Bitowe operatory logiczne

Uwaga: Nie można ich używać do danych typu *float* lub *double*.

&	bitowa koniunkcja
|	bitowa alternatywa
^	bitowa różnica symetryczna
«	przesunięcie w lewo
»	przesunięcie w prawo
~	uzupełnienie jedynkowe

Koniunkcja bitowa &

& stosuje się często do "zasłaniania" pewnego zbioru bitów, np.

$$c = n\&0177;$$

są zerowane wszystkie oprócz siedmiu najniższych bitów wartości zmiennej n.

Różnice między & i &&

```
Np.  Jesli           x=1,   y=2            to

    x & y         ma wartosc   0

    x && y                     1
```

Bitowy operator alternatywy |

Używany do "ustawiania" bitów. Np.

$$x = x|MASK;$$

ustawia 1 na tych bitach w zmiennej x, które w $MASK$ są równe 1.

Bitowa różnica symetryczna ^

Ustawia jedynkę na każdej pozycji bitowej tam, gdzie bity w obu argumentach są różne, a zero tam, gdzie są takie same. Np.

$$x = 011\hat{}022;$$

daje w rezultacie wartość 0.

Operatory przesunięcia « oraz »

Służą do przesuwania argumentu stojącego po lewej stronie operatora o liczbę pozycji określoną przez argument stojący po prawej stronie. Np.

$$y = x << 2;$$

przesuwa x w lewo o dwie pozycje, zwolnione bity wypełnia się zerami (operacja równoważna mnożeniu przez 4.

$$y = x >> 2;$$

dla wielkości typu *unsigned* zwolnione bity zawsze są wypełniane zerami. Natomiast dla wielkości ze znakiem spowoduje to na pewnych maszynach wypełnienie tych miejsc bitem znaku (przesunięcie arytmetyczne), a na innych zerami (przesunięcie logiczne).

Dopełnienie jedynkowe ~

Zamienia każdy bit 1 na 0 i odwrotnie. Typowe użycie:

$$y = x\&\tilde{}077;$$

zasłania się zerami ostatnie sześć bitów zmiennej x.

Uwaga: Ta konstrukcja nie zależy od maszyny, podczas gdy:

$$y = x\&0177700;$$

zakłada 16-bitowe słowo maszynowe.

Przykład:

```
/*  getbits: daj n bitow x od pozycji p
    Zerowa pozycja bitu jest prawy koniec x;
    n, p sa sensownymi wielkosciami calkowitymi
*/
unsigned getbits(unsigned x, int p, int n)
{
  return (x >> (p+1-n)) & ~(~0 << n);
}
```

Operacje bitowe na znakach

Kody ASCII								
Bit	7	6	5	4	3	2	1	0
		0 - cyfry	0 - duże litery	0 - a-o				
		1 - litery	1 - cyfry /małe litery	cyfry /p-z				

```
char toupper(char c)
/*  Funkcja zamienia male litery na duze  */
{
  char maska=223;   /*  223    -    11011111   */
  return c & maska;
}
```

```
char tolower(char c)
/*  Funkcja zamienia duze litery na male  */
{
  return c | 32;    /*  32    -    00100000    */
}
```

```
char swapcase{char c)
/*  Zamiana duzych liter na male i odwrotnie  */
{
  return c ^ 32;
}
```

```
int razy10(int n)
/*  Mnozenie liczby calkowitej przez 10
    przy pomocy operatorow przesuniecia  */
{
  int m,p;
  m=n<<1;
  p=m<<2;
  return m+p;
}
```

Operatory i wyrażenia przypisania *op=*

i=i+3; jest równoważne i+=3;

op jest jednym z operator\' ow:

+ - * / % << >> & ^ |

UWAGA: Jezeli wyr1 i wyr2 sa wyrazeniami, to

wyr1 op= wyr2; jest rownowazne z
wyr1=(wyr1) op (wyr2);

Zatem przypisanie

x *= y+1; jest odpowiednikiem x=x*(y+1);

Przykład:

```
/*  Zlicz bity argumentu o wartosci  1  */
int bitcount(unsigned x)
{
  int b;
  for (b=0; x!=0; x >> 1)
    if (x & 01)
      b++;
  return b;
}
```

Wyrażenia warunkowe

Instrukcja realizująca funkcję z=max(a,b)

```
if (a>b)

  z=a;

else

  z=b;
```

jest rownowazna z=(a>b) ? a:b;

albo z= a>b ? a:b;

Priorytet operatora ?: jest bardzo niski.

Wyrażenie warunkowe często wpływa na zwięzłość programu. Np.

```
1)
for (i=0; i<n; i++)
  printf("%6d%c", a[i],
          (i%10 == 9 || i== n-1) ? '\n':' ');
2)
printf("Masz %d czes%s.\n",n, n==1 ? "c":"ci");
```

Priorytety i kolejność obliczeń

Priorytety i łączność operatorów

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

Jednoargumentowe operatory `+, -, * oraz &` mają wyższy priorytet, niż ich odpowiedniki dwuargumentowe.

Lewostrona łączność operatora oznacza, że jeśli na tym samym poziomie występuje kilka operatorów, to najpierw jest wykonywany operator lewy.

W języku C nie określa się kolejności obliczania argumentów operatora. Wyjątki, to:
`&&, ||, ?:` oraz `,`
Np. w instrukcji `x=f()+g();` f może być wykonana przed g lub odwrotnie.

Nie jest określona kolejność obliczania wartości argumentów funkcji. Instrukcja

$$printf(\texttt{"\%d \%d\textbackslash n"}, ++n, power(2,n));$$

może generowć różne wyniki w zależności od kompilatora.

Możliwe są również inne efekty "uboczne" (generowane przez wywołania funkcji, zagnieżdżone instrukcje przypisania oraz operatory zwiększania i zmniejszania), np.

a[i]=i++;

zależy od kolejności aktualizacji wartości zmiennych biorących udział w obliczeniach.

Konkluzja: Pisanie programów zależnych od kolejności wykonania obliczeń należy do złej praktyki programowania w każdym języku.

Sterowanie

Instrukcje i bloki

Wyrażenie staje się instrukcją po zakończeniu średnikiem, np.

```
x=0;
i++;
printf{"Uwaga!\n");
```

Nawiasy klamrowe `{ i }` są używane do tworzenia bloku. (Zmienne można zadeklarować wewnątrz każdego bloku.) Po nawiasie zamykającym blok nie może występować średnik.

Instrukcja `if-else`

```
if (wyrazenie)
  instrukcja1
else
  instrukcja2
```

Część **else** można pominąć.
Najczęściej pisze się

```
if (wyrazenie)
```

zamiast

```
if (wyrazenie != 0)
```

Każda część **else** jest przyporządkowana najbliższej z poprzednich instrukcji **if** nie zawierającej części **else**, np.

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

Dwuznaczność ta jest szczególnie szkodliwa w takich sytuacjach, jak

```
if (n>0)
  for (i=0; i<n; i++)
    if (s[i] > 0) {
      printf("...");
      return i;
    }
else   /*  Zle  */
  printf("Blad -- n jest ujemne);
```

Wcięcie wyraźnie pokazuje, o co chodzi programiście, ale kompilator przyporządkuje **else** wewnętrznej funkcji **if**.

konstrukcja else-if

```
if (wyrazenie)
  instrukcja
else if (wyrazenie)
  instrukcja
else if (wyrazenie)
  instrukcja
```

Przykład:

```
/* szukanie x metoda bisekcji wsrod
   v[0]<=v[1]<=...<=v[n-1] */
int binsearch(int x, int v[], int n)
{
  int low=0, high, mid;
  high=n-1;
  while (low <= high){
    mid=(low+high)/2;
    if (x<v[mid])
      high=mid-1;
    else if (x>v[mid])
      low=mid+1;
    else    /* znaleziono  */
      return mid;
  }
  return -1;   /*  Nie znaleziono  */
}
```