Język C i C++. Warunki, instrukcje wyboru

Język C nie narzuca stosowania specjalnego typu danych do warunków logicznych. Warunkiem w C może być dowolna liczba całkowita lub wskaźnik, przy czym wartość 0 (NULL) oznacza fałsz, a wartość różna od zera – prawdę.

Z prostych warunków logicznych można składać bardziej złożone zdania logiczne, używając operatorów logicznych.

&&	AND	Iloczyn logiczny. Przykład: if (a==0 && b==7) printf("komunikat");
||	OR	Suma logiczna. Przykład: if (a==999 || f()) printf("komunikat");
!	NOT	Negacja, czyli zaprzeczenie warunku logicznego. Np. if(!a) printf("komunikat"); Negacja prawdy daje w wyniku 0 (fałsz), negacja fałszu daje w wyniku 1 (prawdę).

Końcowy wynik (wartość zdania logicznego złożonego z wielu prostszych warunków) jest wyliczana od lewej do prawej (w takiej kolejności, w jakie czytamy zapisane wyrażenie). Jeśli w pewnym momencie jednoznacznie udaje się ustalić, jaka będzie końcowa wartość całego wyrażenia, to kolejne warunki nie są sprawdzane. Co to oznacza w praktyce? To, że w pewnych przypadkach dalsze instrukcje, z jakich składa się zdanie logiczne będą wykonywane, w innych zaś nie.

Funkcja **rand()**, której za chwilę użyjemy, wymaga dołączenia nagłówka #include <stdlib.h>. Ponadto, aby wartości były chociaż trochę losowe, należy wystartować generator liczb losowych z losową wartością początkową. Najłatwiej można to osiągnąć wykonując jako pierwszą instrukcję w programu **srand(time(NULL));.**

Weźmy następujący przykład:

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

int f()
{
     printf("BŁYSK!\n"); //wyzwolenie_lampy_błyskowej();
     return rand() % 2; //zwróć losowo 0 albo 1, czyli fałsz albo prawdę
}

int main()
{
     srand(time(NULL)); //co sekundę jest to inna wartość
     if ( 1 || f() || f() || f()|| f()|| f() )
          printf("warunek prawdziwy\n");
     else
          printf("warunek fałszywy\n");
}
```

Ten kod zadziała zupełnie inaczej, niż gdybyśmy zapisali: `if ( 0 || f() || f() || f()|| f()|| f() )`. Ile razy błyśnie lampa w obu przypadkach? Czy da się to w ogóle jednoznacznie określić? Przetestuj ten kod uruchamiając go wielokrotnie i obserwuj wyniki.

Wniosek: unikaj używania w warunkach takich instrukcji, które modyfikują wartości zmiennych lub mają na celu wywołanie innych działań w programie. W różnych scenariuszach wykonania programu może to doprowadzić do całkowicie losowego działania aplikacji!

Zadanie do realizacji: popraw powyższy kod w taki sposób, aby lampa błyskała zawsze 5 razy niezależnie od szczęścia i innych czynników zewnętrznych.

Skoro każda wartość różna od zera jest traktowana jak prawda, to prawdziwe są warunki: 1, 1234, zmienna_calkowita (przy założeniu, że zmienna ta ma wartość różną od zera).

Konsekwentnie, wszystkie wskaźniki, które nie są puste (nie są NULL) też odpowiadają wartości „prawda". Tylko 0 i NULL są fałszem.

Przykład:

```
FILE *f = fopen("nazwa.txt", "r");
if (f != NULL) { fscanf(f, "%d", &zmienna); fclose(f); }
```

Możemy uprościć powyższy warunek do postaci:

```
if (f) { fscanf(f, "%d", &zmienna); fclose(f); }
```

Warunek (f) będzie prawdziwy wtedy i tylko wtedy, gdy f będzie różne od NULL (nie będzie wskaźnikiem pustym.

Zdradliwe = i ==

Operator podstawienia (=) w języku C działa w szczególny sposób. Powoduje przypisanie pod zmienną po prawej stronie wartości wyliczonej z wyrażenia stojącego po lewej. Dodatkowo, całe wyrażenie podstawienia ma wartość równą temu, co zostało podstawione. Możliwe jest zatem użycie tej wartości do podstawienia pod kolejną zmienną.

Przykład:

```
int a = 7, b = 7, c = 7;
printf("%d\n", c=113);
printf("%d\n", c);
```

Skoro wyrażenie `(c=113)` ma wartość 113, to czemu by nie podstawić jego wartości pod zmienną b?

Mamy wtedy: `b = (c=113);` przy czym można pominąć nawiasy i ostatecznie otrzymamy

```
b = c = 113;
```

Skoro pod b coś podstawiamy (konkretnie będzie to 113), to ta instrukcja też zwraca wartość 113. Podstawmy ją pod **a**:

```
a = b = c = 113;
```

Jest to wygodny sposób nadawania tej samej wartości wielu zmiennym, zwłaszcza wtedy, gdy często zmieniamy zdanie i chcemy szybko zastąpić 113 czymś innym, na przykład 999:

```
a = b = c = 999;
```

Jest to wygodniejsze niż zamiana `a = 113; b = 113; c = 113;` na `a = 999; b = 999; c = 999;` .

Mniej przyjemne konsekwencje wystąpią wtedy, gdy popełnisz pomyłkę i w warunku użyjesz instrukcji podstawienia zamiast porównania:

```
int a = 0;
if (a==1) printf("prawda"); else printf("fałsz");
printf("\nzmienna a ma wartość %d\n", a);
```

Porównaj wyniki powyżej z następującymi:

```
if (a=1) printf("prawda"); else printf("fałsz");
printf("\nzmienna a ma wartość %d\n", a);
```

Współczesne kompilatory ostrzegają przed takim potencjalnym błędem, warto więc czytać komunikaty ostrzeżeń („warning”) w oknie kompilatora. Celowe użycie takich karkołomnych instrukcji (tzn. użycie instrukcji podstawienia = w warunku) jest możliwe i poprawne pod względem składniowym, ale zwykle przynosi więcej szkody niż pożytku.

Przykład (nie należy go naśladować):

```
int a = 1, b = 2;
if (a || (a=b)) printf("prawda"); else printf("fałsz");
printf("\na==%d, b==%d\n", a, b);
```

Zmień kod, uruchom i ponownie obejrzyj wyniki, tym razem niech a będzie miało początkową wartość 0:

```
int a = 0, b = 2;
```

Pytanie

Dlaczego tak jest?

Instrukcja wielokrotnego wyboru ?:

```
warunek ? wartość_gdy_prawda : wartość_gdy_fałsz;
```

Jeśli warunek jest prawdziwy, to wartością całego wyrażenia jest to, co stoi przed dwukropkiem, w przeciwnym razie tą wartością jest to, co stoi za dwukropkiem.

Jest to chętnie stosowana instrukcja, zwiększa zwięzłość kodu. Działa podobnie jak if..else tylko dodatkowo zwraca wynik jako wartość całego wyrażenia. Tę wartość można np. podstawić pod zmienną lub użyć jako argumentu **return** w funkcji.

Przykład: funkcja **abs(x)**, czyli wartość bezwzględna z x.

```
int abs(int x)
{
     if (x < 0)
          return -x;
     else
          return x;
}
```

albo

```
int abs(int x)
{
     int wynik;
     if (x < 0)
          wynik = -x;
     else
          wynik = x;
     return wynik;
}
```

Wersja bardziej zwięzła:

```
int abs(int x)
{
     return x < 0 ? -x : x;
}
```

Instrukcje ?: można zagnieżdżać, co bardzo ładnie widać na przykładzie funkcji sgn(x):

$$\operatorname{sgn}(x) = \begin{cases} +1 & dla\ x > 0 \\ 0 & dla\ x = 0 \\ -1 & dla\ x < 0 \end{cases}$$

```
int sgn(int x)
{
     if (x > 0) return +1;
     else
     if (x == 0) return 0;
     else
     if(x < 0) return -1;
}
```

Wiedząc, że po **return** funkcja kończy działanie i nie są wykonywane dalsze instrukcje, można ten kod uprościć:

```
int sgn(int x)
{
     if (x > 0) return +1;
     if (x < 0) return -1;
     return 0; //kiedy x>0 oraz x<0 były fałszem, to nie potrzeba
               //sprawdzać że x==0, bo to jest oczywiste.
}
```

Można też jeszcze krócej zapisać:

```
int sgn(int x)
{
     return x>0 ? +1 : (x<0 ? -1 : 0);
}
```

Niektóre mikroprocesory mają wbudowane rozkazy przyspieszające wykonanie **?:**, zaś niezależnie od tego zwykle ta instrukcja daje po kompilacji szybszy kod maszynowy.

Mniej oczywiste użycie ?:

Wskaźniki mogą być używane w wyrażeniach arytmetycznych (są przecież szczególnej postaci adresami, czyli liczbami, a na liczbach można wykonywać pewne obliczenia). Najczęściej używanym wskaźnikiem w C jest wskaźnik znakowy (char*), zatem użyjemy go w poniższym przykładzie.

Chcemy wyświetlić komunikat o tym, czy x jest parzyste czy nieparzyste. Można to zapisać „klasycznie” w taki sposób:

```
if (x % 2 != 0) //jeżeli reszta z dzielenia x przez 2 nie wynosi 0
     printf("Liczba %d jest nieparzysta.", x);
else
     printf("Liczba %d jest parzysta.", x);
```

Zauważ, że printf() to funkcja, która też zwraca wynik (jaki?). W zależności od tego, czy x jest parzyste czy nie, wywołujemy inną wersję printf(…).

Moglibyśmy zapisać to krócej:

```
x%2!=0 ? printf("Liczba %d jest nieparzysta.", x) : printf("Liczba %d jest parzysta.", x);
```

Wyniku całego wyrażenia nie musimy pod nic podstawiać, język C nie zmusza nas do tego.

Wyrażenia po lewej i prawej stronie dwukropka są bardzo podobne. W obu przypadkach jest to printf, pewien napis oraz zmienna x. Czym jest napis w języku C? to wskaźnik. A wskaźniki można dodawać, porównywać, zwracać jako wynik działania funkcji oraz… używać w wyrażeniach ?:.

```
printf(x%2 ? "Liczba %d jest nieparzysta.":"Liczba %d jest parzysta.", x);
```

Pierwszym argumentem printf musi być wskaźnik znakowy (char*). Takim wskaźnikiem jest napis w języku C, np. "abc" czy "Liczba %d jest nieparzysta.". W zależności od wyniku (x%2) posługujemy się wskaźnikiem z lewej lub prawej strony wyrażenia, a zyskujemy o jedno wywołanie funkcji printf() mniej.

Czy można jeszcze bardziej się streścić?

Zauważmy podobieństwo lewego i prawego napisu. Czasem posługujemy się słowem „parzysta”, czasem zaś „nieparzysta”. Słowa te można przedstawić jako napisy języka C. Można też, używając formatowanego stdout i funkcji printf, posłużyć się ciągiem „%s” aby wstawić w odpowiednie miejsce głównego napisu właściwe słowo. Szablon napisu wygląda wtedy tak:

```
"Liczba %d jest %s."
```

przy czym zamiast %d zostanie wyświetlona wartość x, a zamiast %s pojawi się słowo „parzysta” bądź „nieparzysta”.

```
printf("Liczba %d jest %s.", x, x%2 ? "nieparzysta" : "parzysta");
```

Różnica między komunikatami to zaledwie 3 litery (partykuła **nie**). W napisie "Liczba %d jest **nie**parzysta." czerwone „nie” czasem występuje, czasem zaś go nie ma. Kiedy go nie ma, to w tym miejscu można wstawić pusty napis (""), który w C zapisujemy jako dwa znaki cudzysłów, między którymi nic się nie znajduje.

```
printf("Liczba %d jest %sparzysta.",  x,  x%2 ? "nie" : "");
```

Zadanie treningowe

Posługując się instrukcją ?:, w jednej funkcji printf(…) zapisz wynik pomiaru napięcia `double U` w następujący sposób:

Jeśli U>=1.000, to wyświetl „Napięcie wynosi +x.xxx V”.

Jeśli U<1.000 i U>-1.000, to wyświetl „Napięcie wynosi xxx mV”.

Jeśli U <= -1.000, to wyświetl „Napięcie wynosi -x.xxx V”.

Wskazówka: użyj gotowej funkcji fabs(…), która zwróci wartość bezwzględną liczby zmiennoprzecinkowej, lub sam napisz taką funkcję modyfikując kod funkcji int abs(int x) z wcześniejszych stron tego dokumentu.

Instrukcja wielokrotnego wyboru switch..case..default

```
if(znak=='K' || znak=='k') koniec_swiata();
else
if(znak==13) odpal_jakas_duza_rakiete();
else
if(znak=='?') wezwij_na_pomoc_Arnolda();
```

Taki kod z czasem traci na czytelności, ale przede wszystkim ułatwia powstawanie błędów wynikających z powszechnego używania copy-paste. Chcemy, aby K lub k powodowały koniec świata, zaś U lub u pozwalały uratować świat. Jako scenariusz aplikacji nie jest to może zbyt ambitnyc przykład, ale na scenariusz filmu znakomicie się nadaje, oczywiście pod warunkiem, że zagra w nim Bruce Willis.

```
if(znak=='K' || znak=='k') koniec_swiata();
else
if(znak=='K' || znak=='k') uratuj_swiat();
else
nudne_dialogi();
```

Gdyby Bruce zagrał według powyższego scenariusza, świat mógłby długo czekać na ratunek, a bardziej prawdopodobne jest, że wcześniej nastąpiłby jego koniec. Dlaczego?

Wystąpiła literówka w scenariuszu, który w założeniu miał wyglądać następująco:

```
if(znak=='K' || znak=='k') koniec_swiata();
else
if(znak=='U' || znak=='u') uratuj_swiat();
else
nudne_dialogi();
```

Moglibyśmy pominąć ‘else’, ale wtedy kod wykonywałby się dłużej oraz stałby się jeszcze mniej czytelny. Literówki mogłyby być jeszcze bardziej zaskakująco nieprzewidywalne w skutkach.

```
if(znak=='K' || znak=='k') koniec_swiata();
if(znak=='U' || znak=='u') uratuj_swiat();
if(znak!='K'  && znak!='k' && znak!='U' && znak!='u') nudne_dialogi();
```

Autorom scenariuszy nic już nie pomoże, ale programistom przyda się niezastąpiona w takich sytuacjach pomoc, jaką daje język C w postaci instrukcji switch z dodatkami.

```
switch (znak)
{
     case 'K': case 'k':
          koniec_swiata();
          break;
     case 'U': case 'u':
          uratuj_swiat();
          break;
     default:
          nudne_dialogi();
}
```

Zadanie

Poeksperymentuj z kodem: spróbuj wstawić ‘K’ i/lub ‘k’ zamiast ‘U’; spróbuj wyrzucić jedną lub obie instrukcje break.

1. Co będzie, kiedy spróbujesz na samym początku instrukcji switch zdefiniować zmienną lokalną? A jak zachowa się kod, gdy taka zmienna zostanie zadeklarowana miedzy case a break?
2. Czy instrukcje pomiędzy break a case (np. za pierwszym break, ale przed drugim case) kiedykolwiek się wykonają?
3. Czy sekcja default może być umieszczona w innym miejscu (np. przed case ‘K’)? Jak to wpłynie na działanie kodu?

W języku C switch..case ma ogromne możliwości, ale jest bardzo wrażliwy na zgubienie słowa break. Tak się pechowo składa, ze pominięcie break nie jest błędem składniowym i kompilator nie zgłosi nam tego problemu. Pomijanie break bardzo rzadko się wykorzystuje w praktyce i zwykle jest to błąd a nie celowe działanie programisty. Nowsze języki (np. C#) traktują już takie przypadki jako błąd.