Podstawy Programowania 1

Podstawy języka C

Arkadiusz Chrobot

Katedra Systemów Informatycznych

11 października 2022

Plan

1. Inicjacja zmiennych
2. Stałe raz jeszcze
3. Operatory
   - Operatory relacyjne
   - Operatory arytmetyczne
   - Operatory logiczne
   - Operatory bitowe
   - Operator trójargumentowy
   - Rzutowanie
   - Inne operatory
   - Kolejność działań
4. Podstawowe wejście i wyjście

Inicjacja zmiennych

Inicjacja zmiennej jest to przypisanie jej wartości przed pierwszym użyciem, a dokładniej przed pierwszym odczytem jej zawartości. Brak inicjacji jest jedną z najczęstszych przyczyn błędnego działania programów. Zmienne globalne mają zerową wartość początkową. Czasem taka wartość początkowa nie jest prawidłowa i chcielibyśmy ją zmienić.
W tym wykładzie dla wygody większość zmiennych będzie miała nazwy jednoliterowe.

Instrukcja przypisania

Najprostszą metodą inicjacji zmiennej jest przypisanie jej wartości za pomocą instrukcji przypisania, która w języku C zapisywana jest za pomocą symbolu =. Ta instrukcja jest używana również wtedy, gdy chcemy zmienić wartość zmiennej. Formalnie ta instrukcja dokonuje wyznaczenia wartości wyrażenia stojącego po jej prawej stronie i konwersji typu wyniku na typ zmiennej znajdującej się po jej lewej stronie. Efektem ubocznym, którym jako programiści jesteśmy głównie zainteresowani, jest przypisanie tej zmiennej wartości wyrażenia. W języku C instrukcja przypisania jest także operatorem.

Sposoby inicjacji

Inicjacja zmiennych całkowitoliczbowych

Zmiennej można przypisać wartość w miejscu jej deklaracji. Listing zamieszczony poniżej pokazuje jak to zrobić dla zmiennych całkowitoliczbowych.

```
int a = 3, b = 075, c = 0xab, d = 1u;

int main(void)
{
    return 0;
}
```

Jeśli liczba jest poprzedzona zerem, to taki zapis oznacza, że jest ona zapisana w kodzie ósemkowym, jeśli jest poprzedzona przedrostkiem `0x`, to znaczy to, że jest to liczba zapisana w kodzie szesnastkowym. Liczby mogą być zakończone przyrostkiem, który w przypadku liczb całkowitych może być literą `u` lub `U` i oznacza wtedy, że liczba jest bez znaku, lub literą `l` lub `L` oznaczającą wartość typu `long`. Można oba te przyrostki połączyć w `lu` lub `LU`.

Sposoby inicjacji

Inicjacja zmiennych całkowitoliczbowych

Korzystając z faktu, że w języku C instrukcja przypisania jest równocześnie operatorem można zainicjować kilka zmiennych tą samą wartością w następujący sposób:

```
int a, b, c;

int main(void)
{
    a=b=c=3;
    return 0;
}
```

Sposoby inicjacji

Inicjacja zmiennych znakowych

Zmiennym znakowym (typu `char`) można przypisać liczbę oznaczającą kod ASCII lub znak, umieszczając go w apostrofach.

```
char a = 65, b = 'a';

int main(void)
{
    return 0;
}
```

Przypomnijmy, że zmienne takiego typu mogą być używane nie tylko do przechowywania znaków, ale także liczb.

Sposoby inicjacji

Inicjacja zmiennych zmiennoprzecinkowych

Zmienne zmiennoprzecinkowe (`float`, `double` i `long double`) możemy inicjować z użyciem zapisu dziesiętnego, gdzie zamiast przecinka stosujemy kropkę lub za pomocą notacji wykładniczej, gdzie np. zapis `1e-2` oznacza $1 \cdot 10^{-2}$, czyli `0.01`. Liczby w notacji dziesiętnej mogą być zakończone znakiem `f` jeśli chcemy aby były typu `float`.

```
double a = 0.001f, b = 0.02, c = 1e-2;

int main(void)
{
    return 0;
}
```

Stałe i słowo kluczowe const

Zamiast instrukcji #define preprocesora można do definiowania stałej użyć słowa kluczowego const, np.:

```
const int SEVEN = 7;

int main(void)
{
    return 0;
}
```

Taki zapis oznacza, że wartość SEVEN nie zmieni się w czasie wykonania programu[1]. Nazwy tak zdefiniowanych stałych, zgodnie z przyjętą konwencją, piszemy wielkimi literami.

[1] Są sposoby, aby zmienić wartość tej stałej, ale nie będą tu opisane.

Operatory

Operatory, podobnie jak w matematyce służą do budowania wyrażeń, których wartości mogą być zapisane przy pomocy instrukcji przypisania, która też jest operatorem, do zmiennej określonego typu. W języku C dostępnych jest kilka różnych kategorii operatorów. Nie wszystkie będą omówione na tym wykładzie. Podobnie jak w matematyce w informatyce operatory mają przypisaną kolejność w jakiej są wykonywane w wyrażeniach. Dodatkowo w języku C mają także cechę, nazwaną *wiązaniem*, która określa sposób ich działania.

Operatory relacyjne

Ogólnie, operatory relacyjne służą do określania relacji zachodzących między wartościami wyrażeń stojących po ich lewej i prawej stronie. Jeśli ta relacja jest prawdziwa, to zwracają wartość 1, w przeciwnym przypadku 0.

Operatory	**Opis działania**
==, !=	Operator porównania i operator „różne”. Pierwszy daje w wyniku jeden, jeśli oba argumenty są takie same, drugi jeśli są różne.
<, >, <=, >=	operatory „mniejsze”, „większe”, „mniejsze lub równe”, „większe lub równe”

Operatory arytmetyczne

Operatory	Opis działania
++, --	Jednoargumentowe operatory inkrementacji i dekrementacji. Mogą być stosowane zarówno przed, jaki i po argumencie np. `++a;` (preinkrementacja), `a++;` (postinkrementacja), `--a;` (predekrementacja), `a--;` (postdekrementacja). Operatory te zwracają wartość zmiennej. Operatory „post" wiążą lewostronnie, a „pre" prawostronnie.
+, -	Operatory te mogą być dwuargumentowe, i wtedy oznaczają dodawanie i odejmowanie, lub jednoargumentowe. W tym ostatnim przypadku operator - oznacza zmianę znaku wartości zmiennej, a + nie ma żadnego efektu.
*, /	Operatory mnożenia i dzielenia. **Uwaga**: Jeśli argumentami operatora dzielenia są liczby (wyrażenia) całkowite, to i wynik jest liczbą całkowitą, niezależnie od tego w jakiej zmiennej go zapiszemy.
%	Operator reszty z dzielenia (modulo). W języku C działa również dla liczb ujemnych.

Operatory arytmetyczne

Przepełnienie

Należy pamiętać, że operacje arytmetyczne na liczbach całkowitych podlegają *przepełnieniu*, tzn. jeśli obliczona wartość wyrażenia przekracza zakres zmiennej, w której wynik jest zapamiętywany, to zostanie on „ucięty".

```
unsigned char a = 255;
char b = -128;

int main(void)
{
    a = a + 1; //Zmienna "a" ma wartość 0;
    b = b - 1; //Zmienna "b" ma wartość 127.

    return 0;
}
```

Operatory arytmetyczne

Przepełnienie — objaśnienie

$$\begin{array}{rr} & 1\,1\,1\,1\,1\,1\,1\,1 \\ + & 0\,0\,0\,0\,0\,0\,0\,1 \\ \hline & 1\,0\,0\,0\,0\,0\,0\,0\,0 \end{array}$$

$+1 \quad +1 \quad +1 \quad +1 \quad +1 \quad +1 \quad +1 \quad +1$

Mechanizm powstawania nadmiaru w zmiennej typu `unsigned char`

Operatory arytmetyczne

Przepełnienie — objaśnienie

$$\begin{array}{rr} & 1\,0\,0\,0\,0\,0\,0\,0 \\ + & 1\,1\,1\,1\,1\,1\,1\,1 \\ \hline & 1\,0\,1\,1\,1\,1\,1\,1\,1 \end{array}$$

$+1$

Mechanizm powstawania nadmiaru w zmiennej typu `char`

Operatory arytmetyczne

Operator modulo - objaśnienie

Operator reszty z dzielenia działa wyłącznie dla liczb całkowitych. Wynik przez niego obliczony zawsze spełnia równanie `(x/y)*y+(x%y)==x`. Przykłady poniżej pokazują jak on działa dla liczb ujemnych.

```
5%-2  // Wyrażenie ma wartość 1.
-5%2  // Wyrażenie ma wartość -1.
-5%-2 // Wyrażenie ma wartość -1.
```

Podobnie jak w zwykłym dzieleniu, dzielnik nie może być zerem.

Operatory arytmetyczne

Operator dzielenia - objaśnienie

Poniżej podano przykłady wyrażeń, dla których otrzymane wyniki są różne, choć z punktu widzenia matematyki zapisy wyrażeń są takie same. Należy pamiętać, że kompilatory języka C traktują liczby bez kropki dziesiętnej jako liczby całkowite.

```c
double a;

int main(void)
{
    a = 4/5;   // Wynik wyrażenia to 0.
    a = 4.0/5; // Wynik wyrażenia to 0.8.

    return 0;
}
```

Operatory arytmetyczne

Skrócony zapis

Zapis wyrażenia arytmetycznego można skrócić, jeśli występuje w nim ta sama zmienna, do której przypisywany jest wynik.

```
int a=2, b=2;
int main(void)
{
    b+=a; // Zamiast b=b+a;
    b-=a; // Zamiast b=b-a;
    b*=a; // Zamiast b=b*a;
    b/=a; // Zamiast b=b/a;
    b%=a; // Zamiast b=b%a;

    return 0;
}
```

Operatory arytmetyczne

Objaśnienie działania operatorów inkrementacji i dekrementacji

```
int a = 4, b;
int main(void)
{
    b=++a; //Po wykonaniu: zmienna b ma wartość 5, zmienna a ma wartość 5.
    a=4;
    b=a++; //Po wykonaniu: zmienna b ma wartość 4, zmienna a ma wartość 5.
    a=4;
    b=--a; //Po wykonaniu: zmienna b ma wartość 3, zmienna a ma wartość 3.
    a=4;
    b=a--; //Po wykonaniu: zmienna b ma wartość 4, zmienna a ma wartość 3.
    a=4;
    a++; //Po wykonaniu: zmienna a ma wartość 5.
    a=4;
    ++a; //Po wykonaniu: zmienna a ma wartość 5.
    a=4;
    a--; //Po wykonaniu: zmienna a ma wartość 3.
    a=4;
    --a; //Po wykonaniu: zmienna a ma wartość 3.

    return 0;
}
```

Operatory logiczne

Operatory logiczne są używane do budowania wyrażeń, których wartość jest określana jako prawda lub fałsz. W języku C dostępne są następujące operatory należące do tej kategorii:

Operatory	**Opis działania**
||, &&	Dwuargumentowe operatory sumy logicznej (or) i iloczynu logicznego (and).
!	Jednoargumentowy operator negacji logicznej.

Operatory te zwracają wartości 1 (prawda) i 0 (fałsz), ale należy pamiętać, że w języku C **każde wyrażenie, którego wartość jest różna od zera jest prawdziwe, a wyrażenia o wartości zero są fałszywe**. Operator && zwraca w wyniku prawdę, jeśli oba jego argumenty są prawdziwe, w przeciwnym przypadku daje fałsz. Operator || zwraca w wyniku fałsz, gdy oba jego argumenty są fałszywe, w przeciwnym przypadku zwraca prawdę.

Operatory logiczne

Skracanie obliczeń

W złożonych wyrażeniach z użyciem operatorów && i || stosowana jest technika skracania obliczeń (ang. *short–circuit evaluation*). W przypadku operatora && oznacza ona, że jeśli jego pierwszy argument jest fałszywy, to wartość drugiego nie jest wyznaczana, od razu wiadomo, że całe wyrażenie musi być fałszywe. Podobnie dla || - jeśli pierwszy argument jest prawdziwy, to nie wylicza się wartości drugiego - wiadomo, że wyrażenie jest prawdziwe.

```
int a,b;
int main(void)
{
    (a=0)&&(b=4); //Obie zmienne mają wartość 0 po wykonaniu wyrażenia.
    (a=4)&&(b=3); //Po wykonaniu wyrażenia zmienna a ma wartość 4, a zmienna b wartość 3.
    (a=0)||(b=0); //Obie zmienne mają wartość 0 po wykonaniu wyrażenia.
    (a=3)||(b=4); //Po wykonaniu wyrażenia zmienna a ma wartość 3, a zmienna b wartość 0.

    return 0;
}
```

Operatory bitowe

Operatory te są podobne do operatorów logicznych, ale działają na poszczególnych bitach wartości zapisanych w zmiennych całkowitoliczbowych. Nie mogą być stosowane ze zmiennymi zmiennoprzecinkowymi. Podobnie jak w przypadku operatorów arytmetycznych można dla nich stosować zapis skrócony.

Operatory	**Opis działania**
|, &, ^	Operatory sumy bitowej (or), iloczynu bitowego (and) oraz bitowej różnicy symetrycznej (xor).
~	Jednoargumentowy operator negacji bitowej.
>>, <<	Operatory przesunięcia bitowego w prawo i w lewo. **Uwaga**: W języku C operatory te działają także dla liczb ujemnych, w szczególności przesunięcie w prawo liczby ujemnej daje w wyniku liczbę ujemną - najstarszy bit uzupełniany jest jedynką, zgodnie z kodem U2.

Operator &

Ilustracja działania

Operator bitowy & - przykład

$5 \;\&\; 3 = 00000101 \;\&\; 00000011 =$

Działanie operatora iloczynu bitowego &, tak jak wszystkich dwuargumentowych operatorów bitowych, polega na przeprowadzaniu operacji na parach bitów jego argumentów. W przypadku iloczynu to działanie można opisać następująco: *jeśli oba bity w parze są równe 1, to bit na tej samej pozycji w wyniku otrzymuje wartość 1, w przeciwnym przypadku nadawana mu jest wartość 0*. Jeżeli ten operator używany jest celem zbadania wartości poszczególnych bitów w zmiennej to takie działanie nazywa się *maskowaniem*. Jednym z argumentów w tym działaniu jest stała nazywana *maską*.

Operator &

Ilustracja działania

Operator bitowy & - przykład

$5 \;\&\; 3 = 00000101 \;\&\; 00000011 = 0$

Działanie operatora iloczynu bitowego &, tak jak wszystkich dwuargumentowych operatorów bitowych, polega na przeprowadzaniu operacji na parach bitów jego argumentów. W przypadku iloczynu to działanie można opisać następująco: *jeśli oba bity w parze są równe 1, to bit na tej samej pozycji w wyniku otrzymuje wartość 1, w przeciwnym przypadku nadawana mu jest wartość 0*. Jeżeli ten operator używany jest celem zbadania wartości poszczególnych bitów w zmiennej to takie działanie nazywa się *maskowaniem*. Jednym z argumentów w tym działaniu jest stała nazywana *maską*.

Operator &

Ilustracja działania

Operator bitowy & - przykład

$5 \;\&\; 3 = 00000101 \;\&\; 00000011 = 00$

Działanie operatora iloczynu bitowego &, tak jak wszystkich dwuargumentowych operatorów bitowych, polega na przeprowadzaniu operacji na parach bitów jego argumentów. W przypadku iloczynu to działanie można opisać następująco: *jeśli oba bity w parze są równe 1, to bit na tej samej pozycji w wyniku otrzymuje wartość 1, w przeciwnym przypadku nadawana mu jest wartość 0*. Jeżeli ten operator używany jest celem zbadania wartości poszczególnych bitów w zmiennej to takie działanie nazywa się *maskowaniem*. Jednym z argumentów w tym działaniu jest stała nazywana *maską*.

Operator &

Ilustracja działania

Operator bitowy & - przykład

$5 \;\&\; 3 = 00000101 \;\&\; 00000011 = 000$

Działanie operatora iloczynu bitowego &, tak jak wszystkich dwuargumentowych operatorów bitowych, polega na przeprowadzaniu operacji na parach bitów jego argumentów. W przypadku iloczynu to działanie można opisać następująco: *jeśli oba bity w parze są równe 1, to bit na tej samej pozycji w wyniku otrzymuje wartość 1, w przeciwnym przypadku nadawana mu jest wartość 0*. Jeżeli ten operator używany jest celem zbadania wartości poszczególnych bitów w zmiennej to takie działanie nazywa się *maskowaniem*. Jednym z argumentów w tym działaniu jest stała nazywana *maską*.

Operator &

Ilustracja działania

Operator bitowy & - przykład

$5 \;\&\; 3 = 00000101 \;\&\; 00000011 = 0000$

Działanie operatora iloczynu bitowego &, tak jak wszystkich dwuargumentowych operatorów bitowych, polega na przeprowadzaniu operacji na parach bitów jego argumentów. W przypadku iloczynu to działanie można opisać następująco: *jeśli oba bity w parze są równe 1, to bit na tej samej pozycji w wyniku otrzymuje wartość 1, w przeciwnym przypadku nadawana mu jest wartość 0*. Jeżeli ten operator używany jest celem zbadania wartości poszczególnych bitów w zmiennej to takie działanie nazywa się *maskowaniem*. Jednym z argumentów w tym działaniu jest stała nazywana *maską*.

Operator &

Ilustracja działania

Operator bitowy & - przykład

$5 \;\&\; 3 = 00000101 \;\&\; 00000011 = 00000$

Działanie operatora iloczynu bitowego &, tak jak wszystkich dwuargumentowych operatorów bitowych, polega na przeprowadzaniu operacji na parach bitów jego argumentów. W przypadku iloczynu to działanie można opisać następująco: *jeśli oba bity w parze są równe 1, to bit na tej samej pozycji w wyniku otrzymuje wartość 1, w przeciwnym przypadku nadawana mu jest wartość 0*. Jeżeli ten operator używany jest celem zbadania wartości poszczególnych bitów w zmiennej to takie działanie nazywa się *maskowaniem*. Jednym z argumentów w tym działaniu jest stała nazywana *maską*.

Operator &

Ilustracja działania

Operator bitowy & - przykład

$5 \;\&\; 3 = 00000101 \;\&\; 00000011 = 000000$**0**

Działanie operatora iloczynu bitowego &, tak jak wszystkich dwuargumentowych operatorów bitowych, polega na przeprowadzaniu operacji na parach bitów jego argumentów. W przypadku iloczynu to działanie można opisać następująco: *jeśli oba bity w parze są równe 1, to bit na tej samej pozycji w wyniku otrzymuje wartość 1, w przeciwnym przypadku nadawana mu jest wartość 0*. Jeżeli ten operator używany jest celem zbadania wartości poszczególnych bitów w zmiennej to takie działanie nazywa się *maskowaniem*. Jednym z argumentów w tym działaniu jest stała nazywana *maską*.

Operator &

Ilustracja działania

Operator bitowy & - przykład

$5 \;\&\; 3 = 00000101 \;\&\; 00000011 = 0000000$

Działanie operatora iloczynu bitowego &, tak jak wszystkich dwuargumentowych operatorów bitowych, polega na przeprowadzaniu operacji na parach bitów jego argumentów. W przypadku iloczynu to działanie można opisać następująco: *jeśli oba bity w parze są równe 1, to bit na tej samej pozycji w wyniku otrzymuje wartość 1, w przeciwnym przypadku nadawana mu jest wartość 0*. Jeżeli ten operator używany jest celem zbadania wartości poszczególnych bitów w zmiennej to takie działanie nazywa się *maskowaniem*. Jednym z argumentów w tym działaniu jest stała nazywana *maską*.

Operator &

Ilustracja działania

Operator bitowy & - przykład

$5 \;\&\; 3 = 00000101 \;\&\; 00000011 = 00000001$

Działanie operatora iloczynu bitowego &, tak jak wszystkich dwuargumentowych operatorów bitowych, polega na przeprowadzaniu operacji na parach bitów jego argumentów. W przypadku iloczynu to działanie można opisać następująco: *jeśli oba bity w parze są równe 1, to bit na tej samej pozycji w wyniku otrzymuje wartość 1, w przeciwnym przypadku nadawana mu jest wartość 0*. Jeżeli ten operator używany jest celem zbadania wartości poszczególnych bitów w zmiennej to takie działanie nazywa się *maskowaniem*. Jednym z argumentów w tym działaniu jest stała nazywana *maską*.

Operator &

Ilustracja działania

Operator bitowy & - przykład

$$5 \;\&\; 3 = 00000101 \;\&\; 00000011 = 00000001 = 1$$

Działanie operatora iloczynu bitowego &, tak jak wszystkich dwuargumentowych operatorów bitowych, polega na przeprowadzaniu operacji na parach bitów jego argumentów. W przypadku iloczynu to działanie można opisać następująco: *jeśli oba bity w parze są równe 1, to bit na tej samej pozycji w wyniku otrzymuje wartość 1, w przeciwnym przypadku nadawana mu jest wartość 0*. Jeżeli ten operator używany jest celem zbadania wartości poszczególnych bitów w zmiennej to takie działanie nazywa się *maskowaniem*. Jednym z argumentów w tym działaniu jest stała nazywana *maską*.

Operator |

Ilustracja działania

Operator bitowy | - przykład

$5 \mid 3 = 00000101 \mid 00000011 =$

Działanie operatora sumy bitowej | jest podobne do działania iloczynu bitowego, ale zasada wyznaczania wyniku jest trochę inna: *jeśli oba bity w parze bitów argumentów operatora mają wartość 0, to odpowiadający tej parze bit w wyniku ma wartość 0, w przeciwnym przypadku 1*.

Operator |

Ilustracja działania

Operator bitowy | - przykład

$5 \mid 3 = 00000101 \mid 00000011 = 0$

Działanie operatora sumy bitowej | jest podobne do działania iloczynu bitowego, ale zasada wyznaczania wyniku jest trochę inna: *jeśli oba bity w parze bitów argumentów operatora mają wartość 0, to odpowiadający tej parze bit w wyniku ma wartość 0, w przeciwnym przypadku 1*.

Operator |

Ilustracja działania

Operator bitowy | - przykład

$5 \mid 3 = 00000101 \mid 00000011 = 00$

Działanie operatora sumy bitowej | jest podobne do działania iloczynu bitowego, ale zasada wyznaczania wyniku jest trochę inna: *jeśli oba bity w parze bitów argumentów operatora mają wartość 0, to odpowiadający tej parze bit w wyniku ma wartość 0, w przeciwnym przypadku 1*.

Operator |

Ilustracja działania

Operator bitowy | - przykład

$5 \mid 3 = 00000101 \mid 00000011 = 000$

Działanie operatora sumy bitowej | jest podobne do działania iloczynu bitowego, ale zasada wyznaczania wyniku jest trochę inna: *jeśli oba bity w parze bitów argumentów operatora mają wartość 0, to odpowiadający tej parze bit w wyniku ma wartość 0, w przeciwnym przypadku 1*.

Operator |

Ilustracja działania

Operator bitowy | - przykład

$5 \mid 3 = 00000101 \mid 00000011 = 0000$

Działanie operatora sumy bitowej | jest podobne do działania iloczynu bitowego, ale zasada wyznaczania wyniku jest trochę inna: *jeśli oba bity w parze bitów argumentów operatora mają wartość 0, to odpowiadający tej parze bit w wyniku ma wartość 0, w przeciwnym przypadku 1*.

Operator |

Ilustracja działania

Operator bitowy | - przykład

$5 \mid 3 = 00000101 \mid 00000011 = 00000$

Działanie operatora sumy bitowej | jest podobne do działania iloczynu bitowego, ale zasada wyznaczania wyniku jest trochę inna: *jeśli oba bity w parze bitów argumentów operatora mają wartość 0, to odpowiadający tej parze bit w wyniku ma wartość 0, w przeciwnym przypadku 1*.

Operator |

Ilustracja działania

Operator bitowy | - przykład

$5 \mid 3 = 00000101 \mid 00000011 = 000001$

Działanie operatora sumy bitowej | jest podobne do działania iloczynu bitowego, ale zasada wyznaczania wyniku jest trochę inna: *jeśli oba bity w parze bitów argumentów operatora mają wartość 0, to odpowiadający tej parze bit w wyniku ma wartość 0, w przeciwnym przypadku 1*.

Operator |

Ilustracja działania

Operator bitowy | - przykład

$5 \mid 3 = 00000101 \mid 00000011 = 0000011$

Działanie operatora sumy bitowej | jest podobne do działania iloczynu bitowego, ale zasada wyznaczania wyniku jest trochę inna: *jeśli oba bity w parze bitów argumentów operatora mają wartość 0, to odpowiadający tej parze bit w wyniku ma wartość 0, w przeciwnym przypadku 1*.

Operator |

Ilustracja działania

Operator bitowy | - przykład

$5 \mid 3 = 00000101 \mid 00000011 = 00000111$

Działanie operatora sumy bitowej | jest podobne do działania iloczynu bitowego, ale zasada wyznaczania wyniku jest trochę inna: *jeśli oba bity w parze bitów argumentów operatora mają wartość 0, to odpowiadający tej parze bit w wyniku ma wartość 0, w przeciwnym przypadku 1*.

Operator |

Ilustracja działania

Operator bitowy | - przykład

$5 \mid 3 = 00000101 \mid 00000011 = 00000111 = 7$

Działanie operatora sumy bitowej | jest podobne do działania iloczynu bitowego, ale zasada wyznaczania wyniku jest trochę inna: *jeśli oba bity w parze bitów argumentów operatora mają wartość 0, to odpowiadający tej parze bit w wyniku ma wartość 0, w przeciwnym przypadku 1*.

Operator ^

Ilustracja działania

Operator bitowy ^ - przykład

$5 \text{ ^ } 3 = 00000101 \text{ ^ } 00000011 =$

Bitowa różnica symetryczna ^ działa tak samo jak pozostałe dwuargumentowe operatory, ale również różni się zasadą obliczania wyniku: *jeśli oba bity w parze bitów argumentów operatora mają różne wartości, to odpowiadający parze bit wyniku otrzymuje wartość 1, a w przeciwnym przypadku 0*.

Operator ^

Ilustracja działania

Operator bitowy ^ - przykład

$5 \text{ ^ } 3 = 00000101 \text{ ^ } 00000011 = 0$

Bitowa różnica symetryczna ^ działa tak samo jak pozostałe dwuargumentowe operatory, ale również różni się zasadą obliczania wyniku: *jeśli oba bity w parze bitów argumentów operatora mają różne wartości, to odpowiadający parze bit wyniku otrzymuje wartość 1, a w przeciwnym przypadku 0*.

Operator ^

Ilustracja działania

Operator bitowy ^ - przykład

$5 \text{ ^ } 3 = 00000101 \text{ ^ } 00000011 = 00$

Bitowa różnica symetryczna ^ działa tak samo jak pozostałe dwuargumentowe operatory, ale również różni się zasadą obliczania wyniku: *jeśli oba bity w parze bitów argumentów operatora mają różne wartości, to odpowiadający parze bit wyniku otrzymuje wartość 1, a w przeciwnym przypadku 0*.

Operator ^

Ilustracja działania

Operator bitowy ^ - przykład

$5 \text{ ^ } 3 = 00000101 \text{ ^ } 00000011 = 000$

Bitowa różnica symetryczna ^ działa tak samo jak pozostałe dwuargumentowe operatory, ale również różni się zasadą obliczania wyniku: *jeśli oba bity w parze bitów argumentów operatora mają różne wartości, to odpowiadający parze bit wyniku otrzymuje wartość 1, a w przeciwnym przypadku 0*.

Operator ^

Ilustracja działania

Operator bitowy ^ - przykład

$5 \text{ ^ } 3 = 00000101 \text{ ^ } 00000011 = 0000$

Bitowa różnica symetryczna ^ działa tak samo jak pozostałe dwuargumentowe operatory, ale również różni się zasadą obliczania wyniku: *jeśli oba bity w parze bitów argumentów operatora mają różne wartości, to odpowiadający parze bit wyniku otrzymuje wartość 1, a w przeciwnym przypadku 0*.

Operator ^

Ilustracja działania

Operator bitowy ^ - przykład

$5 \text{ ^ } 3 = 00000101 \text{ ^ } 00000011 = 00000$

Bitowa różnica symetryczna ^ działa tak samo jak pozostałe dwuargumentowe operatory, ale również różni się zasadą obliczania wyniku: *jeśli oba bity w parze bitów argumentów operatora mają różne wartości, to odpowiadający parze bit wyniku otrzymuje wartość 1, a w przeciwnym przypadku 0*.

Operator ^

Ilustracja działania

Operator bitowy ^ - przykład

$5 \text{ ^ } 3 = 00000101 \text{ ^ } 00000011 = 000001$

Bitowa różnica symetryczna ^ działa tak samo jak pozostałe dwuargumentowe operatory, ale również różni się zasadą obliczania wyniku: *jeśli oba bity w parze bitów argumentów operatora mają różne wartości, to odpowiadający parze bit wyniku otrzymuje wartość 1, a w przeciwnym przypadku 0*.

Operator ^

Ilustracja działania

Operator bitowy ^ - przykład

$5 \text{ ^ } 3 = 00000101 \text{ ^ } 00000011 = 0000011$

Bitowa różnica symetryczna ^ działa tak samo jak pozostałe dwuargumentowe operatory, ale również różni się zasadą obliczania wyniku: *jeśli oba bity w parze bitów argumentów operatora mają różne wartości, to odpowiadający parze bit wyniku otrzymuje wartość 1, a w przeciwnym przypadku 0*.

Operator ^

Ilustracja działania

Operator bitowy ^ - przykład

$5 \text{ ^ } 3 = 00000101 \text{ ^ } 00000011 = 00000110$

Bitowa różnica symetryczna ^ działa tak samo jak pozostałe dwuargumentowe operatory, ale również różni się zasadą obliczania wyniku: *jeśli oba bity w parze bitów argumentów operatora mają różne wartości, to odpowiadający parze bit wyniku otrzymuje wartość 1, a w przeciwnym przypadku 0*.

Operator ^

Ilustracja działania

Operator bitowy ^ - przykład

$5 \text{ ^ } 3 = 00000101 \text{ ^ } 00000011 = 00000110 = 6$

Bitowa różnica symetryczna ^ działa tak samo jak pozostałe dwuargumentowe operatory, ale również różni się zasadą obliczania wyniku: *jeśli oba bity w parze bitów argumentów operatora mają różne wartości, to odpowiadający parze bit wyniku otrzymuje wartość 1, a w przeciwnym przypadku 0*.

Operator ~

Ilustracja działania

Operator bitowy ~ - przykład

~ 5 = ~ 00000101 =

Ten operator jest jednoargumentowy. Jego działanie polega na zmianie wartości bitów argumentu na przeciwne.

Operator ~

Ilustracja działania

Operator bitowy ~ - przykład

~ 5 = ~ 00000101 = 1

Ten operator jest jednoargumentowy. Jego działanie polega na zmianie wartości bitów argumentu na przeciwne.

Operator ~

Ilustracja działania

Operator bitowy ~ - przykład

$\sim 5 = \sim 00000101 = 11$

Ten operator jest jednoargumentowy. Jego działanie polega na zmianie wartości bitów argumentu na przeciwne.

Operator ~

Ilustracja działania

Operator bitowy ~ - przykład

$\sim 5 = \sim 00000101 = 111$

Ten operator jest jednoargumentowy. Jego działanie polega na zmianie wartości bitów argumentu na przeciwne.

Operator ~

Ilustracja działania

Operator bitowy ~ - przykład

$\sim 5 = \sim 00000101 = 1111$

Ten operator jest jednoargumentowy. Jego działanie polega na zmianie wartości bitów argumentu na przeciwne.

Operator ~

Ilustracja działania

Operator bitowy ~ - przykład

~ 5 = ~ 00000101 = 11111

Ten operator jest jednoargumentowy. Jego działanie polega na zmianie wartości bitów argumentu na przeciwne.

Operator ~

Ilustracja działania

Operator bitowy ~ - przykład

$\sim 5 = \sim 00000101 = 111110$

Ten operator jest jednoargumentowy. Jego działanie polega na zmianie wartości bitów argumentu na przeciwne.

Operator ~

Ilustracja działania

Operator bitowy ~ - przykład

$\sim 5 = \sim 00000101 = 11111101$

Ten operator jest jednoargumentowy. Jego działanie polega na zmianie wartości bitów argumentu na przeciwne.

Operator ~

Ilustracja działania

Operator bitowy ~ - przykład

$\sim 5 = \sim 00000101 = 11111010$

Ten operator jest jednoargumentowy. Jego działanie polega na zmianie wartości bitów argumentu na przeciwne.

Operator ~

Ilustracja działania

Operator bitowy ~ - przykład

$\sim 5 = \sim 00000101 = 11111010 = 250$

Ten operator jest jednoargumentowy. Jego działanie polega na zmianie wartości bitów argumentu na przeciwne.

Operator <<

Ilustracja działania

Operator « - przykład

$5 << 2 = 00000101 << 2 = 00010100 = 20$

Operator << jest nazywany operatorem bitowego przesunięcia w lewo. Jego działanie polega na przesunięciu bitów w słowie bitowym (bajcie, dwóch bajtach itd.) w lewo o zadaną liczbę pozycji i odpowiada ono pomnożeniu lewego argumentu przez potęgę dwójki, tj. jeśli przesuwamy lewy argument o np. cztery miejsca, to mnożymy go przez 2^4. Bity z lewej strony, które „nie mieszczą" się w słowie są „zapominane", a bity z prawej są uzupełniane zerami.

Operator >>

Ilustracja działania

Operator » - przykład

$5 >> 2 = 00000101 >> 2 = 00000001 = 1$

Operator >> jest nazywany operatorem bitowego przesunięcia w prawo. Jego działanie polega na przesunięciu bitów w słowie bitowym (bajcie, dwóch bajtach itd.) w prawo o zadaną liczbę pozycji i odpowiada ono podzieleniu bez reszty lewego argumentu przez potęgę dwójki, tj. jeśli przesuwamy lewy argument o np. cztery miejsca, to dzielimy go bez reszty przez 2^4. Bity z prawej strony, które „nie mieszczą" się w słowie są „zapominane", a bity z lewej są uzupełniane zależnie od początkowej wartości najstarszego (skrajnie lewego) bitu. Jeśli przed rozpoczęciem operacji miał on wartość 1, to uzupełniane są jedynkami, w przeciwnym przypadku zerami.

Operatory bitowe

Podsumowanie

Ponieważ w zapisie niektóre operatory bitowe są podobne do operatorów logicznych, to warto zapamiętać regułę, którą wymyślił Bruce Eckel, aby wiedzieć jak je odróżniać: *„Bity są małe, więc do zapisu operatorów bitowych używany jest tylko jeden znak”*. W przypadku dwuarumentowych operatorów bitowych można stosować skrócony zapis, analogicznie jak w przypadku dwuargumentowych operatorów arytmetycznych.

Operator trójargumentowy

Operator trójargumentowy ma działanie podobne do instrukcji warunkowej, która będzie omawiana na następnym wykładzie, ale dodatkowo zwraca, tak jak pozostałe operatory wartość. Można go opisać następującym wzorcem

```
zmienna=warunek?pierwsze_wyrażenie:drugie_wyrażenie;
```

Jeśli warunek jest spełniony, to operator zwróci wartość wyrażenia za pytajnikiem, natomiast jeśli fałszywy, to zwróci wartość wyrażenia za dwukropkiem. Instrukcja przypisania może być pominięta i wówczas zwrócona wartość jest tracona. Tak możemy postąpić, jeśli zależy nam wyłącznie na wykonaniu jednego z wyrażeń. Jeśli operator przypisania jest jednak użyty, to zmienna stojąca po jego lewej stronie otrzymuje wartość zwróconą przez operator trójargumentowy.

```
int a=5, b=3, max;
int main(void)
{
    max=(a>b)?a:b;
    return 0;
}
```

Rzutowanie

Rzutowanie służy do zamiany bieżącego typu wartości na inny typ. Może ono być wykonywane niejawnie (przez kompilator, bez ingerencji programisty) lub jawny (jest nakazane przez programistę). Przykład konwersji niejawnej:

```
int a;
int main(void)
{
    a=12.3;
    return 0;
}
```

W tym przypadku pierwotny typ liczby `12.3`, czyli `double` zostanie zamieniony na `int`, przy czym część ułamkowa zostanie utracona. Z taką utratą informacji należy zawsze się liczyć przy konwertowaniu typu bardziej pojemnego na mniej pojemny. W drugą stronę to zjawisko nie zachodzi.

Rzutowanie

Jawna konwersja

W niektórych sytuacjach trzeba wprost określić w programie, że ma być wykonana konwersja. Można to wykonać używając operatora rzutowania, czyli umieszczając przed wartością, zmienną lub wyrażaniem nazwę typu docelowego w nawiasach okrągłych.

```
double a;
int x = 4, y = 3;
int main(void)
{
        a=(double)x/y;
        return 0;
}
```

W zamieszczonym przykładzie konwersji uległa wartość zmiennej `x`, dzięki temu uzyskany wynik jest prawidłowy, a nie zaokrąglony w dół do najbliższej zmiennej całkowitej.

Operator wyłuskania adresu

Operator wyłuskania adresu jest zapisywany przy pomocy pojedynczego znaku &, tak samo jak operator iloczynu bitowego. Kompilator rozróżnia te operatory na podstawie kontekstu ich użycia. Operator wyłuskania jest jednoargumentowy, a iloczynu bitowego dwuargumentowy. Operator wyłuskania zwraca adres zmiennej w pamięci, przed którą stoi. Taka informacja potrzebna jest między innymi funkcji `scanf()` używanej do wprowadzania danych z klawiatury.

Operator `sizeof`

Operator `sizeof` zwraca rozmiar zmiennej przed którą stoi. Zmienna ta może być umieszczona w nawiasach okrągłych (preferowany zapis), ale nie jest to wymagane. Zamiast zmiennej można użyć także bezpośrednio identyfikatora typu.

```
long unsigned int a,b,c;

int main(void)
{
    a=sizeof b;
    a=sizeof(c);
    return 0;
}
```

Kolejność działań

Kolejność wykonywania działań w wyrażeniach zapisanych w programie napisanym w języku C określona jest przez ustalone priorytety operatorów. Można ją zmieniać za pomocą nawiasów okrągłych. Kolejność wykonywania omówionych na tym wykładzie operatorów jest następująca: najpierw operatory inkrementacji i dekrementacji w wersji „post", a następnie „pre", potem wszystkie pozostałe jednoargumentowe (wliczając w to `sizeof` i rzutowanie), następnie operatory arytmetyczne, przy czym mnożenie, dzielenie i wyznaczanie reszty z dzielenia ma pierwszeństwo przez dodawaniem i odejmowaniem. Niższy priorytet mają oba operatory przesunięcia bitowego, następnie relacyjne, przy czym operatory „większe", „większe lub równe", „mniejsze", „mniejsze lub równe" są wykonywane przed operatorami „równe" i „różne". Następnie, kolejno wykonywane są operatory bitowe `&`, `^` i `|`, a potem logiczne w tej samej kolejności (za wyjątkiem różnicy symetrycznej, która nie istnieje). Na końcu wykonywany jest operator trójargumentowy i operator przypisania wraz z skróconą formą zapisu wcześniej wymienionych operatorów dwuargumentowych.

Podstawowe operacje wejścia i wyjścia

W tej części wykładu zostaną omówione funkcje `scanf()` i `printf()` z pliku nagłówkowego `stdio.h`, które służą do, odpowiednio, wprowadzania z klawiatury wartości do zmiennej i wypisywania wartości ze zmiennych na ekran. Są również inne funkcje umożliwiające komunikację z użytkownikiem i będą one omawiane na pozostałych wykładach.

Funkcja scanf()

Funkcja scanf() pozwala użytkownikowi wprowadzić wartość z klawiatury, którą zapisuje we wskazanej zmiennej.

```
#include<stdio.h>

int a;

int main(void)
{
    scanf("%d",&a);
    return 0;
}
```

Funkcja ta przyjmuje co najmniej dwa argumenty rozdzielone przecinkiem. Pierwszy to tzw. ciąg formatujący, umieszczony w cudzysłowie, który zawiera sekwencję konwertującą rozpoczynającą się znakiem %. Określa ona spodziewany typ danej wprowadzanej przez użytkownika z klawiatury. Drugi to adres zmiennej, w której ta dana ma być zapisana.

Funkcja `scanf()`

Najważniejsze sekwencje konwertujące

Poniższa tabela zawiera przykłady sekwencji konwertujących dla większości omówionych wcześniej typów danych.

Sekwencja konwertująca	**Opis**
`%d`	Liczba całkowita typu `int`.
`%ld`	Liczba całkowita typu `long int`.
`%hd`	Liczba całkowita typu `short int`.
`%hhd`	Liczba całkowita typu `char`.
`%u`	Liczba naturalna typu `unsigned int`.
`%lu`	Liczba naturalna typu `unsigned long int`.
`%hu`	Liczba naturalna typu `unsigned short int`.
`%hhu`	Liczba naturalna typu `unsigned char`.
`%c`	Znak.
`%f`	Liczba zmiennoprzecinkowa typu `float`.
`%lf`	Liczba zmiennoprzecinkowa typu `double`.

Funkcja scanf()

Wczytywanie pojedynczych znaków

Stosując funkcję scanf() aby odczytać kilkukrotnie pojedyncze znaki należy pamiętać, że klawisz Enter również pozostawia znak, który może zostać przeczytany przez kolejne wywołanie tej funkcji. Aby tego uniknąć należy użyć tej funkcji tak jak to podano w niżej zamieszczonym przykładzie.

```
#include<stdio.h>
char a,b;
int main(void)
{
        scanf("%c",&a);
        scanf(" %c",&b);
        return 0;
}
```

Rozwiązanie polega na dodaniu spacji między znakami ", a % w drugim ciągu formatującym.

Funkcja `printf()`

Funkcja `printf()` jest podobna w działaniu do funkcji `puts()` pod tym względem, że wypisuje umieszczony w cudzysłowie ciąg znaków na ekran. Jednakże ta funkcja nie przenosi kursora do nowego wiersza na ekranie. Ponadto w przeciwieństwie do `puts()` potrafi ona wypisywać na ekran wartości zmiennych i wyrażeń. Aby to wykonać programista musi umieścić w ciągu wypisywanych znaków sekwencję konwertującą podobną do tej dla funkcji `scanf()`, a za tym ciągiem, po przecinku, zmienną lub wyrażenie, którego wartość chce przekazać użytkownikowi programu. Funkcja `printf()` może wypisać kilka zmiennych różnych typów za pomocą jednego wywołania. Wystarczy tylko umieścić odpowiednią liczbę sekwencji konwertujących oraz w odpowiedniej kolejności rozdzielone przecinkami zmienne, które mają być wypisane.

Funkcja printf()

Przykład

```c
#include <stdio.h>

int a,b;

int main(void)
{
    puts("Podaj liczbę całkowitą");
    scanf("%d",&a);
    puts("Podaj drugą liczbę całkowitą");
    scanf("%d",&b);
    printf("%d & %d = %d\n",a,b,a&b);
    return 0;
}
```

Funkcja `printf()`

Sekwencje konwertujące

Sekwencja konwertująca	Opis
`%d`	Liczba całkowita typu `int`.
`%ld`	Liczba całkowita typu `long int`.
`%u`	Liczba naturalna typu `unsigned int`.
`%c`	Znak.
`%f`	Liczba zmiennoprzecinkowa typu `double`. Może zawierać dodatkowe informacje o formatowaniu np. `%.3f` - wartość będzie wypisana z dokładnością do trzech miejsc po przecinku.
`%lf`	Liczba zmiennoprzecinkowa typu `long double`. Podobnie jak wyżej, może zawierać dodatkowe informacje o formatowaniu.
`%e`, `%E`	Liczba zmiennoprzecinkowa typu `double` w notacji wykładniczej, np. `3e-9`, jeśli użyte jest pierwsze formatowanie, lub `3E-9` , jeśli drugie.
`%le`, `%lE`	Liczba zmiennoprzecinkowa typu `long double` w notacji wykładniczej, np. `3e-9`, jeśli użyte jest pierwsze formatowanie, lub `3E-9` , jeśli drugie.
`%x`, `%X`	Liczba naturalna zapisana w kodzie szesnastkowym, np. `a5`, jeśli użyte jest pierwsze formatowanie, lub `A5`, jeśli użyte jest drugie formatowanie.
`%o`	Liczba naturalna w kodzie ósemkowym.

Funkcja `printf()`

Znaki specjalne

Aby umieścić kursor na ekranie w następnym wierszu lub wypisać znak tabulatora należy użyć sekwencji sterujących, które składają się ze znaku lewego ukośnika (ang. *backslash*) i innego znaku. Wyjątkiem jest sekwencja sterująca umożliwiająca wypisanie znaku %. Ona rozpoczyna się od ...znaku %. Tabela poniżej zawiera opis niektórych z nich.

Znak sterujący	**Opis**
`\n`	Znak nowego wiersza.
`\r`	Znak początku wiersza.
`\\`	Znak \ (wypisanie na ekranie).
`\"`	Znak ".
`\t`	Tabulator
`%%`	Procent

Pytania

?

Podziękowania

Składam podziękowania dla dra inż. Grzegorza Łukawskiego i mgra inż. Leszka Ciopińskiego za udostępnienie materiałów, których fragmenty zostały wykorzystane w tym wykładzie.

KONIEC

Dziękuję Państwu za uwagę!