodanie 1 do wskaźnika oznacza powiększenie adresu związanego ze wskaźnikiem od wielkość wskazywanego obiektu.

Wskaźniki można też porównywać i odejmować — wynik odejmowania `a-b` daje się sensownie interpretować tylko gdy `a` i `b` wskazują na elementy tej samej tablicy — jest to wtedy ilość elementów pomiędzy `*a` a `*b`. Mamy tożsamość `a==a+(a-b)`.

Można też traktować wskaźniki jak tablice.

Przykład. Nastepujący fragment jest poprawny

```
int * c;
...
c[7]++;
```

Arytmetyka na wskaźnikach pozwala zapisać operacje na tablicach na dwa sposoby. Jawne użycie tablic i indeksowanie bywa czytelniejsze, z kolei arytmetyka na wskaźnikach często daje wydajniejszy program.

Przykład. Zakładamy że obowiązują deklaracje:

```
#define WIEL 20
int tab[WIEL];
int * pi;
int * epi;
int sum, i;
```

Wtedy następujące fragmenty dają tą samą wartość `sum`:

```
sum=0;
for(i=0;i<WIEL;i++) sum+=tab[i];
```

i

```
pi=tab; epi=pi+WIEL; sum=0;
for(;pi<epi;pi++) sum+=*pi;
```

7.4 Przydział pamięci (`malloc`)

Deklarowane tablice mają ustaloną wielkość — jest ona częścią deklaracji i nie zmienia się w czasie wykonania programu. Jeśli wielkość jest znana dopiero w czasie wykonania programu to trzeba przydzielać pamięć dynamicznie. Służy do tego funkcja `malloc`. Argumentem funkcji `malloc` jest potrzebna ilość pamięci (w bajtach). `malloc` zwraca wskaźnik do przydzielonej pamięci, albo w razie błędu (jeśli taka ilość pamięci nie jest dostępna) wskaźnik pusty `NULL`. Wartość funkcji `malloc` jest typu `void *`, można ją bezpiecznie przypisać do wskoźnika dowolnego typu (oczywiście zakładając że przydzieliliśmy dość pamięci). Gdy pamięć nie jest już potrzebna zwalniamy ją funkcją `free`. Aby wyznaczyć potrzebną ilość pamięci bardzo przydaje się operator `sizeof`, jego argumentem jest albo typ albo zmienna, a zwraca on ilość pamięci (w bajtach $\approx$ znakach) zajmowaną przez zmienną danego typu. Tu uwaga terminologiczna — w języku C bajtem nazywa się ilość pamięci zajmowaną przez znak i zakłada się że każdy inny obiekt potrzebuje całkowitą ilość bajtów (bajt zwykle składa się z 8 bitów, ale komputery VAX mają 9-cio bitowy bajt, a C dopuszcza też inne możliwości). Na naszym PC: `sizeof(int)` daje 4.

`sizeof` jest obliczany w czasie kompilacji (innymi słowy każdy typ zajmuje ustaloną ilość pamięci). Możemy używać tablic zmiennej wielkości, ale wtedy sami musimy pomnożyć wielkość elementu tablicy przez ilość elementów.

Przykład. Obliczanie współczynników dwumianowych

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

```
int * p;
int i, ii, j, n;
int main()
{
        printf("Podaj rząd współczynników:");
        scanf("%d",&n);
        if(n<=0) {
                printf("Rząd musi być liczbą dodatnią\n");
                return 1;
        }
        p=malloc((n+1)*sizeof(int)); /* Przydzielamy pamięć */
        if(!p) {
                printf("Brak pamięci");
                return 1;
        }
        for(i=0;i<=n;i++) {
                p[i]=1; ii=p[0];
                for(j=1;j<i;j++) {
                        int s=ii+p[j];
                        ii=p[j];
                        p[j]=s;
                }
        }
        printf("wynik:");
        for(i=0;i<=n;i++) printf(" %d -> %d,", i, p[i]);
        putchar('\n');
free(p);
        return 0;
}
```

W podanym przykładzie łatwo ustalić dla jakich `n` nastąpi nadmiar przy dodawaniu i zadeklarować tablicę wystarczającą dla poprawnych obliczeń — chcieliśmy jednak zademonstrować użycie `malloc` w przykładzie rozsądnej wielkości.

7.5 Teksty

Teksty przechowuje się w tablicach znaków, a operuje się na nich przy pomocy wskaźników. Ponieważ stałe teksty są stosowane bardzo często, kompilator traktuje je specjalnie — gdy w tekście źródłowym programu pojawi się łańcuch znaków to kompilator przydziela pamięć (tablicę znaków) odpowiedniego rozmiaru i umieszcza tam tekst (łańcuch), natomiast w miejscu (wyrażeniu) gdzie wystąpił łańcuch jest on zastępowany przez wskaźnik.

Przykład.

```
printf("ala");
```

i

```
char *ap="ala";
printf(ap);
```

i

```
char ac[4]={'a','l','a',0};
printf(ac);
```

oraz

```
char at[]="ala";
printf(at);
```

drukują ten sam tekst.

Jeśli chcemy sami wyprodukować nowy tekst, musimy dla niego przydzielić pamięć — albo deklarując odpowienią tablicę, albo przydzielając ją dynamicznie. Bardzo ważne jest poprawne określenie niezbędnej wielkości — błędy prowadzą do modyfikacji nieprzewidzianych miejsc w pamięci i są bardzo niebezpieczne.

8 Deklaracje

Jak już mówiliśmy każdy obiekt występujący w programie w C powinien być zadeklarowany (kompilator zezwala na użycie niezadeklarowanych funkcji, ale lepiej z tego nie korzystać — grozi to błędami). Pojedyńcza deklaracja w języku C może deklarować wiele obiektów:

Przyklad Deklarujemy równocześnie zmienną całkowitą, wskaźnik do zmiennej całkowitej, tablicę liczb całkowitch i funkcję o wartościach całkowitych z jednym argumentem całkowitym.

```
int i, * pi, tab[20], f(int j);
```

8.1 Struktury i unie

Stuktury pozwalają potraktować grupę powiązanych wielkości (nazywanych składowymi) jako jeden obiekt. Do składowej struktury odwołuje się przy pomocy nazwy. W przeciwieństwie do tablic składowe struktury mogą być różnych typów.

DEF. Deklaracja struktury ma postać:

struct <*nazwa rodzajowa*> { <*deklaracja1*>,...,<*deklaracjaN*>
}<*nazwa1*>,...,<*nazwaN*>;

Właściwości:

- <*nazwa rodzajowa*> jest deklarowana jako nowy rodzaj stuktury
- <*nazwa1*>,...,<*nazwaN*> są deklarowane jako zmienne typu **struct** <*nazwa rodzajowa*>
- <*nazwę rodzajową*> albo listę nazw zmiennych można pominąć
- <*deklaracja1*>,...,<*deklaracjaN*> deklarują składowe struktury, mogą one mieć dowolne typy
- strukturom można nadawać wartość początkową

Przykłady.

```
struct data { int dzien, miesiac, rok;};
struct data pierwszy_maja1960={1,5,1960};
struct gauss_int { long re , im;} gauss_i={0,1};
```

Do składowych struktury odwołujemy się przy pomocy znaku (operatora) `.` po którym piszemy nazwę składowej.

Przykłady.

```
struct data moja_data;
....
moja_data.dzien=1;
moja_data.miesiac=1;
moja_data.rok=1999;
...
moja_data.rok++;
```

Struktury bardzo często używa się razem ze wskaźnikami np.

```
struct element { int x, k ; struct element * next};
```

deklaruje element listy (składowa `next` jest wskaźnikiem do następnego elementu). Bardzo często również korzysta się ze wskaźników do struktur — wtedy pomaga operator `->`, jest on skrótem dla odwołania przez wskaźnik (`*`) i wyboru składowej `.`. Jeśli obowiązuje deklaracja

```
struct data * pdata;
```

to `(*pdata).dzien` i `pdata->dzien` oznaczają to samo.

Niekiedy chcemy przechowywać w tym samym obszarze pamięci wielkości różnych typów. Stosujemy w tym celu unie. Deklaracja unii różni się od deklaracji struktury zastąpieniem słowa kluczowego `struct` słowem `union`. Wszystkie składowe unii przechowuje się w tym samym miejscu w pamięci (dokładniej, przydziela się tyle pamięci ile trzeba dla największej składowej) — daje to oszczędność pamięci. Programista powinien zadbać o to by po zapisaniu wielkości jednego typu nie odczytywać jej jako wielkość innego typu (chyba że robi to celowo).

8.2 Funkcje

9 Rzeczy do wykorzystania

W następnym rozdziale przejdziemy do *opisu funkcji*, ale już teraz prezentujemy *nagłówki* dostępnych funkcji

Poszczególne znaki dostępne w komputerze sa identyfikowane poprzez swój kod. **kod ASCII** (*American Standard Code for Information Interchange*). Na przykład: 'A'⟷65, 'z'⟷122, '{'⟷123, '1'⟷123 .
Pojedynczy znak ujęty w apostrofy, jak wyżej, albo kilka znaków, czyli bf tekst należy ujmować w cudzysłów "Ala ma kota" i wówczas nazywa albo nazywa się **stałą znakową**. (Przypomnijmy, że się to **łańcuchem** albo **stałą łańcuchową**.) □

Deklaracje stałych znakowych albo łańcuchowych mają postać:

Łatwo wyobrazić sobie teraz mechanizm wczytywania z klawiatury jednego znaku przez komputer. Polega on na rozpoznaniu momentu naciśnięcia klawisza przez użytkownika, zidentyfikowanie znaku reprezentowanego przez wciśnięty klawisz poprzez przypisanie mu odpowiedniego kodu ASCII (np. 65 dla 'A') i przekazanie tego kodu, w postaci liczby całkowitej (typu *int*)

9.0.1 Stałe znakowe

9.0.2 Stałe całkowite

10 Dodatek. Przegląd funkcji bibliotecznych.

Poniżej przedstawiamy spis funkcji standartowych dostępnych w ANSI C, podzielony na grupy według zastosowania. Ponadto podajemy niektóre funkcje specyficzne dla systemów uniksowych. Podajemy tylko nazwy, szczegółowy opis jest dostępny w komputerze przy pomocy polecenia:

`man` *nazwa*

lub w dokumentacji biblioteki C (w pakiecie info).

Funkcje standartowe ANSI C:

- funkcje matematyczne:
  - `cos cosh exp sin sinh tan tanh`
  - `acos asin atan log`
  - `sqrt log10 atan2 pow`
  - `fabs ceil floor fmod`
  - `modf frexp ldexp`
  - `abs labs div ldiv`
- czytanie i pisanie — stdio:
  - po jednym znaku - `fgetc fputc getc getchar putc putchar ungetc`
  - liniami - `fgets fputs gets puts`
  - blokami - `fread fwrite`
  - formatowane - `fprintf fscanf printf scanf vfprintf vprintf`
  - `fclose fopen feof fgetpos fseek fsetpos ftell rewind clearerr ferror fflush freopen setbuf setvbuf`
  - operacje na plikach — `rename remove tmpfile tmpnam`
- konwersje: `atoi atol atof strtod strtol strtoul sscanf sprintf vsprintf`
- przydział pamięci:

  `calloc malloc realloc free`
- operacje na łańcuchach znaków:

  `strcpy strcat strlen strcmp strncat strncmp strncpy`

  `strchr strrchr strpbrk strstr`

  `strspn strcspn strsep strtok`

- klasyfikacja i zamiana znaków:

  `isalnum isalpha iscntrl isdigit isgraph islower isprint ispunct isspace isupper isxdigit`

  `tolower toupper`

- operacje na blokach pamięci:

  `memcpy memmove memset memcmp memchr`

- pobieranie i konwersja czasu i dat:

  `clock time`

  `asctime ctime mktime gmtime localtime strftime`

  `difftime`

- skoki nielokalne:

  `longjmp setjmp`

Funkcje uniksowe poza ANSI C:

- wejście-wyjście:

  `open close read write fcntl ioctl dup2 select`

- operacje na plikach:

  `link unlink stat mknod`

- czas:

  `gettimeofday times`

- procesy:

  `execve fork wait waitpid wait4`

- sygnały:

  `kill killpg signal sigaction sigprocmask sigpending sigsuspend pause`

- komunikacja sieciowa:

  `socket bind listen accept connect send recv getsockname getsockopt setsockopt getprotoent shutdown socketpair`

- pamięć wspólna:

  `shmget shmat shmdt shmctl`

Bibliografia

[1] Ward Cheney, David Kincaid: *Numerical mathematics and computing*,Brooks Publishing Company, California, rok. .....

[2] Brian W.Kernighan, Dennis M.Ritchie: *Język ANSI C*, WNT, Warszawa 1998.

[3] Alex Ragen: *Leksykon języka C*, WNT, Warszawa 1990.

[4] Bjarne Stroustrup: *Język C++*, WNT, Warszawa rok.

[5] Andrzej Zalewski: *Programowanie w językach C i C++ z wykorzystaniem pakietu Borland C++*, Nakom, Poznań 1995.

[6] Tony Zhang: *Poznaj C w 24 godziny* (tłum. Adam Majczak), Intersoftland, Warszawa 1998.