

1. What's the simplest way to test whether a number is a power of 2 in C++?

a) Wy tłumacz, w jaki sposób działa poniższy algorytm:

$n \& (n - 1) == 0$

n – zmienna typu `int` przechowująca testowaną liczbę

By zrozumieć działanie algorytmu należy wytłumaczyć czym jest używany znak `&` - jest to operator „bitowego i”, czyli np. $x \& y$, spowoduje przetłumaczenie wartości z typu `int` na dwójkowy a następnie dla każdego miejsca bitowego w liczbie zostanie przeprowadzona logiczna operacja „i”. Zastosowanie takiej operacji dla danego miejsca w liczbie zwróci 0 jeżeli cyfry będą na tych miejscach sobie różne, a 1 gdy będą równe. Tak więc, dla $x = 4$ (dwójkowo: 100), $y = 3$ (dwójkowo 011), $x \& y = 100 \wedge 011 = 000 = 0$ (typ `int`).

A dla $x = 4$ (dwójkowo: 100), $y = 5$ (dwójkowo: 101), $x \& y = 100 \wedge 101 = 100 = 4$ (typ `int`).

Dodatkowo wiemy, że w systemie dwójkowym potęga liczby 2 zawsze będzie przedstawiona jako 1 i same zera, a liczba o jeden od niej mniejsza będzie przedstawiona jako 0 i same jedynki.

I właśnie to jest w algorytmie sprawdzane: jeżeli „bitowe i” będzie równe zero, to oznacza, że zadana liczba jest potęgą dwójki – zostanie zwrócone 1 – prawda (ponieważ na pierwszym od lewej miejscu znajdzie się 1 a resztą miejsc będą zera), w innym przypadku zostanie zwrócona wartość odpowiedzialna za fałsz - 0.

b) Jak działa poniższe rozwiązanie i czy jest ono prawidłowe?:

$!!num \& !(num \& (num - 1))$

num – zmienna typu `int` przechowująca testowaną liczbę

Rozwiązanie jest bardzo podobne do tego z podpunktu a (opiera się na porównywaniu wartości kolejnych miejsc bitowych). Jednakże kilkakrotnie zastosowano znak zaprzeczenia - `!`.

Jeżeli spojrzymy na prawą stronę od pierwszego znaku „bitowego i”, to mamy do czynienia z zaprzeczeniem algorytmu z podpunktu a - $!(num \& (num - 1))$, które w przypadku liczby będącej potęgą dwójki zwróci 1, a w innym przypadku zwróci 0 (logiczna wartość zaprzeczenia dowolnej liczby innej niż 0).

Teraz wróćmy do: $!!num$ – zaprzeczenie zaprzeczenia dowolnej liczby innej niż 0 zwróci 1 (ponieważ zaprzeczeniem dowolnej liczby innej niż 0 jest 0, a kolejne zaprzeczenie „odwróci” 0 na 1).

I na sam koniec lewa strona zostanie porównana bitowo z prawą – w przypadku liczby będącej potęgą dwójki wynikiem będzie 1, w innym przypadku 0.

To rozwiązanie działa nawet lepiej niż to z podpunktu a, ponieważ w przypadku sprawdzenia liczby 0, da dobry rezultat (algorytm z podpunktu a, nie da dobrego

rezultatu). Jednakże mam wątpliwości co do jego poprawności, ponieważ wynikiem algorytmu nie jest logiczna wartość (prawda/fałsz – 1/0) tylko wartość liczbowa 1/0 – to taki błąd stylistyczny w moim mniemaniu.

c) Rozwiązanie:

```
v && !(v & (v - 1))
```

v – zmienna typu unsigned int przechowująca testowaną liczbę

Jest bardzo podobne do rozwiązania z podp. b, lecz działa na liczbach bez znaku (unsigned) – dzięki czemu dla liczby 0 zwróci dobry wynik (dzięki liczbom unsigned nie występuje błąd z podp. a). Oczywiście nie widzimy podwójnego zaprzeczenia pierwszego wyrazu, ponieważ w tym rozwiązaniu widać użycie operatora logicznego „i” - && - przyrównuje ono wartości logiczne obu stron i dodatkowo pozbywa się błędu stylistycznego z podpunktu b. Gdyż zwraca wartość logiczną zamiast liczbowej.

d) Rozwiązanie:

```
1 == __builtin_popcount(n)
```

n – zmienna typu int przechowująca testowaną liczbę

Rozwiązanie to jest genialne w swojej prostocie, lecz nie może być nazwane uniwersalnym, ponieważ korzysta z wbudowanego nagłówka kompilatora GCC, a jak dobrze wiemy, nie każdy programista korzysta z kompilatora GCC.

Powyższe rozwiązanie zlicza ilość jedynek w zapisie binarnym zadanej liczby, a jak już wcześniej ustaliliśmy – tylko potęgi liczby 2 mają jedną 1 w zapisie binarnym. Zatem przyrównanie wyniku wbudowanej funkcji GCC do 1 da pożądany wynik – 1 – prawda w przypadku gdy sprawdzana liczba jest potęgą 2 i 0 – fałsz w przypadku innej liczby.

e) Pominąłem (bo nie było wymagane zgodnie z treścią zadania)

f) Rozwiązanie: `v && !(v & (v - 1))`, jest rozwiązaniem uniwersalnym (jego działanie nie zależy od kompilatora), nie daje false-positive wyników i działa na największym zakresie liczb. Najważniejsze jest jednak pamiętać, by działać na liczbach unsigned – bezznakowych.

2. Count bits 1 on an integer as fast as GCC `__builtin_popcount(int)`

a) Metoda przytoczona przez pytającego (`n >= 0`):

```
int countBit1Fast(int n)
{
    int c = 0;
    for (; n; ++c)
        n &= n - 1;
    return c;
}
```

Działa następująco:

Funkcja `countBit1Fast` przyjmuje liczbę jako zmienną `int n`.

W pętli wykonującej się do czasu aż `n` nie będzie zerem, wykonuje się operacja przyrównania do siebie bitowo liczb `n` oraz `n-1`.

Tam, gdzie na tych samych pozycjach w obu liczbach znajduje się 1, w wyniku tego działania również będzie 1, w przypadku każdej innej konfiguracji wystąpi 0.

Po każdym takim przyrównaniu, które w wyniku nie da nam 0, zmienna c oznaczająca ilość jedynek w zapisie liczby zostaje powiększona o 1 (domyślnie c = 0 – na wypadek gdyby podana przez nas liczba była równa zero).

Zmienna c jest równa ilości jedynek w binarnym zapisie liczby przez nas podanej.

b) Pomińmy (bo nie było wymagane zgodnie z treścią zadania)

c) Rozwiązanie:

```
int countBit1Fast(uint32_t n) {  
    n = (n & 0x55555555u) + ((n >> 1) & 0x55555555u);  
    n = (n & 0x33333333u) + ((n >> 2) & 0x33333333u);  
    n = (n & 0x0f0f0f0fu) + ((n >> 4) & 0x0f0f0f0fu);  
    n = (n & 0x00ff00ffu) + ((n >> 8) & 0x00ff00ffu);  
    n = (n & 0x0000ffffu) + ((n >> 16) & 0x0000ffffu);  
    return n;  
}
```

Działa następująco:

Funkcja countBit1Fast przyjmuje liczbę jako zmienną unsigned 32-bit int n.

A następnie pięciokrotnie przeprowadza operacje przyrównywania wartości, pięciokrotnie przesunięcia bitowego w prawą stronę (>>), pięciokrotnie sumuje wartości oraz 10 krotnie porównuje bitowo („bitowe i”).

By zrozumieć działanie powyższego należy wiedzieć, że:

0x oznacza liczbę w zapisie szesnastkowym (heksadecymalnym);

Przesunięcie bitowe w prawo przesuwają każdą wartość w zapisie binarnym liczby o jedno miejsce w prawo, a tą na miejscu pierwszym od prawej na miejsce pierwsze od lewej. Zatem dla $n = 100101$, $n >> 1$ da wynik $n = 110010$;

Dodatkowo liczby heksadecymalne wyżej tworzą swego rodzaju wzorce próbkowania liczb (które przyda się w przypadku porównywania bitowego):

$$(55555555)_{16} = (01010101010101010101010101010101)_2$$

$$(33333333)_{16} = (00110011001100110011001100110011)_2$$

$$(0f0f0f0f)_{16} = (1111000011110000111100001111)_2$$

$$(00ff00ff)_{16} = (111111110000000011111111)_2$$

$$(0000ffff)_{16} = (1111111111111111)_2$$

Tak więc pierwsze przyrównanie porówna bitowo wskazaną przez nas liczbę z pierwszym wzorcem (ma to na celu „przechwycenie” jedynek na miejscach nieparzystych liczby w systemie binarnym) a następnie zsumuję otrzymaną liczbę z liczbą przez nas wskazaną o bitowym przesunięciu 1 porównaną ze wzorcem. To zliczy nam jedynki występujące na miejscach nieparzystych (jednak są to jedynki miejsc parzystych liczby przez nas wpisanej – do tego potrzebne nam było bitowe przesunięcie w prawo).

Następnie operujemy na nowej wartości n – to już nie jest przez nas wpisana liczba, teraz n jest sumą dwóch liczb otrzymanych po przekształceniach pierwszego stopnia.

Tu dokonujemy takich samych działań jak wyżej, tym razem na podstawie wzorca który kończy się ...00110011. Dzięki temu wzorcowi możemy porównać bitowo miejsca 1,2,5,6,9,10 itd. a następnie dodać do otrzymanej liczby liczbę wynikową przesunięcia o dwa i porównania bitowego.

Ważne jest zrozumienie, że jeżeli w którymkolwiek kroku porównanie bitowe przesunięcia da liczbę 0, to n będzie równe liczbie jedynek we wpisanej przez nas liczbie.

Algorytm działa dobrze, ponieważ pierwsza linijka sprawdza występowanie jedynek na dwóch pierwszych miejscach z użyciem wzorca okresowego 01 i połowicznego przesunięcia (czyli o 1). Wynikiem takiego działania może być 0, 1 lub 2.

Druga linijka sprawdza cztery miejsca, wzorcem okresowym 0011 i przesunięciem równym połowie długości wzorca, czyli 2.

Trzecia linijka sprawdza 8 pierwszych miejsc, czwarta 16, piąta 32. W zależności od wielkości sprawdzanej liczby, wcześniej lub później otrzymamy liczbę jedynek w zapisie binarnym liczby.