Stałe

Przykład	Nazwa
1234	int
2345l	long int
2345L	long int
1234u	unsigned int
2345U	unsigned int
5678ul	unsigned long int
12.34	double
1e-2	double
1.2e-2	double
2.3f	float
2.3F	float
3.4e-2l	long double
3.4e-3L	long double

Zapis wartości stałych całkowitych w różnych kodach

Rodzaj kodu		
dziesiętny	ósemkowy	szesnastkowy
31	037	0x1f
31	037	0X1F

Stała znakowa – 'a'
'0' versus 48

Bitowy wzorzec bajtu

```
'\ooo' -- ooo  oznacza jedna, dwie
          lub trzy liczby osemkowe
albo
'\xhh' -- hh  oznacza jedna lub wiecej
          cyfr szesnastkowych
```

Lista sekwencji specjalnych języka C

```
\a      znak alarmu
\b      znak cofania
\f      znak nowej strony
\n      znak nowego wiersza
\r      znak powrotu karetki
\t      znak tabulacji poziomej
\v      znak tabulacji pionowej
\\      znak \
\?      znak zapytania
\'      znak apostrofu
\"      znak cudzyslowu
\ooo    liczba osemkowa
\xhh    liczba szesnastkowa
```

Wyrażenie stałe - wyrażenie, w którym występują wyłącznie stałe.

```
#define MAXL 10000
#define VTAB '\013'
#define VTAB '\x7'
#define VTAB '\X7'
#define VTAB '\v'
```

Stała napisowa - ciąg złożony z zera lub większej liczby znaków umieszczonych pomiędzy znakami cudzysłowu, np.

```
"To jest napis!"
lub
"" /* napis pusty */
```

Technicznie napis jest tablicą o liczbie elementów o jeden większą niż liczba znaków. Dochodzi znak końca znaku '
0'.

Stała 'x' jest różna od "x".

Napisy mogą być sklejane podczas kompilacji programu:

```
"Hej!" "Przygodo!"
jest identyczne z:
"Hej!Przygodo!"
```

Stała wyliczeniowa (enumeration constant)
(Lista wartości stałych całkowitych)

```
enum boolean {NO,YES};
enum escapes {BELL='\a', BACKSPACE='\b', TAB='\t',
              NEWLINE='\n', VTAB='\v', RETURN='\r'};

enum months {STY=1, LUT, MAR, KWI, MAJ, CZE, LIP,
             SIE, WRZ, PAZ, LIS, GRU};
   /*  miesiace: luty jest drugi, marzec trzeci itd. */
```

Nazwy w różnych wyliczeniach muszą być różne. W tym samym wyliczeniu wartości mogą się powtarzać.

Nazwa typu *enum* dzieli tą samą przestrzeń, co nazwy typów struktur i unii.

Nazwy zmiennych wyliczeniowych należą do tej samej klasy, co identyfikatory zwykłych zmiennych.

Deklaracje

```
int lower,upper, step;
char c,lin[1000];
```

```
int lower;
int upper;
int step;
char c;
char lin[1000];
```

W deklaracjach można nadać wartości początkowe:

```
char esc='\\';   /*  znak \ */
int  i=0;        /*  licznik iteracji  */
int  limit=MAXLINE+1;  /* maksymalna liczba iteracji */
float eps=1.0e-5;      /* parametr dok\l adno\' sci  */
```

Wartości początkowe domyś lne zmiennych:

```
statycznych i zewn\c etrznych -- 0
automatycznych -- przy jawnie okreslonych wartosciach
                  poczatkowych ta sama wartosc
                  przy kazdym wywolaniu funkcji badz
                  wejsciu do bloku.
                  bez jawnie okreslonej wartosci
                  poczatkowej maja wartosci
                  przypadkowe.
```

Kwalifikator *const* (stały) (można użyć do deklaracji dowolnej zmiennej); Mówi, że jej wartość nie będzie zmieniana.

```
const double e=2.1234e-2;
const char msg[]="Uwaga";
int strlen(const char []);  /* zawartosc tablicy
               - argumentu nie moze byc zmieniona
               wewnatrz funkcji  */
```

Próba zmiany wartości zmiennej zadeklaqrowanej jako *const* kończy się w sposób zależne od implementacji.

Operatory arytmetyczne	
+	dodawanie
-	odejmowanie
	mnożenie
/	dzielenie
%	dzielenie modulo

Operatora % nie można stosować do zmiennych typu *float* i *double*.

Dla ujemnych argumentów operacji zarówno kierunek zaokrąglenia wyniku po obcięciu części ułamkowej przez dzielenie całkowite, jak i znak liczby, która jest wynikiem dzielenia modulo %, są zależne od maszyny. Akcje podejmowane po wystąpieniu nadmiaru lub niedomiaru także zależą od maszyny.

Dwuargumentowe operatory + i - mają ten sam priorytet. Jest on niższy od priorytetu operatorów *,/ oraz %, który z kolei jest niższy od priorytetu operatorów jednoargumentowych + i -. Operatory arytmetyczne są lewostronnie łączne.

Relacje i operatory logiczne

Operatory relacji:

```
>, >=, < <=
\end{verbatim

Operatory por\' ownania:

\vspace{0.2cm}
\begin{verbatim}
==, !=



&& -- operator koniunkcji logicznej

|| -- operator alternatywy logicznej

! -- opeartor negacji



i<j-1  jest r\' ownowa\. zne i<(j-1)



for (i=0; i<lim-1 && (c=getchar())
          != '\n' && c!= EOF; ++i)
  s[i]=c;
```

Przekształcenia typów

Ogólna zasada:

automatycznie wykonuje się tylko takie przekształcenia, w których argument "ciaśniejszy jest zamieniany na obiekt "obszerniejszy" bez utraty informacji.

Wyrażenia, ktŕe nie mają sensu, jak np. użycie obiektu typu *float* do indeksowania tablicy, są niedozwolone.

Wyrażenia, w których może wystąpić utrata informacji, mogą powodować wypisanie komunikatu ostrzegawczego, ale nie są zabronione.

Przykład:

```
\*  lower: zamie\' n c na ma\l\c a liter\c e;
    tylko dla ASCII  */
int lower(int c)
{
  if (c >= 'A' && c <= 'Z')
    return c+'a'-'A';
  else
    return c;
}
```

Niejawne przekształcenia arytmetyczne działają zgodnie z oczekiwaniami. Ogólnie, jeśli argumenty operatora dwuargumentowego (np. + czy *) są różnego typu, to typ "mniejszy" jest promowany do typu "większego" przed wykonaniem operacji. Typem wyniku jest typ "większy".

Nieformalny zestaw reguł przedstawia się następująco:

- Jeśli którykolwiek z argumentów jest typu long double, to ten drugi zostanie przekształcony do long double.
- W przeciwnym przypadku, jeśli typem któregokolwiek argumentu jest double, to ten drugi zostanie przekształcony do double.
- W przeciwnym przypadku, jeśli typem jednego z argumentów jest float, to drugi argument zostanie przekształcony do typu float.
- W przeciwnym przypadku, wszystkie obiekty char i short są przekształcane do typu int.
- następnie, jeżeli którykolwiek z argumentów ma kwalifikator long, to to drugi zostanie przkształcony do long.

Reguły przekształceń są znacznie bardziej skomplikowane dla argumentów *unsigned*.

Każdy argument funkcji jest wyrażeniem, dlatego przy przekazywaniu funkcjom argumentów również zachodzą przekształcenia typów.

Oeprator nazywany *rzutem*:

(nazwa typu) wyrażenie

wyrażenie jest przekształcane według podanych reguł do typu określonego przez *nazwa-typu*. Np.

sqrt((double) n);

Operatory zwiększania i zmniejszania: ++ i –

```
#include <stdio.h>
/* zlicz znaki wejsciowe; wersja 1  */
void main()
{
  long nc=0;

  while (getchar() != EOF)
    ++nc;
  printf("%ld\n",nc);
}
```

Działają tylko w odniesieniu do zmiennych.

(i+1)++ jest BŁEDEM!