C

MANIPULACJE BITAMI

Karol Kuczmarski „Xion”
karol.kuczmarski@gmail.com

Lepiej uczyć się rzeczy zbytecznych niż żadnych.
Seneka

Od czasu sławetnego stwierdzenia, że 640 kB będzie po wsze czasy wystarczającą ilością pamięci, minęło już ponad ćwierć wieku. W informatyce taki okres to już całe eony – nic dziwnego więc, że dzisiaj operujemy już pojemnościami będącymi nie tylko tysiąc, ale nawet kilkanaście czy kilkaset tysięcy razy większymi od tej „ostatecznej”. Cóż, proroctwa rzadko się sprawdzają :)
Wobec takich ogromnych ilości pojedynczy **bit** – jak wiemy, najmniejsza jednostka informacji – może wydawać się nieskończenie mały i przez to zupełnie nieznaczący. Co mianowicie możemy zapisać przy pomocy pojedynczego bitu lub ich grupki – bajtu, słowa?… Okazuje się, że całkiem sporo.

Manipulacje pojedynczymi bitami czy zespołami wciąż są bowiem przydatną umiejętnością. Wśród ich zastosowań można wymienić tak ważne zagadnienia jak:
- odczyt i zapis plików w formatach binarnych
- implementacja protokołów wielu popularnych aplikacji sieciowych
- współpraca programu z większością bibliotek programistycznych

W tym dodatku przedstawiam zatem pewne techniki operowania na tych niewielkich porcjach informacji – bitach i ich ciągach, czyli **słowach**. Na początku więc przyjrzymy się podstawowym działaniom na ciągach bitów, jakie oferuje większość języków programowania (w tym oczywiście C++). Następnie spróbujemy przy ich pomocy skontruować procedury pozwalające na dostęp do dowolnego zakresu bitów w słowie; dzięki temu nabierzesz wprawy w opisywanych tu operacjach. Wreszcie przyjrzymy się pewnym prawdziwym zastosowaniom takiej zabawy, przede wszystkich technice **flag bitowych**. Na koniec zerkniemy jeszcze na kilka przydatnych „sztuczek” realizowanych przy pomocy opisanych wcześniej operacji.
Temat jako taki wydaje się więc bardzo niskopoziomowy, lecz wcale nie oznacza to, że musi on być trudny. Prawdopodobnie jest on po prostu zbyt często pomijany, a bez jego znajomości nawet wyedukowany programista może być bezradny wobec kodu zawierającego instrukcje manipulacji bitami. Celem tego dodatku jest zatem uchronienie cię przed takimi sytuacjami :)

Operacje bitowe

Rozpoczniemy od dość wnikliwego rzutu oka na najważniejsze elementarne operacje bitowe, z jakich zwykle się korzysta. Naturalnie, zależy nam szczególnie na tych, które występują w C++ w postaci operatorów, choć pod tym względem wszystkie najważniejsze języki programowania są do siebie bardzo podobne.

Wśród tych najprostszych działań wyróżnimy sobie dwa typy: funkcje logiczne oraz przesunięcia bitowe.

Funkcje logiczne

Przypomnijmy, co w ogóle rozumiemy pod pojęciem pojedynczego **bitu**. Zazwyczaj mówi się, że jest to najmniejsza jednostka informacji, mogąca przyjmować tylko jeden z dwóch stanów: 0 lub 1. Zero w tej sytuacji utożsamiamy z logicznym fałszem, a jedynkę – z prawdą. Taka konwencja pozwala nam stosować do bitów spójniki znane z logiki, na przykład koniunkcję czy alternatywę. Możemy więc określić, ile wynosi $0 \wedge 1$, $1 \vee 1$, itd. Ale to nie jeszcze nie wszystko. Zastosowanie funkcji logicznych wobec pojedynczych bitów pozwala nam także powiedzieć, jak dana funkcja działa dla ciągów bitów (słów bitowych). Oznacza to po prostu zastosowanie jej do kolejnych bitów, np.:

	1	0	0	1	0	1	1	0
$\wedge$	1	0	1	0	1	1	0	1
=	1	0	0	0	0	1	0	0

Tabela 1. Koniunkcja dwóch ciągów bitów

Głównie na takich właśnie ciągach działają operatory bitowe w C++ i innych jezykach programowania. Reprezentują one rzecz jasna pewne funkcje logiczne, którym przyjrzymy się po kolei[1].

Bardzo podobny przegląd jest częścia Dodatku B, *Reprezentacja danych w pamięci*. Tutaj jednak skoncentrujemy się raczej na tym, jak omawiane funkcje pozwalają nam zmieniać wartości bitów zamiast na ich znaczeniu w logice.

Negacja

Prawdopodobnie najprostszą funkcją jest negacja (czyli zaprzeczenie), gdyż jest ona operacją jednoargumentową. Dla danego bitu wynikiem jest bit do niego **przeciwny** – a zatem jedynka zmienia się w zero, zaś w zero w jedynkę. Dla ciągu bitów jest to więc analogicznie zamiana wszystkich zer w jedynki i wszystkich jedynek w zera.
Negację oznacza się na wiele sposobów: tyldą (~) lub znakiem $\neg$ przed negowanym wyrażeniem, a także apostrofem ` za nim. Czasami też można spotkać poziomą kreską nad argumentem. Naturalnie we wszystkich przypadkach funkcja działa tak samo :) A ponieważ programujemy w C++, będziemy tu używali symbolu operatora negacji z tego języka, czyli ~.

b	***~b***
0	1
1	0

Tabela 2. Tabelka prawdy dla działania negacji bitowej

Negacja zmienia zatem wartość bitu na przeciwną. Zastosowanie jej dwa razy powinno nam więc dać wyjściową wartość – **jest to prawo podwójnego zaprzeczenia**, prezentujące się następująco:

~~b = b

Jak zobaczymy, działanie negacji będzie dla nas szczególnie przydatne w połączeniu z przesunięciami bitowymi. W sumie jednak trudno mówić o zastosowaniach tej operacji – po prostu stosujemy ją tam, gdzie jest konieczna :D

[1] W tym podrozdziale literką *b* będę oznaczał dowolny bit.

Suma

Pozostałe interesująca nas funkcje są już dwuargumentowe. Jedną z nich jest **alternatywa** (**suma**), którą można interpretować jako pewnego rodzaju dodawanie. Zastrzeżenie 'pewnego rodzaju' jest całkiem na miejscu, jako że zaraz poznamy inny sposób dodawania bitów.
Logicznym oznaczeniem alternatywy jest $\vee$, ale w przypadku działań na bitach częściej używamy symbolu `|` - odpowiada on rzecz jasna operatorowi z C++ i języków o podobnej składni. Czasem używamy też słowa *or* ('lub') ze względu na spójnik logiczny, jakiemu odpowiada alternatywa.

Jak zatem działa alternatywa tudzież suma bitowa? Dla dwóch bitów wynikiem jest zero **tylko wtedy**, gdy **oba bity** są zerami. W przypadku, gdy któryś z nich jest jedynką, rezultatem zawsze będzie jeden. W szczególności będzie tak również wtedy, kiedy zarówno pierwszy jak i drugi bit są jedynkami – wynikiem też będzie jeden. Właśnie to odróżnia alternatywę od opisanego dalej drugiego sposobu dodawania bitów.
Dla ciągów bitów opisana operacja ta jest oczywiście wykonywana dla każdej odpowiadającej sobie pary, zaś wynik jest złożeniem tych wszystkich rezultatów.
Nie zaszkodzi jeszcze podkreślić, że działanie sumy jest przemienne – niezależnie od kolejności argumentów wynik będzie więc taki sam.

a	***b***	***a | b***
0	0	0
0	1	1
1	1	1
1	0	1

Tabela 3. Tabelka wyników działania sumy bitowej

Wymienone wyżej własności pozwalają nam w łatwy sposób ustawiać stan bitów na 1. Z powyższego opisu wynika bowiem, iż:

```
b | 1 = 1
b | 0 = b
```

Te dwie tożsamości będą nam bardzo pomocne w dostępie do wybranego zakresu bitów w słowie.

Iloczyn

Drugim działaniem jest **iloczyn**, zwany też **koniunkcją**. W kontekście wartości bitów interpretujemy go jako mnożenie – działa bowiem tak, jak „normalny" iloczyn liczb 0 i 1. Jako operator logiczny koniunkcja jest natomiast równoważna spójnikowi 'i' (ang. *and*), oznaczanemu przez $\wedge$. W C++ jako odpowiedni operator bitowy dla iloczynu używany jest symbol `&`.

Przewidywanie wyników zastosowania tego działania nie jest trudne, gdy wiemy, że pełni ono rolę iloczynu. Istotnie, dla dwóch bitów wynikiem jest jedynka **wyłącznie wówczas**, gdy **oba bity** są jedynkami. W innym przypadku, kiedy którykolwiek bit jest zerem (a zwłaszcza oba), rezultatem też będzie zero. Jest to w sumie dość oczywiste – pomnożenie czegokolwiek przez zero powinno dać właśnie zero. Bity nie są tu żadnym wyjątkiem :)

a	***b***	***a & b***
0	0	0
0	1	0
1	1	1

a	*b*	*a & b*
1	0	0

Tabela 4. Tabelka wyników działania iloczynu bitowego

Podobnie jak suma, iloczyn jest przydatnym narzędziem do modyfikacji wartości bitów. Zauważmy mianowicie, że wobec powyższych własności mamy:

b & 0 = 0
b & 1 = b

Iloczyn pozwala zatem zerować stan wybranych bitów, nie ruszając innych. Niedługo zobaczymy to w praktyce.

Suma modulo 2

Dwa bity możemy dodać w jeszcze jeden sposób. Przedstawiona na początku zwykła suma tak naprawdę bowiem nie jest wcale ich „normalnym" sposobem dodawania. W przypadku sumowania dwóch jedynek wynikiem arytmetycznym byłoby oczywiście 2 (binarnie 10). Przyjęcie w tym przypadku rezultatu 1 zapewnia nam jednak zgodność z logicznym pojęciem alternatywy oraz – jak widzieliśmy – jest przydatne do ustawiania bitów.

Jednak możemy przecież ustalić sumę dwóch jedynek na 0 i ma to więcej sensu niż się może początkowo wydawać. Zachowujemy bowiem cyfrę z „prawdziwego" arytmetycznego wyniku, a jedynie ignorujemy tzw. **przeniesienie** (ang. *carry*) na następny bit. W ten sposób powstaje działanie **sumy modulo 2**, zwanej tej **różnicą symetryczną**. W logice istnieje mniej znane, acz równoważne działanie alternatywy wykluczającej. Nie ma ono jakiegoś oczywistego spójnika w języku naturalnym[2], jak również matematycznego symbolu – używa się zwykle $\oplus$ lub $\veebar$.
W języku angielskim spotyka się za to często słówko *xor*, od *eXclusive OR* – 'alternatywa wykluczająca'. Pod tym względem nie zawodzi nas także język C++, gdzie działanie bitowej sumy modulo 2 ma swój operator ^.

Jak zatem funkcjonuje to działanie? Różni się ono od poprzedniej sumy tylko na jednym miejscu, lecz łatwiej je zapamiętać przy pomocy innej własności. Otóż różnica symetryczna dwóch bitów daje jedynkę **tylko wtedy**, gdy bity te są **różne**. Dla **równych** bitów wynikiem działania sumy modulo 2 jest zawsze zero.
Jeszcze inaczej możemy to interpretować przy pomocy pojęcia reszty z dzielenia – stąd zresztą operacja ta bierze jedną ze swoich nazw. Mianowicie, suma modulo 2 jest właśnie… sumą modulo 2 :), tzn. zwykłym (arytmetycznym) dodaniem wartości dwóch bitów i wyciągnięcia z wyniku reszty z dzielenia przez 2:

$$a \oplus b = (a + b) \bmod 2$$

a	*b*	*a* $\oplus$ *b*
0	0	0
0	1	1
1	1	0
1	0	1

Tabela 5. Tabelka wyników działania sumy modulo 2

[2] Niekiedy podawany jest spójnik *albo* (w znaczeniu „albo, albo").

Możesz się zastanawiać – do czego mogłoby nam się przydać takie dziwne działanie? Prawdopodobnie nie widać tego na pierwszy rzut oka, ale stosują się do niego takie oto dwie tożsamości (w razie czego można to sprawdzić :D):

```
b ^ 0 = b
b ^ 1 = ~b
```

Operacja sumy modulo 2 pozwala zatem przeprowadzać negację wybranych bitów przy jednoczesnym zachowaniu wartości pozostałych.

Pozostałe funkcje dwuargumentowe

W dalszych przykładach będziemy korzystali niemal wyłącznie z powyższych trzech operacji dwuargumentowych (i jednej jednoargumentowej). Nie zaszkodzi jednak przyjrzeć się także innym, a nawet… wszystkim :)

NAND i NOR

Istnieją dwie ciekawe dwuargumentowe funkcje logiczne, które nie są tak znane jak przytoczone wyżej, a pewnym sensie są „równie dobre" jak one. Te dwie funkcje są oznaczane najczęściej swoimi angielskimi nazwami NAND i NOR – od *Negative AND* i *Negative OR*.
Określenia te wskazują, że rzeczone funkcje są związane z operacjami sumy oraz ilocznu, a ponadto w grę wchodzi jeszcze negacja. Faktycznie, są to po prostu **zanegowane** operacje *and* i *or*:

```
nand(x, y) = ~(x & y)
nor(x, y) = ~(x | y)
```

Specjalność tych dwóch formuł polega na tym, że teoretycznie używając **tylko i wyłącznie** jednej z nich można zapisać **wszystkie** funkcje logiczne działające na **dowolnie wielu** argumentach. Dlaczego tak jest? Wiemy rzecz jasna, że jest to możliwe przy korzystaniu z oddzielnej operacji negacji oraz zwykłej sumy i iloczynu. Otóż okazuje się, że te „normalne" działania można uzyskać posługując się tylko NANDem lub tylko NORem:

```
~x = nand(x, x) = nor(x, x)
x | y = nand(nand(x, x), nand(y, y)) = nor(nor(x, y), nor(x, y))
x & y = nand(nand(x, y), nand(x, y)) = nor(nor(x, x), nor(x, x))
```

Nie wydaje się to szczególnie efektowne (ani efektywne), jako że suma i iloczyn wymagają zastosowania swych zanegowanych odpowiedników aż trzykrotnie. Okazuje się jednak, że ta nieefektywność może być pozorna. W układach elektronicznych składających się z **bramek logicznych** (czyli części symulujących działanie omawianych tutaj funkcji) zastosowanie NAND i NOR daje często lepsze rezultaty niż bramek odpowiadających „zwykłym" funkcjom logicznym negacji, sumy i iloczynu. Zarówno koszt układu, jak i jego opóźnienia – mimo zwiększonej ogólnej liczby bramek – są wtedy bowiem mniejsze.

Wszystkie funkcje logiczne

Poznaliśmy więc już pięć dwuargumentowych operacji logicznych. Skoro tak dobrze nam idzie, to może zapytajmy od razu: ile jest ich w sumie? Okazuje się, że nie jest to wcale trudne pytanie. Możemy odpowiedzieć nawet na ogólniejsze; otóż:

Istnieje dokładnie 2^{2^n} funkcji logicznych biorących *n* argumentów i zwracających jeden rezultat.

Liczba ta jest więc całkiem spora dla dużych *n*, ale my zajmujemy się tylko dwuargumentowymi operacjami, zatem nie powinno nam grozić przepełnienie :) Dla *n* = 2 wartość wynosi oczywiście 16 – mamy zatem 16 różnych funkcji logicznych biorących na wejściu dwa argumenty.
Mimo wszystko może się to wydawać sporo – jeśli znamy tylko pięć, to gdzie są pozostałe? Spieszę z odpowiedzią, że całkiem niedaleko – przedstawia je mianowicie poniższa tabela:

formuła	**x:** **y:**	**0** **0**	**0** **1**	**1** **0**	**1** **1**	***nazwa***
0		0	0	0	0	zero (fałsz)
x & y		0	0	0	1	koniunkcja (iloczyn)
x & ~y		0	0	1	0	zaprzeczenie implikacji
x		0	0	1	1	pierwszy argument
~x & y		0	1	0	0	
y		0	1	0	1	drugi argument
x ^ y		0	1	1	0	alternatywa wykluczająca (suma modulo 2)
x | y		0	1	1	1	alternatywa (suma)
~(x | y)		1	0	0	0	NOR
x = y		1	0	0	1	równoważność
~y		1	0	1	0	zaprzeczenie drugiego argumentu
x | ~y		1	0	1	1	
~x		1	1	0	0	zaprzeczenie pierwszego argumentu
~x | y		1	1	0	1	implikacja
~(x & y)		1	1	1	0	NAND
1		1	1	1	1	jedynka (prawda)

Tabela 6. Wszystkie dwuargumentowe funkcje logiczne

Widać w niej, że niemal każda możliwość daje się interpretować w logiczny lub bitowy sposób. W praktyce korzystamy jednak z funkcji omówionych wcześniej, jako że ich interpretacja jest najbardziej oczywista (a ponadto będą one dla nas najbardziej przydatne).

Przesunięcia bitowe

Możliwość działania na ciągach słowach bitowych jak na ciągach wartości logicznych będzie nam niezbędna dalej. Są jednak inne ważne operacje na bitach, które nie mają logicznego rodowodu. Chodzi tu właśnie o **przesunięcia bitowe** (ang. *bitwise shift*).

Operacje te działają na pojedynczym słowie i manipulują jego bitami w bardziej mechaniczny sposób. Dobrze oddaje to ich nazwa – działania te zajmują się niczym innym, jak właśnie przesuwaniem bitów. Możliwa jest ich translacja zarówno w prawo (w stonę mniej znaczących bitów), jak i w lewo (w stronę bardziej znaczących). Ponadto tego przesuwania można dokonywać na parę różnych sposobów, różniących się postępowaniem z bitami „wypadającymi" oraz pojawiającymi się na „wolnych" miejscach. Z bardziej formalnego punktu widzenia przesunięcią są funkcjami przyjmującymi dwa parametry. Pierwszym z nich jest oczywiście ciąg bitów, który chcemy odpowiednio potraktować; drugim jest **offset**, czyli liczba określająca ilość miejsc, o które chcemy przesunąć bity w słowie. Tych dwóch argumentów wymaga każdy rodzaj przesunięcia bitowego.

Skoro wiemy już mniej więcej, jaka jest ogólna idea tego typu operacji, przystąpimy teraz do bliższego przyjrzenia się im.

Zwykłe przesunięcia

Oto najprostszy, a jednocześnie chyba najbardziej przydatny sposób realizacji przesunięcia. Jednak on tak naturalny, że nie ma nawet żadnej specjalnej nazwy :) Jest też jedynym, któremu odpowiadają operatory dostępne w języku C++.

Działanie zwyczajnego przesunięcia bitowego nie jest zbyt skomplikowane. Wydaje się ono bowiem zupełnie nie przejmować jakimikolwiek efektami ubocznymi - mianowicie:

- bity, które w wyniku przesuwania „wypadają", są po prostu pomijane i wszelki ślad po nich ginie
- nowe bity, pojawiające się z drugiej strony słowa, są zawsze zerami

Efekty te ilustruje poniższa tabelka, przedstawiająca bajt przed i po przesunięciu go o 2 miejsca w lewo:

	1	1	0	0	1	0	1	1
<<								2
=	0	0	1	0	1	1	0	0

Tabela 7. Przykład ilustrujący działanie zwykłego przesunięcia bitowego

Linią przerywaną otoczone są dwa skrajnie lewe bity – one zostaną „zepchnięte" w wyniku przesuwania i znikną ze słowa. Po drugiej stronie zrobi się natomiast wolne miejsce i tam pojawiają się zera – oznaczone tutaj ciągłą obwódką. Jak można się domyślić, analogicznie wyglądałoby przesuwanie w prawo - z tym że bity wypadające znalazłyby się rzecz jasna po prawej stronie, zaś nowe zera po lewej.

Wspomniałem już, że w języku C++ istnieją operatory realizujące opisywane tutaj przesunięcia. Są to << i >>, zajmujące się – odpowiednio: przesuwaniem bitów w lewo i prawo. Każdy z nich wymaga dwóch argumentów – słowa bitowego (w postaci zmiennej typu liczbowego) oraz określenia ilości miejsc, o które chcemy przesuwać. Wygląda to na przykład w taki oto sposób:

```
BYTE byFoo = 0x3F;   // binarnie: 00111111
byFoo = byFoo >> 4;  // wynik: 0x03, binarnie: 00000011
```

Jednym z zastosowań przesunięcia bitowego jest otrzymywanie spójnych obszarów zer i jedynek określonej długości. Niedługo zobaczymy, że w ten sposób można łatwo odwoływać się do wybranych zakresów bitów w słowie.

W C++ istnieje pewna niejednoznaczność odnosząca się do operacji przesunięcia bitowego w prawo (>>). Przy jego stosowaniu tak naprawdę są bowiem możliwe dwa warianty: lewa (bardziej znacząca część słowa) może być zawsze wypełniana zerami albo wypełniana swoim najbardziej znaczącym bitem. Tenże bit w przypadku wartości całkowitych oznacza **znak liczby** (0 dla dodatniej, 1 dla ujemnej) i dlatego stosowanie tego drugiego sposobu wypełniania nazywa się często **przesunięciem ze znakiem** (ang. *signed bitwise shift*) lub rozszerzeniem znaku (ang. *sign extension*).
Problem polega na tym, że faktyczny sposób działania operatora >> w C++ może być różny dla różnych kompilatorów, jako że standard dopuszcza tu dowolność. Dokładniej mówiąc, jeżeli przesuwamy w prawo wartość ze znakiem (np. typu `int`), to zależnie od kompilatora możemy otrzymać albo wypełnienie zerami, albo bitem znaku. Na szczęście większość porządnych kompilatorów (w tym Visual C++) wybiera rozsądniejsze rozwiązanie, tj. rozszerzenie znaku. Tym niemniej jest to kolejny przykład na to, że szczegóły działania operatorów w C++ lubią być zależne od typów swoich argumentów – tak jest też np. w przypadku dzielenia. Co ciekawe, ta niskopoziomowa kwestia jest jednoznacznie rozwiązana w języku Java, uchodzącym za język wyższego poziomu niż C++.

W dalszej części, gdy będziemy używali przesunięcia w prawo, będzie to zawsze wariant wypełniający lewą stronę zerami. Aby uniknąć wszelkich niejednoznaczności zalecam więc wykonywanie operacji bitowych **wyłącznie na wartościach bez znaku**.

Przesunięcia cykliczne

Inny rodzaj przesunięcia gwarantuje nam, że żadne bity nie zostaną stracone i żadne zera się nie pojawią. Tak działa **przesunięcie cykliczne** (ang. *bitwise rotation*). Polega ono na „zawinięciu" słowa w ten sposób, iż:

- bity wypadające z jednej strony automatycznie pojawiają się po drugiej stronie słowa
- ich kolejność w obu miejscach jest taka sama

Jeśli nie brzmi to zbyt przekonująco, warto przyjrzeć poniższemu przykładowi. Ponownie przesuwamy bajt o 2 miejsca w lewo, lecz tym razem czynimy to cyklicznie:

	1	0	0	0	1	0	1	1
RL								2
=	0	0	1	0	1	1	1	0

Tabela 8. Przykład ilustrujący działanie przesunięcia cyklicznego

Otoczone linią przerywaną bity nie są już teraz tracone. Zamiast tego pojawiają się z drugiej strony słowa – i to w tym samym porządku, w jakim były pierwotnie. Faktycznie więc można wyobrazić sobie działanie przesunięcia cyklicznego jak zwykłego, tyle że działającego na „zawiniętym" słowie.

Niestety w języku C++ nie ma operatorów realizujących ten rodzaj przesunięcia. W przykładzie powyżej użyłem symbolu RL, gdyż tak w asemblerze nazywa się instrukcja dokonująca rotacji w lewo (i analogicznie RR dla rotacji w prawo). Jest jednak dobra wiadomość – można w całkiem prosty sposób zaimplementować przesunięcie cykliczne przy pomocy zwykłego i paru innych operacji na bitach. W następnym podrozdziale zobaczymy, jak można by to zrobić.

Dostęp do bitów

Pierwszym i szerokim zastosowaniem dla poznanych przed chwilą operacji bitowych będzie dla nas szeroko rozumiany dostęp do bitów w słowie. Inaczej mówiąc, zobaczymy w jaki można odczytywać wartości zarówno pojedynczych bitów, jak i ich zakresów; zajmiemy się także ich modyfikacją. Łącznie pozwoli nam na przykład odczytywać gęsto upakowane dane, których używa wiele bibliotek programistycznych, formatów plików oraz protokołów aplikacji internetowych.

Nieco wyjaśnień

Zanim to się stanie, musimy sobie jednak wyjaśnić kilka rzeczy :) Chodzi tu między innymi o kwestie nieco techniczne, związane ze sposobem zapisu omawianych dalej algorytmów. Ponadto powiemy sobie krótko o znaczeniu masek bitowych i długości używanych słów.

Konwencja

Wynikowa postać danej operacji będzie bardzo często zaledwie pojedynczą linijką kodu. W języku C zapisywalibyśmy ją zazwyczaj w postaci makrodefinicji. W C++ także możemy to robić, ale istnieją inne (w większości lepsze sposoby), jak chociażby użycie

zwykłej czy też szablonowej funkcji. Zapisywanie gotowego algorytmu w każdej możliwej postaci byłoby głównie stratą miejsca, jako że byłyby one do siebie bardzo podobne. Zamiast tego użyjemy zatem czegoś w rodzaju pseudokodu, który będzie jednak bardzo podobny do normalnego kodu C(++). Widzieliśmy to już w poprzednim podrozdziale. Dzięki takiemu rozwiązaniu unikniemy wszystkich niezwiązanych z tematem elementów składniowych, typu nagłówki funkcji czy dyrektywy `#define`. Dodanie ich nie będzie jednak żadnych problemem dla programisty znającego język C++ - czyli dla ciebie tym bardziej :)

Dla porządku biędzemy też używali pewnych oznaczeń, dzięki czemu możliwe będzie łatwe odróżnienie danych wejściowych (zazwyczaj ciągów bitów), wyjściowych i ewentualnych zmiennych pomocniczych. Wedle tych zasad:

- *x* będzie argumentem wejściowym w postaci słowa (ciągu bitów), który wystąpi właściwie w każdym algorytmie. Jeżeli zajdzie potrzeba użycia większej liczby argumentów będących słowami, kolejne oznaczane będą przez *y* i *z*. Tyle już powinno wystarczyć :)
- *k*, *n*, *p* będą argumentami wejściowymi w postaci liczb całkowitych bez znaku. Oczywiście nie wszędzie będą nam potrzebne wszystkie trzy
- *r* będzie reprezentował rezultat działania naszego algorytmu. Zazwyczaj będzie to ciąg bitów
- przez *m* będziemy oznaczali tzw. maskę – specjalny ciąg bitów używany przy kilku opisywanych dalej operacjach. O tym, czym są maski bitowe powiemy sobie dosłownie za chwilę
- *s* oznaczać będzie rozmiar słów (ilość bitów), na których przeprowadzamy wszystkie nasze działania
- inne litery (np. *t*, *a*, *b*) posłużą nam za pomocnicze zmienne „lokalne”

Muszę jeszcze przyznać, że właściwie całą tą konwencję podaję głównie dla formalności. Poszczególne algorytmy są bowiem okraszone na tyle obszernymi opisami, iż prawdopodobnie z ich zrozumieniem nie powinno być żadnego problemu – przynajmniej w teorii :) Jak jest naprawdę? O tym wypadałoby się czym prędzej przekonać.

Maski bitowe

Bardzo duża ilość technik manipulacji bitami opiera się na niezwykle ciekawym pomyśle wykorzystania masek bitowych. Wiele spośród algorytmów prezentowanych dalej nie mogłoby się bez nich obyć. Należałoby zatem najpierw wyjaśnić sobie, czym właściwie te maski są. Otóż:

Maska bitowa to specjalnie spreparowany ciąg bitów, który w połączeniu z pewną operacją logiczną pozwala modyfikować bity podanego słowa w określony sposób.

Jako że nie brzmi to zbyt konkretnie, przyjrzyjmy się raczej typowemu schematowi użycia maski do osiągnięcia zamierzonego efektu. Mianowicie, w algorytmie korzystającym z maski bitowej możemy często wyróżnić dwie jego części. Są to kolejno:

- **utworzenie maski**, dokonywane np. poprzez odpowiednio zastosowane przesunięcie bitowe
- **wykonanie operacji logicznej** (sumy, iloczynu, sumy modulo 2, itd.) na masce oraz wejściowym argumencie

Oba te etapy są rzecz jasna ze sobą **powiązane**: konkretna maska bitowa jest przygotowywana do wykorzystania ze ściśle określoną operacją bitową. Dopiero łącznie pozwala to osiągnąć efekt, o jaki nam chodzi. Nie ma czegoś takiego jak maska „uniwersalna”: jeżeli określony ciąg był przygotowany do zastosowania z iloczynem, to posłużenie się nim jako argumentem dla sumy da oczywiście zupełnie inne, błędne wyniki - a przynajmniej nie takie, jakich byśmy oczekiwali.

Najbardziej interesujące pytanie brzmi naturalnie: jak odpowiednio dobrać te dwa elementy – maskę i operację? Jak się zapewne domyślasz, tym właśnie będziemy zajmowali się dalej. Pewnych wskazówek dostarczają tutaj własności omówionych na początku operacji logicznych – zwłaszcza ich zachowanie się w zależności od tego, czy jeden z argumentów jest jedynką czy zerem. Z tych właściwości będziemy wkrótce intensywnie korzystali.

Kwestia długości słowa

Ostatnia sprawa dotyczy długości (tudzież rozmiaru) używanych przez nas słów bitowych. Do tej pory dosyć skrzętnie lawirowałem wokół tego tematu, jednak musimy wyklarować sobie to zagadnienie zanim będziemy mogli przejść dalej.

Przez długość słowa rozumiemy ilość bitów wchodzących w jego skład. Jest to niekiedy bardzo istotna wartość i dlatego zarezerwowaliśmy sobie specjalne dla niej oznaczenie (*s*) – gdyby była nam ona kiedykolwiek potrzebna w naszych algorytmach. Mimo to **nie chcielibyśmy** jej używać. Mamy ku temu przynajmniej dwa istotne powody.
Do tej pory obywaliśmy się przecież całkiem dobrze bez korzystania z pojęcia długości słowa. Nie oznacza to bynajmniej, że dla nas słowo jest ciągiem bitów o dowolnej długości - raczej o takiej długości, którą nie warto sobie zaprzątać głowy :) Duża część algorytmów działających na bitach może się bowiem obejść bez znajomości rozmiaru słów i właśnie takie algorytmy będziemy preferowali. Dlaczego?…

- na różnych platformach rozmiar **słowa maszynowego** (czyli domyślnie używanego) bywa różny. Między innymi to jest właśnie powodem, dla którego mówimy o systemach 16-, 32- czy 64-bitowych. Opieranie się na określonej długości słów (zapisanej na sztywno) czyniłoby nasze programy nieprzenośnymi
- jeżeli zamiast tego zdecydowaliśmy się pobierać każdorazowo rozmiary przekazywanych argumentów, otrzymalibyśmy rozwiązanie kłopotliwe i często nieefektywne – czyli o cechach dokładnie przeciwnych do tych, jakie oczekujemy po algorytmach bitowych

Istnieją naturalnie problemy, których nie da się rozwiązać bez jawnego użycia rozmiaru słowa bitowego. Ponadto wypadałoby, abyśmy zawsze wiedzieli, jakiej wielkości są ciągi bitów używane przez nas w danej chwili. W C++ sprowadza się to głównie do odpowiedniej kontroli typów – tak, żeby za naszymi plecami nie były dokonywane żadne niejawne konwersje.
Najprostsze i najczęściej stosowane rozwiązanie polega po prostu na używaniu tylko i wyłącznie jednego typu liczbowego do przechowywania ciągów bitów. Musimy przy tym pamiętać, że:

Do przedstawianych tutaj operacji bitowych powinniśmy używać wyłącznie typów **całkowitych bez znaku**.

Najlogiczniejsze wydaje się więc użycie `unsigned int`. W odniesieniu do wartości tego typu wszystkie działania bitowe będą bowiem funkcjonowały dokładnie tak, jak tego oczekujemy.

Działania na bitach

Czas przejść w końcu do omawiania konkretnych algorytmów czy – jak kto woli – „sztuczek" działających na słowach bitowych. Rozpoczniemy od potencjalnie pożądanych i uniwersalnych czynności, a mianowicie od dostępu do określonych zakresów bitów w słowie. Przez dostep rozumiemy tu zarówno odczytanie tychże bitów (czyli ich ekstrakcję w postaci mniejszego słowa), jak i hurtowe ustawianie danego zakresu na daną wartość (0 lub 1).

Dostęp do zakresu bitów

Zakres bitów będzie u nas reprezentowany przez parę liczb, oznaczanych *k* i *n*, gdzie:
- *k* jest indeksem pierwszego bitu z zakresu, począwszy od najmniej znaczacego
- *n* jest ilością bitów w zakresie

Nasz zakres będzie zatem miał długość *n* i obejmował wszystkie bity począwszy od *k* aż do *k*+*n*-1 włącznie. Tak więc dla przykładowego poniższego słowa:

0 0 1 0 1 1 1 0

zakres (3, 2) rozpoczyna się od bitu o indeksie 3 (czyli *czwartego* bitu – liczymy od zera) i ma długość 2. Wygląda więc tak:

* * * 0 1 * * *

(gwiazdki oznaczają pozostałe, nienależące do zakresu bity).

Dalej będziemy zatem zajmować się kilkoma prostymi operacjami na takich właśnie zakresach bitów. Oczywiście wszystko co tam powiemy w gruncie rzeczy można zastosować do pojedynczych bitów. Chcąc więc np. odczytać wartość *k*-tego bitu, stosujemy podane niżej metody dla zakresu (*k*, 1).

Odczyt

Zaczniemy od odczytania wybranego zakresu bitów. Co to jednakowoż znaczy? Otóż mając dane słowo bitowe z wyróżnionym zakresem:

...	*k*+*n*-1	*k*+*n*-2	...	*k*+1	*k*	...

chcielibyśmy otrzymać takie, w którym:
- nasz zakres jest dosunięty maksymalnie do prawej strony (strony najmniej znaczących bitów)
- reszta bitów (poza zakresem) jest zerowa

0	...	0	*k*+*n*-1	*k*+*n*-2	...	*k*+1	*k*

Dlaczego właśnie tak? Jeśli bowiem w tym zakresie jest jakaś interesująca nas wartość liczbowa, takie słowo pozwoli nam odczytać ją właśnie jako liczbę. W przypadku pojedynczego bitu będziemy mogli ją interpretować jako 0 lub 1, czyli np. jako logiczną prawdę lub fałsz. Niezależnie od naszych intencji, takie wyczyszczenie interesujących nas informacji ze „śmieci” (pozostałych bitów) będzie więc najprawdopodobniej przydatne.

Jak więc to zrobić?… Jest to mianowicie proste ćwiczenie z przesuwania bitów. Prawdopodobnie spełnienie pierwszego warunku (dosunięcia do prawej strony) nie będzie problemem – wystarczy początkowe słowo przesunąć o *k* miejsc w prawo:

```
t = x >> k
```

Problem polega na tym, że po lewej stronie nadal będziemy mieli „śmieci”, a nie zera:

*	...	*	*k*+*n*-1	*k*+*n*-2	...	*k*+1	*k*

Istnieją dwa sposoby na rozwiązanie tego problemu. Pierwszy, łatwiejszy, wymaga jawnego podania długości słowa. Drugi będzie pierwszym przykładem użycia masek bitowych.

Z użyciem długości słowa

Jak wiemy, zwykłe przesuwanie (a takiego chcemy użyć) wyrzuca nam wszystkie „niemieszczące" się bity ze słowa, a nowopowstające pozycje są zapełnione zerami. Te własności możemy wykorzystać do pozbycia się niechcianej części naszego słowa – po prostu ją z niego **wypchniemy**.

Pomysł jest całego algorytmu jest raczej prosty i zawiera sięw dwóch krokach:

- najpierw dosuwamy nasz zakres maksymalnie w lewo – czyli tak, aby jego ostatni bit (k+n-1) stał się na chwilę najbardziej znaczącym bitem w słowie
- następnie przesuwamy go maksymalnie w prawo, w pożądaną pozycję

W pierwszym etapie przesuwanie w lewo wykasuje nam wszystkie bity znajdujące się na lewo od zakresu. W drugim etapie kasujemy bity po prawej stronie zakresu oraz korzystamy z faktu, iż zwykłe przesuwanie w prawo wypełni nam też miejsca po lewej stronie zakresu zerami. Ostatecznie więc otrzymamy słowo w szukanej postaci: z wybranym zakresem bitów po prawej stronie i resztą wypełnioną zerami.

Kluczowe pytanie brzmi więc: o ile przesuwać? Pomyślmy zatem, ile bitów znajduje się początkowo po lewej stronie zakresu. Są to bity począwszy od tego o indeksie k+n aż do końca słowa. „Do końca" – znaczy w tym przypadku do najbardziej znaczącego bitu, tego o indeksie s-1, gdzie s to długość słowa. W sumie więc tych bitów jest (s-k-n) (polecam sprawdzenie na jakimś przykładzie):

```
a = x << (s – k – n)
```

k+n-1	k+n-2	...	k+1	k	*	...	0

Tak więc przedstawia się krok pierwszy. Krok drugi zakłada dosunięcie naszego zakresu do prawej strony; po drodze musimy zatem przejść niemal całe słowo. Niemal – gdyż należy przesunąć się dokładnie o (s-n) miejsc:

```
r = a >> (s – n)
```

0	...	0	k+n-1	k+n-2	...	k+1	k

Ostatecznie całe zadanie wykonujemy za pomocą dwóch przesunięć i tylko przesunięć:

```
// odczyt zakresu bitów począwszy od k i długości n w słowie x
a = x << (s – k – n)
r = a >> (s – n)
```

Ponieważ jest to nasz pierwszy algorytm, zaprezentuję jeszcze sposób, w jaki zapisalibyśmy go w postaci makrodefinicji języka C. Po skondensowaniu do jednego wyrażenia wyglądałaby ona tak:

```
#define UBITS (8*sizeof(unsigned))
#define BITS_RANGE(x,k,n) (((x) << ((UBITS – (k) – (n))) >> (UBITS – (n)))
```

Podobnie (w postaci jednej linijki) można będzie zapisać właściwie każdy z pokazanych dalej algorytmów. Naturalnie, zastosowanie bardziej nowoczesnego rozwiązania w rodzaju funkcji inline skutkować będzie z pewnością większą czytelności kodu (brak tysiąca nawiasów!).

Jak zatem widać, odczyt zakresu bitów nie byo to specjalnie trudny. Nasz sposób używa jednak długości słowa, co – jak opisywałem wcześniej – nie jest specjalnie zadowalające.

Dlatego też zobaczymy teraz inne rozwiązanie, nie korzystające jawnie z tej wartości. Zamiast tego posługuje się ono odpowiednią maską bitową.

Z użyciem maski bitowej

Naszą maskę dla operacji odczytu chcemy przygotować w ten sposób, aby (po użyciu odpowiedniej operacji bitowej):

- zachowała bity z wybranego zakresu w niezmienionym stanie
- wyzerowała wszystkie pozostałe bity

Jeżeli bowiem tak zrobimy, pozostanie nam jedynie użyć operacji, o której wspomnieliśmy na początku: przesunąć powstałe słowo o *k* miejsc w prawo, by uzyskać je w żądanej pozycji. Pozostaje więc „tylko" skonstruować odpowiednią maskę i wybrać do niej właściwy operator logiczny.

A tak naprawdę zaczniemy właśnie od tego operatora. Spośród znanych nam dwuargumentowych działań bitowych chcemy bowiem znaleźć takie, które potrafi zerować bity lub zachowywać je w stanie niezmienionym – zależnie od tego, czy jego drugim argumentem jest 0 czy 1. Rzut oka na początek rozdziału pozwala stwierdzić, że właściwym działaniem jest **iloczyn** ze względu na poniższe własności:

b & 0 = 0
b & 1 = b

Maska użyta w naszym przypadku powinna zatem:

- zawierać jedynki w miejscach odpowiadających odczytywanemu zakresowi bitów
- mieć zera na wszystkich pozostałych miejscach

0 ... 0	1 1 ... 1 1	0 ... 0

Uzyskanie tego efektu nie jest trudne, acz wymaga znajomości ogólnej techniki konstruowania masek bitowych. Otóż w większości przypadków korzysta się tutaj na przemian z dwóch rodzajów operacji: **przesunięć bitowych** oraz **negacji**. Postępujemy tak, gdyż pozwala nam to zmieniać znaczenie pojawiających się przy przesuwaniu zer. Ich negacja zmienia je bowiem w jedynki i sprawia, że kolejne przesunięcia (tworzące nowe zera) będą wnosiły coś nowego do struktury maski. W ten sposób rozmieszczamy bity maski w sposób zapewniający właściwe zachowanie po potraktowaniu jej (i wejściowego słowa) wybraną operacją logiczną.
Nasuwa się jednak pytanie: od czego zacząć budowanie maski?… Właściwie jedynymi stałymi, których możemy na pewno bezpiecznie użyć jako początkowych są 0 i 1. Przesuwanie bitów zera nic nam jednak nie daje: nowopowstałe wypełnienia będą także zerami. Przesuwanie jedynki dałoby nam z kolei maskę operującą co najwyżej na pojedynczych bitach. Co zatem zrobić w ogólnym przypadku?

Otóż – **zanegować zero**. Negacja słowa zawierającego same zera da nam słowo składające się z samych tylko jedynek. I to jest bardzo dobry punkt startowy! Wykonanie dowolnego przesunięcia stworzy nam bowiem obszar zer o wybranej długości, czyli w końcu coś nietrywialnego. Korzystając dalej z negacji w celu odwrócenia znaczenia bitów możemy tworzyć pełnowartościowe maski do praktycznych zastosowań.
A zatem zaczynamy od nie-zera:

m = ~0

Sprytne, prawda? W C++ powinniśmy jeszcze zwrócić uwagę na typ owego zera (czyli stałej dosłownej). Jako że posługujemy się liczbami typu `unsigned`, powinno to być zero tego właśnie typu: `0u`.

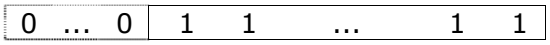
Pora w końcu wrócić do właściwego problemu: tworzenia maski dla odczytu zakresu bitów. Zaczniemy od wyodrębnienia w niej obszaru odpowiadającego długością naszemu zakresowi bitów:

```
m = m << n
```

Będzie to oczywiście obszar zer, powstałych podczas przesuwania. Nie uśmiecha nam się to za bardzo, gdyż chcieliśmy przecież, aby w masce naszemu zakresowi odpowiadały jedynki. Nic prostszego – wystarczy zastosować negację:

```
m = ~m
```

W tej chwili maska wygląda zatem następująco:



Mamy w niej ciąg jedynek odpowiedniej długości umieszczony wśród zer. Obecnie jest on tylko na złej pozycji: rozpoczyna się bitem o indeksie 0, a chcielibyśmy umieścić go w słowie począwszy od *k*-tego bitu. A więc… przesuńmy ponownie - tym razem o *k* miejsc:

```
m = m << k
```

I tak dostaniemy dokładnie taką maskę, jaka jest nam potrzebna.
Przypomnijmy, że chcieliśmy wobec niej zastosować iloczyn bitowy, dzięki czemu żądany zakres bitów w słowie zostałby wyodrębniony. Teraz już możemy to zrobić. Ostatnim krokiem jest jeszcze wyrównanie wydobytego zakresu bitów do prawej strony słowa, co czynimy odpowiednim przesunięciem.
Oto więc kompletny algorytm wyłuskiwania ciągłego obszaru bitów z podanego słowa:

```
// odczyt zakresu bitów w słowie x o długości n, zaczynającego się od bitu k
m = ~0
m = m << n
m = ~m
m = m << k

t = x & m
r = t >> k
```

Ceną za brak zależności od wielkości *s* (długości słowa maszynowego) jest większa liczba instrukcji w algorytmie. Ponadto jednak zobaczyliśmy pierwszy przykład użycia masek bitowych – w sumie więc była to bardzo korzystna transakcja :)

Ustawianie

Następne zadanie polega na ustawieniu wszystkich bitów danego zakresu na 1 (bity w takim stanie określa się często po prostu jako **ustawione**). Zamierzamy już więc dokonać na słowie pewnych miejscowych modyfikacji – tego nie da się wykonać korzystając wyłącznie z przesunięć. Każde rozwiązanie iniejszego problemu będzie zatem wymagało użycia maski i którejś z dwuargumentowych operacji bitowych.

Żeby jednak nie było bardzo trudno, okazuje się, iż stworzona w poprzednim paragrafie maska doskonale nadaje się do tego celu. Zmienić musimy jedynie działanie na niej: zastosujemy **sumę bitową**. Decydują o tym jej następujące własności:

```
b | 1 = 1
b | 0 = b
```

„I to zadziała?”, możesz spytać. Odpowiadam prędko, że jak najbardziej. Przypomnijmy sobie, że stworzona poprzednio maska ma taką postać:

Jedynki występują w niej na miejscach odpowiadających naszemu zakresowi. W wyniku działania sumy w wejściowym słowie ten właśnie obszar zostanie wypełniony jedynkami, co wynika z pierwszej z przytoczonych własności. Druga reguła w połączeniu z zerami na pozostałych miejscach maski zapewni nam natomiast, iż reszta słowa pozostanie nietknięta.

Konkludując, algorytm ustawiania wygląda następująco:

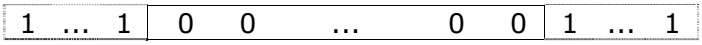
```
// ustawienie zakresu bitów w słowie x o długości n począwszy od k
m = ~0
m = m << n
m = ~m
m = m << k

r = x | m
```

To budujące, że stworzona w pocie czoła maska ma więcej niż jedno zastosowanie, nieprawdaż? :)

Zerowanie

Czas na operację przeciwną: chcemy mianowicie **wyzerować** zakres bitów. Tak naprawdę to już robiliśmy coś takiego. Odczytując nasze bity, pozbyliśmy się przecież wszystkiego poza nimi – gdzie „pozbycie się” oznaczało właśnie wyzerowanie bitów, aby nam więcej nie przeszkadzały.
Teraz chcemy zrobić coś podobnego, z tym że to właśnie wybrany zakres chcemy wyzerować, pozostawiając resztę bez zmian. Właściwości iloczynu bitowego nadal będą nam więc pomocne, lecz potrzebujemy innej maski. Musi ona mianowicie:

- zawierać zera w miejscach odpowiadających wybranemu obszarowi. Dzięki temu operacja koniunkcji wyzeruje te miejsca także w wejściowym słowie
- mieć jedynki na wszystkich pozostałych pozycjach. To sprawi, że po pomnożeniu zostaną one bez zmian



Czy widać jakąś zależność z maską, której używaliśmy dotąd? Cóż, mam nadzieję :)
Nietrudno bowiem zauważyć, że chodzi nam o jej **negację**: tam, gdzie mieliśmy wcześniej jedynki, chcemy zer - i na odwrót.
W sumie zatem maskę dla operacji zerowania zakresu tworzymy w identyczny sposób, lecz dodajemy jeszcze jedne, finalne odwrócenie znaczenia jej bitów:

```
// wyzerowanie zakresu bitów w słowie x o długości n począwszy od k
m = ~0
m = m << n
m = ~m
m = m << k
m = ~m

r = x & m
```

Stosowanym działaniem dla maski i wejściowego słowa *x* jest rzecz jasna ponownie iloczyn.

Zmiana bitów na przeciwne

Ostatnia operacja na zakresie bitów to jego **negacja**: chcemy zamienić w nim wszystkie bity na przeciwne. nie tykając jednocześnie żadnych bitów spoza zakresu. Zdajesz sobie naturalnie sprawę, że zwykła operacja negacji za pomocą operatora ~ nie jest absolutnie tym, czego szukamy: odwróciłaby nam ona bity w całym słowie.
Po raz kolejny więc uciekniemy się do masek. Wpierw jednak musimy znaleźć taką dwuargumentową operację bitową, którą możnaby zastosować do selektywnej negacji. Pytanie brzmi: czy znamy taką?…

Jak zapewne wiesz, odpowiedź jest pozytywna :) Tym szukanym działaniem jest nieco zapomniana **suma modulo 2**, czyli różnica symetryczna. Ma ona bowiem dwie bardzo cenne właściwości:

```
b ^ 0 = b
b ^ 1 = ~b
```

Dzięki nim będziemy mogli zanegować wartości tych bitów słowa, którym odpowiadają jedynki w masce. Pozostałe (te powiązane z zerami w masce), pozostaną natomiast bez zmian.
Widać zatem od razu, że poprawną maską jest ta z jedynkami w wybranym zakresie i zerami na innych pozycjach:

0 … 0	1 1 … 1 1	0 … 0

Użyjemy jej więc po raz trzeci, a algorytm negacji różnił się będzie od algorytmu ustawiania tylko jednym znakiem:

```
// zanegowanie zakresu bitów w słowie x o długości n począwszy od k
m = ~0
m = m << n
m = ~m
m = m << k

r = x ^ m
```

Podstawowe manipulacje na zakresie bitów przeprowadzamy więc bardzo podobnie, korzystając z (niemal) tej samej maski bitowej.

Przesunięcia cykliczne

Pod koniec poprzedniego podrozdziału obiecałem, iż wkrótce uzupełnimy występujący w C++ brak przesunięć cyklicznych. Przyszedł właśnie na to czas.

Zastanówmy się więc, jak przy użyciu znanych nam operacji na bitach – czyli zwykłych przesunięć oraz operatorów logicznych – możemy zaimplementować przesunięcia cykliczne. Przypominam, że mają one tę szczególną cechę, iż bity „wypchnięte” z jednej strony słowa pojawiają się automatycznie po jego drugiej stronie – zamiast zer, jak przy zwykłym przesunięciu.
Jak zatem moglibyśmy osiągnąć taki efekt?… Spróbujmy może **wyciąć** ów tracony przy normalnym przesuwaniu fragment i **wstawić** go w miejsce powstających zer na drugim końcu słowa. W ten sposób osiągniemy żądany rezultat.

Cykliczne przesunięcie w lewo

Dla porządku popatrzmy jeszcze raz, jak wygląda cykliczne przesuwanie; przykładem będzie niniejsza tabelka:

	1	0	0	1	1	0	1	0
RL								3
=	1	1	0	1	0	1	0	0

Tabela 9. Jeszcze jeden przykład przesunięcia cyklicznego

Normalne przesunięcie w lewo o *k* miejsc powoduje utratę *k* najbardziej znaczących bitów. Powinny one pojawić się po prawej stronie słowa, tam gdzie zwykle otrzymujemy *k* zer. Jeśli więc chcemy otrzymać rotację, nie pozostaje nam nic innego jak:

- wydobyć owych *k* lewostronnych bitów
- dokonać zwykłego przesunięcia w lewo o *k* miejsc
- wstawić zachowane bity po prawej stronie słowa

W tym momencie możemy zauważyć, że interesujące nas lewostronne bity zajmują *k* miejsc, począwszy od *s*-*k*, a skończywszy na *s*-1. Moglibyśmy więc wydobyć ten właśnie zakres przy użyciu metod opisanych w poprzednich paragrafach. Ponieważ i tak konieczne jest użycie zapisanej wprost długości słowa (*s*), możemy po prostu **dosunąć** te *k* bitów do prawej strony:

	s-1	...	*s*-*k*	*	*	*	...	*	*	*
>>										*s*-*k*
=	0	0	0	...	0	0	0	*s*-1	...	*s*-*k*

Tabela 10. Pierwszy etap implementacji przesunięcia cyklicznego w lewo

```
a = x >> (s - k)
```

Pierwszym krokiem jest zatem przesunięcie bitów w wejściowym słowie o *s*-*k* miejsc **w prawo**. Tak oto zachowamy zwykle tracony zakres bitów i będziemy mogli go wstawić w powstające wolne miejsce.

Miejsce to powstanie zaś wtedy, gdy nasze wejściowe słowo przesunięmy w lewo, tworząc zera po jego prawej stronie:

```
b = x << k
```

W sumie otrzymamy więc dwa kawałki wyniku, które należy skleić. *a* to zapisane lewostronne bity, wyrównane do prawej. Jest ich dokładnie *k*, zaś reszta tego słowa to zera. W *b* mamy natomiast pozostałe bity słowa x, dla odmiany wyrównane do lewej i także uzupełnione zerami. Jak to połączyć?…
Nie jest to trudne. Owe zera w obu słowach pozwalają nam bowiem zastosować **sumę bitową** do złożenia tych dwóch fragmentów:

a		0	...	0	*s*-1	... *s*-*k*
b	|	*	...	*	0	... 0
r	=	*	...	*	*s*-1	... *s*-*k*

Tabela 11. Trzeci i ostatni etap implementacji przesunięcia cyklicznego w lewo

```
r = a | b
```

Wiemy przecież, że gdy jeden z argumentów sumy jest zerem, drugi w wyniku pozostanie bez zmian. Dlatego też dwa idealnie dopasowane fragmenty zostaną w ten sposób sklejone – tak, jak widać to w powyższej tabelce.

Reasumując: implementacja przesuwania cyklicznego w lewo w całości przedstawia się następująco:

```
// przesunięcie cykliczne słowa x w lewo o k miejsc
a = x >> (s – k)
b = x << k
r = a | b
```

Potencjalnie skomplikowane zadanie typu „wytnij i wklej” w rzeczywistości ma więc całkiem proste rozwiązanie.

Cykliczne przesunięcie w prawo

Zupełnie analogicznie przebiega proces cyklicznego przesuwania w prawo. Tym razem musimy zadbać o wypadające prawostronne bity, a zatem powinniśmy:

- wyłuskać *k* najmniej znaczących bitów
- dokonać zwykłego przesunięcia bitowego w prawo
- wstawić zachowane bity po lewej stronie słowa

Wszystko przebiega więc niemal identycznie i jedyną różnicą są dokładnie przeciwne zwroty operatorów przesunięcia w tym i poprzednim algorytmie:

```
// przesunięcie cykliczne słowa x w prawo o k miejsc
a = x << (s – k)
b = x >> k
r = a | b
```

Przykłady te udowadniają nam, że niektóre operacje logiczno-bitowe mogą być z powodzeniem stosowane nie tylko w odniesieniu do specjalnie przygotowanych masek bitowych, ale też służyć choćby tak, jak tutaj – sklejaniu kilku słów w jedno.

Rozwiązania oparte na bitach

Po tych wprowadzających, acz niezbyt banalnych zagadnieniach przyjrzymy się nieco bardziej wyrafinowanym zastosowaniom operacji na bitach. Przyjrzymy się więc powszechnie używanemu mechanizmowi flag bitowych oraz jednemu ze sposobów komputerowej reprezentacji zbiorów, implementowanego właśnie przy pomocy operacji na bitach.

Flagi bitowe

Każdy szanujący się język programowania posiada typ zmiennych odpowiedzialny za przechowywanie wartości typu logicznego. Mówię tutaj o takich zmiennych, które mogą przyjmować jedynie dwie wartości: prawdę (`true`, zazwyczaj odpowiada jej liczbowa wartość 1 lub -1) oraz fałsz (`false`, zwykle 0). W C++ typem typem jest `bool`, w innych językach nazwy są bardzo podobne.

To oczywiste, że zmienne takich typów wykazują naturalne podobieństwo do pojedynczych bitów. Są one jednak normalnymi zmiennymi, posiadającymi chociażby swój własny adres w pamięci, a zatem zajmującymi co najmniej jeden bajt. Zatem przynajmniej cały bajt jest tu wykorzystywany na zapisanie elementarnej informacji, która mogłaby zajmować osiem razy mniej.
Nie zawsze rzecz jasna taki stan rzeczy nam przeszkadza. W przypadku, gdy mamy do czynienia z jedną czy kilkoma odrębnymi flagami logicznymi, marnotrawstwo pamięci nie jest wielkie, a rozsądek nakazuje użycie po prostu zestawu zmiennych boolowskich. Jednakże sytuacja nie zawsze tak musi wyglądać.

Rzadziej w kodzie samego programu, częściej na styku program-biblioteka zewnętrzna występuje konieczność przekazywania wielu wartości logicznych naraz. Typowy przykład

to określenie formatowania wypisywanego tekstu (pogrubienie, kursywa, podkreślenie itd.), opisywanego zwykle kilkoma „włącznikami”. Wielu innych przykładów dostarczają powszechnie wykorzystywane systemy GUI, API graficzne czy w końcu funkcje wielu systemów operacyjnych.
W takich właśnie wypadkach obsługa zestawów wartości logicznych odbywa się najczęściej przy pomocy **flag bitowych**, będących tematem tego paragrafu. Nawet jeśli generalnie uważasz, że znajomość „niskopoziomowych” operacji na bitach jest raczej zbędna, to trudno będzie ci uniknąć programowania z użyciem interfejsów, które korzystają właśnie z flag bitowych. Zatem zachęcam do uważnego przestudiowania następnych akapitów.

Idea flag bitowych

Właściwie nie jest specjalnie trudno wydedukować, na czym polega istota samego pomysłu wykorzystania flag bitowych. Wobec tego, co powiedzieliśmy wcześniej, staje się jasne, że chodzi tu o zastąpienie kilku zmiennych typu logicznego jednym **słowem bitowym**. Zgadza się; funkcję wartości „prawda” i „fałsz” poszczególnych zmiennych pełnią tutaj stany poszczególnych bitów. Naturalne i wygodne jest oczywiście przyjęcie, że prawda odpowiada bitowi ustawionemu (na 1), a fałsz – wyzerowanemu. To nie tylko, *nomen omen*, logiczne, ale także ułatwia kodowanie i dekodowanie tak spreparowanego słowa.

A zatem wszystko sprowadza się do wzięcia słowa odpowiedniej długości (zazwyczaj najmniejszego koniecznego) i umówieniu się co do znaczenia stanu poszczególnych jego bitów. W poprzednim podrozdziale poznaliśmy sposoby na dostęp do zakresów bitów (a więc także i do pojedynczych), więc nie mielibyśmy problemów zarówno z odczytywaniem, jak i zapisywaniem tak kodowanych wartości. To w gruncie rzeczy dobry pomysł, prawda?…

Nieprawda :) Same flagi bitowe są oczywiście bardzo dobrym pomysłem, ale konieczność pamiętania znaczenia indeksów poszczególnych bitów i ich ręcznego odczytywania nie jest absolutnie tym, czego chcemy. Tak naprawdę bowiem mechanizm flag bitowych może być od strony programisty-klienta wykorzystywany nawet bez znajomości operacji na bitach i w ogóle wiedzy o tym, że takie operacje są w niego zamieszane!
Nas oczywiście taka postawa nie zadowala :) Chcielibyśmy bowiem nie tylko świadomie korzystać z flag używanych choćby przez potrzebne nam biblioteki programistyczne, ale też w razie potrzeby tworzyć własne. ich zestawy, dostosowane pod nasze konkretne wymagania. Zobaczmy więc, jak można to zrobić.

Definiowanie odpowiednich stałych

Oto zupełnie kawałek autentycznego kodu wykorzystujący flagi bitowe:

```
// tworzymy okno funkcją CreateWindowEx
g_hwndOkno = CreateWindowEx(WS_EX_TOOLWINDOW,
                            g_strKlasaOkna.c_str(),
                            NULL,
                            WS_OVERLAPPED | WS_BORDER
                            | WS_CAPTION | WS_SYSMENU,
                            0, 0,
                            125,
                            80,
                            NULL,
                            NULL,
                            hInstance,
                            NULL);
```

Jest to wywołanie funkcji tworzącej nowe okno w systemie operacyjnym Windows. Wśród licznych jej parametrów znajdziemy między innymi tytuł nowego okna, jego wymiary oraz tzw. uchwyt do tworzącej je aplikacji. Nas jednak najbardziej interesuje tutaj argument czwarty, gdyż jest on właśnie zestawem flag bitowych. Określają one pewne aspekty okna nazywane jego stylem; dzięki nim możemy na przykład sprecyzować, czy tworzone okno ma posiadać grubsze obramowanie, pasek tytułu, menu, itd.
Jak widać chodzi tutaj o kilka(naście) włączników, mogących przyjmować dwa logiczne stany. Możliwości akurat tutaj jest mnóstwo i zapewne nie wykorzystamy ich wszystkich. Gdyby jednak w powyższej funkcji dla każdej z nich zarezerwowano osobny parametr, chcąc nie chcąc musielibyśmy podawać wszystkie – z których większość i tak byłaby równa `false`. Takie wywołanie nie należałoby zapewne do bardzo czytelnych.
I tutaj z pomocą przychodzą flagi bitowe. Gdy ich używamy, każda możliwość posiada swoją nazwę, która w dodatku jest widoczna w wywołaniu funkcji. Programista znający dany interfejs bez trudu odczyta więc, że akurat tutaj chodzi nam o okno posiadające pasek tytułu (`WS_CAPTION`), obramowanie (`WS_BORDER`), przycisk systemowego menu (`WS_SYSMENU`) oraz niebędące tzw. oknem wyskakującym (`WS_OVERLAPPED`). Jak widzimy, opcje te podajemy w jednym parametrze funkcji, oddzielając je znakiem pionowej kreski `|`. Znajomym zresztą.

Wiemy już zatem, jak tego używać, ale jeszcze niezupełnie domyślamy się, jak to działa. Otóż kluczową rolę pełnią tutaj wartości tych specjalnych stałych. Są one odpowiednio dobranymi słowami bitowymi. Każde takie słowo charakteryzuje się tym, iż:

- ma ustawiony pewien określony bit, przy czym wybór tego bitu jest unikalny dla danej opcji
- wszystkie pozostałe bity tego słowa są zerami

Ogólnie stałe flag bitowych są zwykle definiowane w taki sposób, że:

- pierwsza stała ma jedynkę w bicie o indeksie 0 i zera we wszystkich pozostałych
- druga stała ma jedynkę w bicie o indeksie 1 i zera we wszystkich pozostałych
- trzecia stała ma jedynkę w bicie o indeksie 2 i zera we wszystkich pozostałych
- itd.

Odpowiadające tym stałym liczby w zapisie binarnym to oczywiście 1, 10, 100, 1000, 10000, i tak dalej. Czy wyglądają one znajomo?… Oczywiście (mam nadzieję :)), to kolejne potęgi dwójki: 1, 2, 4, 8, 16, itd. Następny paragraf wyjaśni nam, dlaczego właśnie takie wartości są tutaj konieczne.
Na razie zastanówmy się nad wygodnym sposobem definiowania takich stałych. Możemy rzecz jasna wpisywać wartości dziesiętne, a nawet szesnastkowe, jeśli nam to odpowiada. Zwykle jednak rozsądniej jest posłużyć się **przesunięciem bitowym w lewo** – w taki oto sposób:

```
const unsigned FLAG1 = 1 << 0;   // 0...000001
const unsigned FLAG2 = 1 << 1;   // 0...000010
const unsigned FLAG3 = 1 << 2;   // 0...000100
const unsigned FLAG4 = 1 << 3;   // 0...001000
const unsigned FLAG5 = 1 << 4;   // 0...010000
// (itd.)
```

Ma to tę zaletę, iż ukazuje dosłownie w kodzie, któremu bitowi odpowiada konkretna stała; jest to naturalnie drugi argument operatora przesunięcia. Jest to tym bardziej pomocne, jako że C++ nie udostępnia przecież możliwości definiowania stałych w systemie dwójkowym, co zapewne byłoby tu najlepsze.

Kodowanie i dekodowanie zestawu flag

Wiemy już zatem, że stałe odpowiadające flagom bitowym powinny mieć wartości opisane powyżej. Jest to bowiem związane ze sposobem, w jaki będziemy z nich tworzyć (kodować) zestawy rzeczywistych flag oraz z ich późniejszym odczytem (dekodowaniem).

Przykład kodowania takiego zestawu flag widzieliśmy już w przykładzie z poprzedniego paragrafu. Absolutnie nie jest to trudne, gdyż:

W celu utworzenia zestawu opcji oznaczonych flagami bitowymi, należy odpowiadające im stałe potraktować operatorem **sumy bitowej** (`|`).

Dla naszych uprzednio zdefiniowanych flag możemy zatem tworzyć choćby takie przykładowe kombinacje:

```
FLAG1 | FLAG3                    // 0...000101
FLAG2 | FLAG3 | FLAG5            // 0...010110
FLAG4 | FLAG5                    // 0...011000
FLAG2                            // 0...000010
```

Jak można zaobserwować, korzystanie z operatora sumy dla tak określonych stałych powoduje po prostu **ustawianie odpowiednich bitów**. W tym celu możnaby nawet użyć zwykłego dodawania, ale nie zaleca się tego robić ze względu na to, iż nie mamy tu przecież na myśli liczb w sensie arytmetycznym.

Mamy zatem kilka opcji zapisanych już w postaci jednego słowa z wielce znaczącymi bitami. Należałoby teraz znaleźć sposób na wygodne sprawdzanie, czy wśród nich ustawiony jest ten jeden konkretny bit (czyli czy została wybrana dana opcja). Opierając się na naszym przykładzie, możemy sobie wyobrazić, że podobnych sprawdzeń dokonuje wewnętrznie funkcja `CreateWindowEx()`.
Do tego celu także posłużymy się zdefiniowanymi stałymi oraz jednym z operatorów bitowych. Mianowicie:

Odczytanie stanu konkretnej flagi bitowej polega na użyciu operatora **iloczynu bitowego** (`&`) wobec odpowiadającej jej stałej oraz sprawdzanego zestawu flag.

Przekładając to na odpowiedni kod, możemy sobie wyobrazić jakąś nadzwyczaj przykładową funkcję wykorzystującą określane przez nas flagi:

```
void DoSomething(unsigned uOptions)
{
    if (uOptions & FLAG1)
    {
          // FLAG1 jest ustawiona
    }

    if (uOptions & FLAG2)
    {
          // FLAG2 jest ustawiona
    }

    // (itd.)
}
```

Bitowe koniunkcje w warunkach instrukcji `if` będą produkowały wyniki niezerowe **wyłącznie wtedy**, gdy odpowiednie bity są ustawione. Wyjaśnienie jest proste. Nasze flagi mają przecież zera na wszystkich miejscach poza jednym określonym; w wyniku

zastosowania iloczynu te zerowe bity w rezultacie nadal będą zerami. Liczył się więc będzie tylko ten jeden wyróżniony bit będący jedynką: da on jedynkę również w rezultacie - ale tylko wtedy, gdy w sprawdzanym zestawie flag na tym właśnie miejscu bit również był ustawiony.
W sumie zatem otrzymamy wartość będącą zerem, jeśli dana flaga nie jest ustawiona – co oczywiście da nam fałszywy warunek. W przeciwnym wypadku wynik będzie niezerowy (dokładniej: będzie on równy stałej użytej w koniunkcji). Ostatecznie zatem całe to sprawdzenie, nawet jeśli wydaje się trochę skomplikowane, działa dokładnie tak, jak byśmy chcieli.

Reprezentacja zbiorów

Mechanizm flag bitowych może nam posłużyć jako podstawa dla bardzo podobnego, acz nieco bardziej wyrafinowanego rozwiązania. Możliwe jest mianowicie **reprezentowanie zbiorów** i wykonywanie na nich podstawowych operacji - takich jak suma czy przecięcie – przy użyciu słów (ciągów) bitowych i znanych nam już dobrze działań na takich słowach.
Jest to zgrabne i bardzo efektywne (z pewnością bardziej niż na przykład klasa `set` z STL), wymaga jednak spełnienia pewnego warunku. Otóż nasze zbiory nie mogą być zbyt duże. Mówiąc precyzyjnie, liczba **dopuszczalnych** elementów zbioru nie może przekraczać długości słowa. Jeżeli takie wymaganie nie może być spełnione, należy poszukać innego rozwiązania (jak choćby wspomnianego pojemnika z STL).

Zanim przyjrzymy się technice reprezentacji zbiorów jako słów bitowych, musimy sobie powiedzieć, co w takich sytuacjach informatycy rozumieją pod pojęciem zbiór. W matematyce zbiory mogą zawierać absolutnie dowolne elementy należące do jakiejkolwiek kategorii pojęć (liczby, funkcje, punkty, inne zbiory, itd.), a jedynym zastrzeżeniem jest ich **unikalność** – każdy z tych elementów może wystąpić co najwyżej raz[3].
W informatyce zbiory zawierają jedynie elementy należące do tego samego typu danych i ich asortyment jest oczywiście ograniczony. Można zatem mówić o zbiorach liczb, napisów czy nawet struktur, ale także to zbiorach zawierających stałe należące do jakiegoś **typu wyliczeniowego**. Takie właśnie zbiory są dla nas najbardziej interesujące, gdyż:

- stałe typów wyliczeniowych są zwykle tylko symboliczne (nie ma znaczenia ich konkretna wartość)
- możliwych wartości typów wyliczeniowych nie jest zazwyczaj zbyt wiele, zatem nierzadko mieszczą się one w wymienionym wcześniej ograniczeniu liczbą bitów w słowie

Tak naprawdę takimi zbiorami już się zajmowaliśmy. Zestawy flag bitowych są przecież tak rozumianymi zbiorami, w których elementami są poszczególne opcje, mogące wystąpić (należeć do zbioru) lub nie.
I oto właśnie tutaj chodzi. Różnica jest jedynie koncepcyjna: teraz stan określonego bitu będzie nam mówił nie to, czy dana flaga jest ustawiona, ale czy dany **element należy do zbioru**. Poza tym jest bardzo podobnie: w obu przypadkach – zestawu flag bitowych i zbioru – reprezentacją są słowa bitowe, zaś wartości stałych (dla zbiorów odpowiadające **możliwym** elementom) są takie same:

```
const unsigned ELEMENT1 = 1 << 0;      // 0...000001
const unsigned ELEMENT2 = 1 << 1;      // 0...000010
const unsigned ELEMENT3 = 1 << 2;      // 0...000100
const unsigned ELEMENT4 = 1 << 3;      // 0...001000
```

[3] Jeśli elementy mogą występować wielokrotnie, wtedy mówimy o wielozbiorze. Taki twór również może (na podobnych zasadach jak zwykłe zbiory) reprezentowany w komputerze, acz już nie przy pomocy ciągów bitowych.

```
const unsigned ELEMENT5 = 1 << 4;      // 0...010000
// (itd.)
```

Oczywiście nadal stałe te mogłyby być częścią jakiegoś typu wyliczeniowego. Pamiętajmy jednak, że tak być nie może już dla wynikowych zbiorów, będących kombinacjami potencjalnie wielu takich wartości.
Dla zachowania ogólności dalej będę posługiwał się zwykłym typem liczbowym `unsigned` zarówno dla zbiorów, jak i ich elementów – z tym że stworzę sobie dla niego dwa aliasy:

```
typedef unsigned SET;            // zbiór
typedef unsigned ITEM;           // element zbioru
```

Typ `ITEM` w prawdziwym programie mógłby więc odpowiadać odpowiedniemu `enum`'owi.

Teraz pozostaje nam tylko przyjrzeć się, jak wykonywać podstawowe operacje z dziedziny teorii zbiorów na tak reprezentowanych zbiorach.

Działania na elementach

Pusty zbiór nie zawiera żadnych elementów, zatem wszystkie bity odpowiadającego mu słowa powinny być wyzerowane. Jest to więc po prostu **zero**. Ponieważ jednak taki zbiór jest mało interesujący, nauczmy się **dodawa**ć do niego elementy. Robi się to całkiem prosto:

```
SET Set_Include(SET Set, ITEM Item)    { return Set | Item; }
```

Identycznie jak w przypadku flag bitowych, elementy (opcje) łączymy przy pomocy operatora bitowej sumy. To ustawia nam odpowiednie bity w słowie będącym przedstawieniem naszego zbioru.

Nową, pożądaną operacją dla zbiorów byłoby **usuwanie** z ich elementów. Wymaga to dokonania niespotykanych wcześniej bitowych manipulacji, lecz nie powinno być bardzo trudne:

```
SET Set_Exclude(SET Set, ITEM Item)    { return Set & ~Item; }
```

Chodzi nam o wyzerowanie specyficznego bitu, co czynimy koniunkcją. Maską dla tej operacji jest słowo będące negacją „normalnego" elementu; będzie ono miało jedynki na wszystkich miejscach poza wyróżnionym. To nam zresetuje odpowiedni bit, nie naruszając pozostałych. Wynikiem będzie słowo bez ustawionego odpowiedniego bitu – czyli zbiór bez podanego elementu.

Trzecią operacją odnoszącą się do zbiorów i ich elementów jest **sprawdzanie przynależności**:

```
bool Set_Has(SET Set, ITEM Item)       { return Set & Item; }
```

I znów nie ma tu nic zaskakującego. Jest to, tak jak przy flagach, odczytanie stanu konkretnego bitu. Czynimy to poprzez iloczyn i interpretację zerowego lub niezerowego wyniku. Robi to rzutowanie na `bool`, chociaż w niektórych kompilatorach trzeba jawnie napisać `!= 0`, by uniknąć ostrzeżenia podczas kompilacji.

Suma i przecięcie

Powyższe elementarne operacje są naturalnie konieczne, ale najwyższa pora na coś bardziej „zbiorowego" :) W matematyce chyba najważniejszymi działaniami na zbiorach są ich suma oraz przecięcie, zatem należałoby zadbać o ich dobrą implementację.

I tu niespodzianka: obie te operacje są bardzo proste w realizacji:

```
// suma zbiorów
SET Set_Union(SET Set1, SET Set2)      { return Set1 | Set2; }

// iloczyn (przecięcie) zbiorów
SET Set_Intersect(SET Set1, SET Set2)  { return Set1 & Set2; }
```

Bo przecież suma to suma, a iloczyn to iloczyn ;) Wyjaśnimy aczkolwiek po kolei, jak to działa.

Suma dwóch zbiorów to, jak wiemy, taki zbiór, który zawiera elementy z obu składników. Czyli jeżeli jakiś element należy do chociaż jednego z dwóch zbiorów, to będzie należał też do ich sumy. U nas relacja przynależności to po prostu ustawienie odpowiedniego bitu. Zatem jeśli ów bit jest ustawiony chociaż w jednym słowie, powinien być ustawiony w słowie będącym wynikiem. A tak przecież działa suma bitowa, co zresztą wiemy doskonale.
Iloczyn dwóch zbiorów „działa" analogicznie. Jeśli jakiś element należy do iloczynu dwóch zbiorów, to należy do ich obu. Czyli jeśli jakiś bit w wyniku będzie ustawiony, to znaczy iż był on ustawiony także w obu wejściowych słowach-zbiorach. To zapewnia nam z kolei iloczyn bitowy.

Te dwie operacje wymagają co najwyżej kilku instrukcji procesora, podczas gdy np. zbiory implementowane przy pomocy drzew wymagają tutaj czasu logarytmicznego. Różnica polega oczywiście na tym, że tutaj maksymalny rozmiar zbioru jest dosyć ograniczony, a elementy (symbolicznie) ustalone.

Dopełnienie i różnica zbiorów

Zostały dwie operacje z typowego zakresu teorii zbiorów – dopełnienie zbioru oraz różnica dwóch zbiorów.

W matematyce zdefiniowanie dopełnienia zbioru wymaga określenia tzw. przestrzeni. **Przestrzeń** jest to zbiór, w którym zawierają się wszystkie rozważane przez nas w danej chwili podzbiory. Przykładowo, jeśli matematyk zajmuje się podzbiorami R, to właśnie cały zbiór liczb rzeczywistych będzie dla niego przestrzenią.
Dopełnienie danego zbioru jest wówczas tymi elementami przestrzeni, które do zbioru **nie należą**. Dopełnienie jest to więc, mówiąc prosto, wszystko to co jest „poza zbiorem". Jak zaimplementować taką operację w przypadku naszych zbiorów?…

Przede wszystkim musimy określić, co jest przestrzenią. Tutaj sprawa jest dość oczywista: zbiór zawierający wszystkie inne to jednocześnie taki zbiór, który ma wszystkie możliwe elementy. Jego binarna reprezentacja to po prostu słowo ze wszystkimi bitami ustawionymi na 1. Przy takiej przestrzeni implementacja dopełnienia jest łatwa:

```
SET Set_Complement(SET Set)      { return ~Set; }
```

Negacja bitów wykonuje tutaj całą pracę. Odwracając stan bitów czynimy wyrzucamy ze zbioru wszystkie jego elementy i zamiast tego wstawiamy te, które wcześniej do niego nie należały. Oto kolejna (po sumie i iloczynie) operacja na zbiorach, która przy prezentowanej ich reprezentacji ma dokładny bitowy odpowiednik.

Przy użyciu dopełnienia możemy teraz uzupełnić nasze zbiory o ostatnie działanie, czyli **różnicę**. Istnieje bowiem sposób na zapisanie różnicy dwóch zbiorów przy pomocy iloczynu i dopełnienia:

$A \setminus B = A \cap \overline{B}$,

co w przekładzie na kod wygląda następująco:

```
SET Set_Diff(SET Set1, SET Set2) { return Set1 & ~Set2; }
```

Mając teraz kompletny obraz wszystkich operacji, widzimy, że dodawanie i usuwanie elementów oraz sprawdzanie ich przynależności są tak naprawdę szczególnymi przypadkami (odpowiednio) sumy, różnicy oraz iloczynu – wykonywanymi na zbiorach jednoelementowych.

Inne przydatne operacje

Ostatni podrozdział poświecimy kilku sztuczkom, które nie pasują nigdzie indziej :) Wszystkie one będą oczywiście wykorzystywały manipulacje na bitach przy użyciu zwykłych operatorów bitowych.

Potęgi dwójki

Ze względu na to, iż komputery powszechnie pracują z użyciem systemu binarnego, dwójka i jej potęgi są bardzo ważnymi liczbami. Znajomość pierwszych kilkunastu z nich bywa zresztą nierzadko całkiem przydatna. Dalej zobaczymy dwie procedury związane właśnie z tymi szczególnymi liczbami.

Sprawdzanie, czy liczba jest potęgą dwójki

Bardzo łatwo jest na przykład sprawdzić, czy podana liczba jest potęgą dwójki. Możnaby rzecz jasna zastosować przeglądową tablicę lub pętlę, ale czyż nie ładniej wykorzystać taką oto formułę:

```
bool IsPowerOf2(unsigned x)      { return !(x & (x - 1)); }
```

Pytanie oczywiście brzmi: dlaczego to działa? :) Trik tkwi w odpowiednim zastosowaniu koniunkcji i interpretacji wyniku.

Przy okazji poznawania flag bitowych wspomniałem o tym, jak w systemie binarnym są kodowane potęgi dwójki. Odpowiadały one stałym używanym jako flagi, gdyż mają one dokładnie jeden bit ustawiony, zaś pozostałe są zerami.
A co z liczbami dokładnie o jeden mniejszymi od potęg dwójki?… Być może łatwiej wyobrazić to sobie na przykładzie dziesiętnym. Weźmy taki milion, jeden i sześć zer; odejmując od niego 1 otrzymamy 999999, czyli sześć dziewiątek. Podobnie, jeśli mamy liczbę 2^i (z ustawionym *i*-tym bitem), to liczba 2^i-1 będzie się miała *i*-1 jedynek i nic poza tym!
Ilustruje to poniższa tabelka:

	0	…	0	1	0	…	0
&	0	…		0	1	…	1
=	0	0	0	…	0	0	0

Tabela 12. Sprawdzanie, czy liczba jest potęgą dwójki

W wynik koniunkcji takich dwóch wartości (*x* i *x*-1), otrzymujemy same zera, jeśli tylko *x* jest potęgą dwójki. Widać, że w takim wypadku po prostu dla żadnej pary odpowiadających sobie bitów nie jest możliwy inny wynik.

Z logicznego punktu widzenia zero jest jednak wynikiem dokładnie odwrotnym niż żądany. To już aczkolwiek nie jest problem: korzystając z **logicznego** zaprzeczenia otrzymujemy właściwy rezultat.

Zaokrąglanie w górę do najbliższej potęgi dwójki

W przypadku, gdy poprzednia funkcja zwróci nam `true`, możemy być pełni szczęścia ;) Co jednak zrobić w pozostałych sytuacjach? Wówczas możemy liczbę zaokrąglić do najbliższej potęgi dwójki. W praktyce szczególnie ważne jest **zaokrąglanie w górę** – na przykład w celu ustalenia rozmiarów tekstur mieszczących nieforemne obrazki (wciąż są karty graficzne wymagające tekstur, których wymiary są potęgami liczby 2).

Prawdopodobnie najprostszym sposobem zrealizowania takiego zaokrąglania jest… przeszukanie po kolei wszystkich potęg dwójki mniejszych od danej liczby. Realizuje to choćby ta oto funkcja:

```
unsigned CeilToPowerOf2(unsigned x)
{
   unsigned n = 1;
   while (n < x)     n <<= 1;
   return n;
}
```

Mimo iż nie jest ona zbyt odkrywcza, stanowi całkiem dobre rozwiązanie problemu.

Skrajne bity

Na deser podaję jeszcze dwie formuły, które raczej nie mają praktycznych zastosowań, ale są interesujące :) Służą one do tzw. wyodrębniania skrajnych bitów. Wskazuje one najmniej znaczący bit, który w podanym słowie jest jedynką lub zerem:

```
// skrajna jedynka
int GetLSBSet(int x)       { return x & -x; }

// skrajne zero
int GetLSBUnset(int x)     { return ~x & (x + 1); }
```

Rezultatem jest takie słowo, które zawiera bit ustawiony w pozycji szukanego. Jeśli na przykład x = ...10110, to pierwszy funkcja zwróci ...00010 (jedynka na pozycji skrajnej jedynki w x), a druga ...00001 (jedynka na pozycji skrajnego zera w x).

Jak i dlaczego te funkcje działają?… To drobne ćwiczenie pozostawiam do samodzielnego rozwiązania :D

Podsumowanie

Na tym kończymy nasze spotkanie z różnymi technikami programistycznymi, opierającymi się na manipulacjach bitami. Jeśli nawet nie wydaje ci się, aby były ci one w tej chwili przydatne, z pewnością nie możesz wykluczyć, że będą one takie w przyszłości. A poza tym pamiętaj, że żadna wiedza nie hańbi :)