B

REPREZENTACJA DANYCH W PAMIĘCI

Adam Sawicki „Regedit”
sawickiap@poczta.onet.pl

Jest 10 rodzajów ludzi –
- ci, którzy rozumieją kod dwójkowy
i ci, którzy go nie rozumieją.
hakerskie ujęcie socjologii

W tym dodatku mowa będzie o sprawach, które mają miejsce w komputerze praktycznie na najniższym możliwym poziomie. Będziemy się zajmowali zerami i jedynkami. Poznamy także sposób, w jaki komputer zapisuje w pamięci wszelkie informacje.
Wbrew temu, co mogłoby się wydawać, wiadomości tego rodzaju nie są bezużyteczne. Mają one ogromne zastosowanie w praktyce programistycznej. Dlatego radzę podejść do tej lektury poważnie i postarać się zrozumieć opisane tu, miejscami niestety niełatwe informacje.
Pamiętaj: **Wszystko wydaje się trudne, dopóki nie stanie się proste!**

Algebra Boole’a

Wbrew groźnie brzmiącej nazwie, zaczniemy od rzeczy całkiem prostej. Poznamy podstawowe, teoretyczne zasady operowania na zerach i jedynkach, zwane algebrą Boole’a lub logiką dwuwartościową.

Boole George (1815-1864), logik i matematyk angielski, od 1849 profesor matematyki w Queen's College w Cork (Irlandia), członek Towarzystwa Królewskiego (Royal Society) w Londynie. Zajmował się logiką formalną, rachunkiem prawdopodobieństwa, opracował algebrę dla zbioru dwuelementowego (algebra Boole'a). Główne dzieło - *An Investigation of The Laws of Thought* (1854).[97]

Algebra Boole’a posługuje się jedynie dwiema możliwymi cyframi. Przyjęło się zapisywać je jako 0 i 1. Można też wyobrazić je sobie jako dwa przeciwne stany – prawda (ang. *true*) i fałsz (ang. *false*), stan wysoki (ang. *high* – w skrócie H) i niski (ang. *low* – w skrócie L), gruby i chudy, yin i yang czy cokolwiek innego :)

Działania

Na tych dwóch dostępnych liczbach definiuje się kilka podstawowych działań.

Negacja

Jest to działanie jednoargumentowe oznaczane symbolem ~ (tzw. tylda – czyli taki wężyk pisany nieco u góry :) Bywa też oznaczane przez takie coś: ¬ lub przez pisany za

[97] Źródło: http://wiem.onet.pl/

negowanym wyrażeniem apostrof: `. Możnaby je porównać znanej z normalnej matematyki zamiany liczby na przeciwną za pomocą poprzedzającego znaku minus -. Tak jak liczba –5 jest przeciwna, do liczby 5, tak ~x oznacza stan przeciwny do stanu oznaczonego przez x. Ponieważ w logice dwuwartościowej wartości są tylko… dwie, nietrudno jest wypisać tabelkę dla tego działania:

x	*~x*
0	1
1	0

Tabela 13. Wartości logiczne negacji

Jak widać, zanegowanie wartości powoduje jej zamianę na wartość przeciwną, czyli drugą spośród dwóch możliwych.

Można jeszcze dodać, że negacja nazywana bywa też przeczeniem, a jej słownym odpowiednikiem jest słowo „nie” (ang. *not*). Jeśli głębiej zastanowisz się nad tym, wszystko okaże się… logiczne! Stan, który nie jest zerem – to jedynka. Stan, który nie jest jedynką – to zero :D

Koniunkcja

Przed nami kolejne działanie kryjące się pod tajemniczą nazwą. Jest to działanie dwuargumentowe, które można porównać znanego nam mnożenia. Symbolizuje go taki oto dziwny znaczek przypominający daszek: $\wedge$.

Mnożąc jakąkolwiek liczbę przez 0, otrzymujemy 0. Z kolei 1*1 daje w wyniku 1. Identycznie wynika iloczyn wartości Boole’owskich. Skonstruujmy więc tabelkę:

x	*y*	*x* $\wedge$ *y*
0	0	0
0	1	0
1	0	0
1	1	1

Tabela 14. Wartości logiczne koniunkcji

Koniunkcja bywa też nazywana iloczynem, a odpowiadającym jej słowem jest „i”. Faktycznie możemy zauważyć, że aby działanie dało w wyniku jedynkę, jedynką muszą być obydwa argumenty działania: pierwszy i drugi.

Alternatywa

Skoro jest mnożenie, powinno być też dodawanie. Pan Boole o nim nie zapomniał, więc mamy kolejne działanie. Jego symbol jest przeciwny do symbolu koniunkcji (odwrócony daszek) i wygląda tak: $\vee$.

Tylko dodawanie dwóch zer daje w wyniku zero. Jeśli choć jednym ze składników jest jedynka, wynikiem dodawania jest liczba większa od zera – 1 albo 2. Ponieważ dwójka w algebrze Boole’a nie występuje, zamienia się na… nie nie! Nie „zawija się” z powrotem na zero, ale zostaje jakby „obcięta” do jedynki.
Tabelka będzie więc wyglądała tak:

x	*y*	*x* $\vee$ *y*
0	0	0
0	1	1
1	0	1
1	1	1

Tabela 15. Wartości logiczne alternatywy

Słówkiem odpowiadającym alternatywie jest „lub”. Widzimy, że wynikiem działania jest 1, jeśli wartość 1 ma przynajmniej jeden spośród argumentów działania – pierwszy lub drugi. A więc wszystko się zgadza.

Różnica symetryczna

Działanie to jest często pomijane w podręcznikach logiki. Tymczasem jego znacznie z punktu widzenia programisty jest ogromne. Jak bardzo – to okaże się później.

Na razie zajmijmy się jego zdefiniowaniem. Aby sporządzić tabelkę, przyda się angielska nazwa tej operacji. Brzmi ona *exclusive or* (w skrócie *xor*) – co oznacza „wyłącznie lub”. Aby w wyniku otrzymać 1, jedynką musi być koniecznie tylko pierwszy lub tylko drugi argument tego działania, nie żaden ani nie obydwa.

x	***y***	***x ⊕ y***
0	0	0
0	1	1
1	0	1
1	1	0

Tabela 16. Wartości logiczne różnicy symetrycznej

To by było na tyle, jeśli chodzi o operacje logiczne konieczne do wprowadzenia cię w świat komputerowych bitów. Aby jednak twoja wiedza z dziedziny zwanej logiką (tak, tak! – na pierwszym roku informatyki jest osobny przedmiot o takiej nazwie, na którym uczą właśnie tego! :) była pełna, opiszę jeszcze szybciutko pozostałe dwa działania.

Ekwiwalencja

Ekwiwalencja to inaczej równoważność i odpowiada jej nieco przydługie stwierdzenie o treści: „wtedy i tylko wtedy, gdy”. Daje ono w wyniku jedynkę wtedy i tylko wtedy, gdy obydwa argumenty są takie same. Można więc utożsamiać to działanie z równością. Symbolizuje go taka zwrócona w obydwie strony strzałka: ⇔.

x	***y***	***x ⇔ y***
0	0	1
0	1	0
1	0	0
1	1	1

Tabela 17. Wartości logiczne ekwiwalencji

Implikacja

To zdecydowanie najbardziej zakręcone i najtrudniejsze do zapamiętania działanie logiczne. Cieszmy się więc, że programista raczej nie musi go pamiętać :)

Inna nazwa implikacji to wynikanie, a odpowiadające mu stwierdzenie brzmi: „jeżeli …, to …”. Oznaczane jest strzałką skierowaną w prawo: ⇒. Oto jego tabelka:

x	***y***	***x ⇒ y***
0	0	1
0	1	1
1	0	0
1	1	1

Tabela 18. Wartości logiczne implikacji

Logicznego wyjaśnienia takiej a nie innej postaci tej tabelki nawet nie będę próbował się podjąć[98]. Przejdźmy teraz lepiej do dalszej części logiki, by jak najszybciej mieć ją już za sobą :)

Aksjomaty

Poznamy teraz kilka prostych wzorów, które ukażą nam podstawowe zależności pomiędzy poznanymi działaniami logicznymi.

Przemienność

$a \vee b = b \vee a$
Dodawanie też jest przemienne – jak w matematyce.
$a \wedge b = b \wedge a$
Mnożenie też jest przemienne.

Łączność

$(a \vee b) \vee c = a \vee (b \vee c)$
Dodawanie jest łączne – jak w matematyce.
$(a \wedge b) \wedge c = a \wedge (b \wedge c)$
Mnożenie też jest łączne.

Rozdzielność

$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
Mnożenie jest rozdzielne względem dodawania.
$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$
Dodawanie też jest rozdzielne względem mnożenia – a w normalnej matematyce nie!!!

Identyczność

$a \vee 0 = a$
$a \vee 1 = 1$
$a \wedge 0 = 0$
$a \wedge 1 = a$
To wynika bezpośrednio z tabelek.

Dopełnienie

$a \vee \sim a = 1$
$a \wedge \sim a = 0$
Bo jeden z argumentów zawsze będzie przeciwny do drugiego.

Prawa De Morgana

$\sim(a \vee b) = \sim a \wedge \sim b$
$\sim(a \wedge b) = \sim a \vee \sim b$

Logika w programowaniu

Uff… Pora wrócić do sedna sprawy, czyli do programowania. Tutaj często zachodzi potrzeba reprezentowania jednego z dwóch stanów. Przykładowo zmienna `Blad` w stanie

[98] Może niesłusznie, gdyż wyjaśnienie jest dość proste. Implikacja ilustruje wynikanie jednych faktów z drugich, a takie rozumowanie jest słuszne zawsze, z wyjątkiem sytuacji, gdy ze stwierdzenia prawidziwego wyprowadzamy stwierdzenie fałszywe. Odpowiada to trzeciemu wierszowi tabelki. [przypis: Xion]

1 oznaczałaby fakt wystąpienia błędu, a w stanie 0 fakt jego niewystąpnienia – czyli że wszystko jest w porządku.

Typ logiczny

Typem danych w C++ reprezentującym wartości logiczne jest `bool`. Dwa stany reprezentowane są zaś przez specjalne słowa kluczowe – `true` oraz `false`. Można też używać identyfikatorów `TRUE` i `FALSE` pisanych dużymi literami.

Dla przykładu weźmy linijkę kodu, który tworzy wspomnianą zmienną i wstępnie ją inicjalizuje:

```
bool Blad = false;
```

Wyrażenia logiczne

Oprócz bezpośrednich wartości `true` oraz `false`, wartości typu `bool` zwracane są także przez operatory porównania takie, jak `==` (równy), `!=` (różny), `<` (mniejszy), `>=` (większy lub równy) itp.

Każda okazja jest dobra, aby po raz kolejny przestrzec przez typowym błędem, na który (niestety?) w całej swej zachwalanej przez wielu elastyczności pozwala język C++. Chodzi o różnicę pomiędzy operatorem przypisania `=`, a operatorem porównania (równości) `==`. Ten pierwszy także zostanie zawsze zaakceptowany w miejscu drugiego, ale z pewnością spowoduje inny (czyli nieprawidłowy) efekt.
Uważaj na to!

Wartość innego typu – np. liczba – także może zostać potraktowana jako wartość logiczna. Przyjęte zostanie wówczas 0 (fałsz), jeśli wartość jest zerowa (np. liczbą jest 0) oraz prawda w każdym innym wypadku.
Ta cecha języka C++ jest całkiem przydatna, ponieważ pozwala sprawdzać „niezerowość" zmiennych (szczególnie wskaźników) bez posługiwania się operatorem porównania, np.:

```
if (Zmienna)
   std::cout << "Zmienna jest niezerowa";
```

Operatory logiczne

Poznane na początku działania algebry Boole'a mają, jak można się domyślać, swoje odpowiedniki w języku programowania. W C++ są to symbole odpowiednio:

- `!` – negacja – przeczenie – „nie" (jednoargumentowy)
- `&&` - koniunkcja - iloczyn - „i" (dwuargumentowy)
- `||` - alternatywa – suma – „lub" (dwuargumentowy)

Najłatwiej zrozumieć istotę działania tych operatorów zapamiętując ich słowne odpowiedniki (te w cudzysłowach). Rozważmy przykład:

```
int Liczba = 7;
void* Wskaznik = 0;
bool Wartosc = ( !(Wskaznik) || (Liczba == 6) ) && false;
```

`Wskaznik` jest zerowy, a więc jego wartością logiczną jest `false`. Po zanegowaniu zmienia się w `true`. Zmienna `Liczba` nie jest równa 6, a więc wartością porównania będzie `false`. `true` lub `false` daje `true`, `true` i `false` daje w końcu `false`. Zmienna `Wartosc` zostanie więc ostatecznie zainicjalizowana wartością `false`.
Postaraj się przeanalizować to jeszcze raz, dokładnie, i w pełni wszystko zrozumieć.

Systemy liczbowe

Odkąd wynaleziono pieniądze i koło, ludzie zaczęli kręcić interesy :) Równie dawno temu ludzie zaczęli liczyć. Policzyć trzeba było nie tylko pieniądze, ale np. upolowane mamuty i inne mniejsze albo większe rzeczy.

Liczby trzeba było jakoś zapisywać. Powstały więc różne sposoby na to. Na co dzień posługujemy się systemem dziesiętnym oraz cyframi arabskimi. Jednak znamy też np. cyfry rzymskie. Także podział na 10, 100 czy 1000 części nie jest wcale tak oczywisty, jak mogłoby się wydawać patrząc na jednostki miar takie, jak kilometr, centymetr czy kilogram. Doba ma przecież 24 godziny, a godzina 60 sekund.

To wszystko są pozostałości po przeszłości, które uświadamiają nam względność naszego sposobu liczenia i możliwość tworzenia nieskończenie wielu różnych, nowych sposobów zapisywania liczb.

Teoria

Poznamy teraz różne systemy liczbowe oraz nauczymy się zapisywać liczby w dowolnym z nich i zamieniać między nimi.
Na początek porcja nieco ciężkostrawnej teorii, którą jednak trzeba jakoś przetrawić :)

Wstęp

Zastanówmy się przez chwilę, w jaki sposób zapisywane są liczby. Dowolnie dużą liczbę jesteśmy w stanie zapisać za pomocą pewnej ilości cyfr, których mamy do dyspozycji dziesięć: 0, 1, 2, 3, 4, 5, 6, 7, 8 i 9. Stąd nazwa naszego systemu – system dziesiętny.

Jednak cyfra cyfrze nierówna. Na przykład w liczbie 123, cyfra 1 ma inne znaczenie niż cyfra 2 czy 3. Ta pierwsza nazwana bywa cyfrą setek, druga – cyfrą dziesiątek, ostatnia zaś – cyfrą jedności.
Skąd te nazwy? Zauważmy, że 1, 10, 100 itd. to kolejne potęgi liczby 10 – która jest podstawą naszego systemu dziesiętnego.

$10^0 = 1$
$10^1 = 10$
$10^2 = 100$
$10^3 = 1000$
itd.

System pozycyjny to taki, w którym znaczenie znaków zależy od ich pozycji.
System wagowy to taki, w którym każdej pozycji cyfry przypisana jest inna waga.

Wynika z tego, że nasze używane na co dzień cyfry arabskie w systemie dziesiętnym są systemem pozycyjnym wagowym. Cyfry rzymskie są wyłącznie systemem pozycyjnym, bo poszczególne pozycje cyfr nie mają w nim przypisanych na stałe wag, takich jak 1, 10, 100 itd.

Zostawmy już cyfry rzymskie w spokoju i zajmijmy się normalnymi cyframi arabskimi. Pomyślmy co by było, gdyby do zapisywania liczb używać innej ilości cyfr – np. tylko pięciu? Za ich pomocą także dałoby się zapisać dowolną liczbę. Rodzi się jednak pytanie: jakie byłyby to cyfry?

W systemach o podstawie N mniejszej niż 10 używamy N pierwszych cyfr, tzn. cyfr od 0 do (N-1) włącznie. Np. w systemie siódemkowym używalibyśmy siedmiu cyfr: 0, 1, 2, 3, 4, 5 i 6.

Kiedy zabraknie cyfr, stosuje się kolejne litery alfabetu. Mogą być małe albo duże, ale chyba lepiej wyglądają duże. Np. w systemie trzynastkowym używalibyśmy znaków: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B i C i wszystkie je nazywalibyśmy cyframi.

Wzór

A teraz uwaga, bo będzie straszny wzór ;)
Pokaże nam on, w jaki sposób „zbudowana jest" każda liczba w dowolnym systemie.

$$L = \sum_{i=m}^{n} a_i N^i$$

$m, n \in C, m \leq 0, n \geq 0, m \leq n$

L to nasza liczba.
N to podstawa systemu (np. 10 dla systemu dziesiętnego).
m to indeks ostatniej cyfry (tej z prawej strony), albo inaczej mówiąc liczba przeciwna do ilości cyfr po przecinku, np. w liczbie 1984.0415 m=-4.
n to indeks pierwszej cyfry (tej z lewej strony), albo inaczej mówiąc ilość cyfr przed przecinkiem pomniejszona o 1, np. w liczbie 1984.0415 n=3.
Wynika z tego, że pierwsza cyfra przed przecinkiem ma indeks 0, poprzednie cyfry mają kolejne indeksy dodatnie, a cyfry po przecinku mają kolejne indeksy ujemne numerowane w drugą stronę.
i to indeksy kolejnych cyfr.
$\mathbf{a_i}$ to kolejne cyfry w naszej liczbie.

Przykład

Zanim jednak pokażę przykład, musisz wiedzieć jeszcze jedną ważną rzecz. Otóż musimy nauczyć się oznaczania, w jakim systemie zakodowana (czyli zapisana) jest dana liczba. Inaczej nie wiedzielibyśmy np., czy liczba 320 zapisana jest w systemie czwórkowym, piątkowym czy może dziewiątkowym.

Dlatego wprowadźmy następujące oznaczenie: Przyjmujemy, że system, w jakim zakodowana jest liczba, zapisywali będziemy w indeksie dolnym za nawiasem, w który ujęta jest dana liczba, np. $(320)_5$. Jeśli liczba występuje bez nawiasu i indeksu umawiamy się, że zakodowana jest w naszym normalnym systemie dziesiętnym.

Możemy już przystąpić do przeliczenia liczby z jakiegoś systemu na system dziesiętny. Weźmy liczbę $(320)_5$. Rozwijając ją wg przedstawionego wyżej wzoru mamy:

$(320)_5 = 3*5^2 + 2*5^1 + 0*5^0 = 3*25 + 2*5 + 0*1 = 75 + 10 + 0 = 85$

Okazuje się, że liczba $(320)_5$ zapisana w systemie piątkowym przyjmuje w systemie dziesiętnym postać liczby 85.

Nie mniej ważne od zapisywania jest odpowiednie czytanie liczb. Liczby w systemie innym niż dziesiętny **nie wolno** czytać tak, jak np. „trzysta dwadzieścia"! Należy mówić zawsze „trzy, dwa, zero".
Dlaczego? Zauważ, co oznaczają tamte słowa. „Trzysta dwadzieścia" to „trzy setki" i „dwie dziesiątki". Nieświadomie mówimy więc w ten sposób o cyfrze setek i cyfrze dziesiątek, a te kolejne potęgi dziesiątki są wagami kolejnych cyfr jedynie w systemie dziesiętnym.

Patrząc na powyższy przykład można przy okazji wysnuć wniosek, że w systemie piątkowym mamy do czynienia z „cyfrą dwudziestek piątek”, „cyfrą piątek” i „cyfrą jedności”, a wcześniej zapewne z „cyfrą sto dwudziestek piątek” (bo $5^3 = 125$).

Być może zwróciłeś uwagę na prawidłowość, że do zapisania tej samej liczby w systemie o niższej podstawie (mniejszej ilości dostępnych cyfr) potrzeba więcej cyfr.

Ćwiczenia

Począwszy od tego miejsca zamieszczał będę zadania do samodzielnego wykonania wraz z odpowiedziami w przypisie. Mocno zalecam wykonanie przynajmniej niektórych z nich, ponieważ pozwolą ci one lepiej zrozumieć istotę sprawy oraz wyćwiczyć umiejętności potrzebne do zrozumienia dalszej partii materiału.

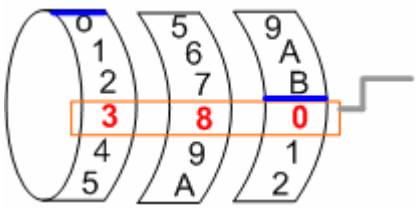
Zadanie 1[99]
Rozkoduj do systemu dziesiętnego liczby:

1. $(13)_7$
2. $(666)_7$
3. $(666)_{11}$
4. $(ABBA.1)_{13}$

Praktyka

Jeśli po tych teoretycznych rozważaniach nie bardzo potrafisz wyobrazić sobie to wszystko, nie martw się. Właśnie teraz jest czas i miejsce, by spróbować wyjaśnić systemy liczbowe trochę bardziej „łopatologicznie”.

Wyobraź sobie mechaniczny licznik, np. gazu, prądu, wody, kilometrów lub jakikolwiek inny, który masz w domu albo w samochodzie.



Rysunek 6. Liczba jako mechaniczny licznik z tarczami.

Licznik taki składa się z kilku tarcz, które mogą się obracać. Na ich obwodzie napisane są kolejne cyfry. Granicę między cyfrą ostatnią a pierwszą zaznaczyłem na rysunku niebieską linią (jaki system liczbowy przedstawia rysunek?)[100].

Zasada działania licznika jest następująca: Kręcimy za szarą korbkę powodując obracanie się ostatniej tarczy (tej po prawej stronie). Tarcza pokazuje kolejne cyfry. Kiedy dojdzie do ostatniej i zostanie po raz kolejny obrócona, pokazywała będzie z powrotem pierwszą cyfrę (czyli 0). Dodatkowo spowoduje wtedy przekręcenie następnej tarczy o jedną cyfrę do przodu.

Nietrudno wyobrazić sobie co będzie, kiedy ta druga tarcza osiągnie ostatnią cyfrę. Po następnym obróceniu pokaże 0 oraz spowoduje zwiększenie o jedną pozycję tarczy

[99] 1) $1*7^1 + 3*7^0 = 1*7 + 3*1 = 7 + 1 = 8$; 2) $6*7^2 + 6*7^1 + 6*7^0 = 294 + 42 + 6 = 342$; 3) $6*11^2 + 6*11^1 + 6*11^0 = 726 + 66 + 6 = 798$; 4) $10*13^3 + 11*13^2 + 11*13^1 + 10*13^0 + 1*13^{-1} = 23982.0769...$
[100] Cyframi są znaki od 0 do B, cyfr jest wobec tego 12, a więc chodzi o system dwunastkowy.

trzeciej. Ogólnie można powiedzieć, że każdy pełny obrót tarczy poprzedniej powoduje na koniec obrócenie tarczy następnej (na lewo od niej) o jedną pozycję.
Dlatego w systemie dziesiętnym po liczbie 9 następuje liczba 10, a po liczbie 99 występuje liczba 100.

Ćwiczenia

Zadanie 2[101]
Wyprowadź tabelkę kilku kolejnych liczb systemu trójkowego począwszy od 0 korzystając z wyobrażenia liczby jako licznika z tarczami.

Kodowanie liczb całkowitych

Potrafimy już rozkodować liczbę zapisaną w dowolnym systemie na system dziesiętny. Pora nauczyć się kodować liczbę dziesiętną w dowolnym innym systemie.

Nie bój się, nie będzie kolejnego strasznego wzoru :) Takie przeliczanie to czysta praktyka i doskonała zabawa. A więc zaczynamy!

Reszta z dzielenia

Do zabawy potrzebny będzie kalkulator oraz przypomnienie pewnego dawno zapomnianego drobiazgu matematycznego. Zanim jeszcze poznaliśmy w szkole podstawowej ułamki, dzielenie liczb wykonywaliśmy „pod kreskę". Nad liczbą dzieloną zostawał wynik dzielenia, a na dole otrzymywaliśmy coś, co nazywało się resztą.

Właśnie owa reszta z dzielenia jest czymś, co tutaj i w wielu innych zagadnieniach programistycznych zajmuje bardzo ważne miejsce. Przypomnijmy sobie więc, jak to było…

10:3 = 3.3333…, ale równie dobrze 3 i reszty 1

Dlaczego właśnie 1?
Po pierwsze dlatego, że kiedy pomnożymy wynik dzielenia przez dzielnik, iloczyn będzie się różnił od liczby dzielonej właśnie o resztę (3*3 + 1 = 10).
Po drugie, ponieważ w liczbie 10 trójka „mieści się" 3 razy i zostaje jeszcze liczba 1.

Takie dzielenie z obcięciem reszty nazywane bywa dzieleniem całkowitym, a działanie dające w wyniku samą resztę z dzielenia (z pominięciem właściwego ilorazu) – resztą z dzielenia albo modulo.

W C++ do dzielenia całkowitego służy ten sam operator, co do dzielenia liczb rzeczywistych: `/`. Działa on jako operator dzielenia całkowitego wtedy, kiedy obydwa argumenty działania są typu całkowitego.
Reszta z dzielenia to działanie zdefiniowane tylko dla liczb całkowitych, któremu odpowiada w C++ operator `%`.

Zadanie 3[102]
Używając kalkulatora oblicz, ile będzie wynosiła reszta z dzielenia:
1. 100:10
2. 113:20
3. 512:65

[101] Dostępne cyfry to 0, 1 oraz 2. Po przejściu od 2 do 0 poprzednia cyfra zwiększa się o jeden. 0, 1, 2, 10, 11, 12, 20, 21, 22, 100, 101, 102, 110, 111, 112, 120, 121, 122, 200 itd.
[102] Wskazówka: wykonaj dzielenie, wynik z obciętą częścią ułamkową pomnóż przez dzielnik i odejmij ten iloczyn od liczby dzielonej. 1) 0; 2) 13; 3) 57; 4) 1

4. 666:7

Popracuj nad metodą obliczania tej reszty i spróbuj zauważyć pewne prawidłowości zachodzące w tym ciekawym działaniu.

Sposób postępowania

OK, pora teraz przejść do sedna sprawy. Naszym zadaniem będzie zakodowanie liczby 1984 w systemie siódemkowym, czyli znalezienie niewiadomej *x* w poniższym równaniu:

$$1984 = (x)_7$$

Algorytm postępowania jest bardzo prosty. Dzielimy liczbę przez podstawę systemu, następnie jako nową liczbę pod spodem zapisujemy wynik, a po prawej stronie zapisujemy resztę z dzielenia. Powtarzamy tą czynność do momentu, kiedy jako wynik z dzielenia otrzymamy 0.

1984 : 7	3
283 : 7	3
40 : 7	5
5 : 7	5
0	

Tabela 19. Kodowanie liczby całkowitej w systemie siódemkowym.

Postaraj się dobrze zrozumieć tą tabelkę. Zwróć też uwagę na ostatnią operację. 5 da się podzielić przez 7. Wtedy reszta wynosi 5, a wynikiem jest 0, co dopiero kończy obliczenia.

Teraz spisujemy cyfry w kolejności **od dołu do góry** i mamy gotowy wynik :DD

$$1984 = (5533)_7$$

Prawda, że to proste?

Może zdążyłeś już zwrócić uwagę na fakt, że reszta z dzielenia nigdy nie będzie większa niż liczba, przez którą dzielisz. Np. reszta z dzielenia przez 5 wynosi zawsze 0, 1, 2, 3 lub 4. Mówiąc ogólnie: x % n = 0, 1, …, n-1.
Dzięki temu reszty z dzielenia przez podstawę systemu możemy używać jako cyfry w tym systemie.

Ćwiczenia

Zadanie 4[103]

Zakoduj:

1. liczbę 13 w systemie czwórkowym
2. liczbę 64 w systemie jedenastkowym
3. liczbę 666 w systemie dziewiątkowym
4. liczbę $(FF)_{17}$ w systemie trójkowym

Kodowanie ułamków

Potrafimy już kodować liczby całkowite. Pora na opanowanie umiejętności kodowania ułamków.

[103] 1. $(31)_4$; 2. $(509)_{11}$; 3. $(820)_9$; 4. $(FF)_{17} = 270 = (101000)_3$

Algorytm postępowania jest bardzo podobny do zamiany liczb całkowitych. Tym razem jednak mnożymy liczbę przez podstawę systemu, jako nową liczbę pod spodem zapisujemy część ułamkową otrzymanego iloczynu (0.cośtam), natomiast część całkowitą (to, co w wyniku otrzymanym po pomnożeniu stało przed przecinkiem) zapisujemy po prawej stronie.

Zakodujmy tym razem liczbę 0.0415 w systemie dwudziestkowym!

0.0415 * 20	0
0.83 * 20	G (16)
0.6 * 20	C (12)
0.0	

Tabela 20. Kodowanie ułamka w systemie dwudziestkowym.

Uwaga! Podczas kodowania ułamków otrzymane cyfry spisujemy, odwrotnie niż w przypadku liczb całkowitych, **od góry do dołu**!

A więc $0.0415 = (0.0GC)_{20}$.

Tym razem obliczenia zakończyły się otrzymaniem po przecinku wyniku 0 (czyli otrzymaniem liczby całkowitej). Jednak nie zawsze musi tak być. Okazuje się, że liczba posiadająca w pewnym systemie skończone rozwinięcie (skończoną ilość cyfr po przecinku potrzebną do dokładnego zapisania tej liczby) w innym systemie może mieć rozwinięcie nieskończone. Otrzymywalibyśmy wtedy coraz to inne wyniki mnożenia (a może te same? wówczas mielibyśmy do czynienia z ułamkiem okresowym) i w końcu musielibyśmy ograniczyć się do pewnej ustalonej ilości cyfr po przecinku, żeby nie zaliczyć się na śmierć :)

Czy potrafisz znaleźć przynajmniej ogólny sposób szacowania, czy ułamek będzie miał w danym systemie skończone rozwinięcie?

Ćwiczenia

Zadanie 5[104]
Zakoduj:

1. liczbę 0.3333 w systemie trójkowym
2. liczbę 0.12 w systemie piątkowym
3. liczbę 0.777 w systemie piętnastkowym
4. liczbę 123.456 w systemie ósemkowym

Przelicz jedną z tych liczb z powrotem na system dziesiętny i sprawdź, jak duża niedokładność powstała w związku z obcięciem jej zakodowanej postaci do skończonej ilości cyfr po przecinku.

Algorytm Hornera – dla leniwych

Jeśli wykonałeś zadanie 5 (a na pewno wykonałeś – w końcu jesteś pilnym uczniem, który chce zostać dobrym koderem :) doszedłeś pewnie do wniosku, że zakodować trzeba było osobno część całkowitą i część ułamkową liczby z podpunktu 4. Czy nie istnieje prostszy sposób?

Okazuje się, że tak - nazywa się on algorytmem Hornera. Pozwala on za jednym zamachem zakodować liczbę rzeczywistą posiadającą zarówno część całkowitą, jak i ułamkową.

[104] 1. $(0.222222...)_3$; 2. $(0.03)_5$; 3. $(0.B9E959...)_{15}$; 4. $(173.351361...)_8$

Jest tylko jedno ograniczenie. Trzeba z góry określić ilość cyfr, na jakiej maksymalnie kodowali będziemy część ułamkową – czyli ilość cyfr po przecinku.

Mało brakowało, a zapomniałbym dodać jedną bardzo ważną, chociaż może oczywistą rzecz. W każdej liczbie zapisanej w każdym systemie możemy dopisywać dowolną ilość zer do części całkowitej (przed przecinkiem) po lewej stronie i do części ułamkowej (po przecinku) po prawej stronie, co nie zmieni nam wartości tej liczby. Np.:

$12.34 = 00012.3400$
$(2010.012)_3 = (002010.012000)_3$

Zakodujemy teraz liczbę 1984.0415 w systemie siódemkowym. Sposób postępowania jest następujący:

Przyjmujemy dokładność do 6 cyfr po przecinku. Następnie mnożymy naszą kodowaną cyfrę przez podstawę systemu podniesioną do potęgi takiej, ile cyfr ustaliliśmy.

$1984.0415 * 7^6 = 1984.0415 * 117\,649 = 233\,420\,498.5$

Uff… Tylko spokojnie, nie ma się czego bać. Kalkulator jest po naszej stronie :))

Wyszło coś wielkiego. Co dalej? Najpierw zauważmy, że otrzymana liczba zawiera część ułamkową. Niby nie ma w tym niczego nadzwyczajnego, ale tkwi w tym fakcie pewien szczegół. Otóż obecność w tym iloczynie części ułamkowej informuje nas, że danej liczby nie będzie się dało zakodować z wybraną dokładnością precyzyjnie – zostanie ona obcięta do wybranej ilości cyfr po przecinku.
Po przyjęciu tej informacji do wiadomości zaokrąglamy wynik do liczby całkowitej, a następnie kodujemy ją w wybranym systemie tak, jak koduje się zwyczajne liczby całkowite. Zatem do dzieła!

233 420 499 : 7	4
33 345 785 : 7	4
4 763 683 : 7	1
680 526 : 7	0
97 218 : 7	2
13 888 : 7	0
1984 : 7	3
283 : 7	3
40 : 7	5
5 : 7	5
0	

Tabela 21. Kodowanie liczby algorytmem Hornera.

Nie takie to straszne, jak mogłoby się wydawać. Spisujemy teraz cyfry **od dołu do góry**, tak jak podczas kodowania liczb całkowitych. Otrzymujemy takie coś: 5533020144.

Na koniec, zgodnie ze wstępnym założeniem, oddzielamy ostatnie 6 cyfr przecinkiem. Ostateczny wynik wygląda tak:

$1984.0415 = (5533.020144)_7$

Ćwiczenia

Pozostaje nam już tylko przećwiczyć przeliczanie liczb algorytmem Hornera…

Zadanie 6[105]
Zakoduj używając algorytmu Hornera:
1. liczbę 11.2222 w systemie trójkowym
2. liczbę 10.5 w systemie piątkowym
3. liczbę 0.0016 w systemie szesnastkowym
4. liczbę 2048.128 w systemie dziewiątkowym

Przelicz jedną z tych liczb z powrotem na system dziesiętny i sprawdź, jak duża niedokładność powstała w związku z obcięciem jej zakodowanej postaci do skończonej ilości cyfr po przecinku.

Podsumowanie

W ten oto sposób kończymy podrozdział poświęcony systemom liczbowym i przeliczaniu liczb. Mam nadzieję, że choć trochę poćwiczyłeś takie przeliczanie, posiadłeś umiejętności zamiany wszelkich liczb – małych i dużych – między dowolnymi systemami oraz dobrze się przy tym bawiłeś.
To była taka mała odskocznia od spraw ściśle związanych z komputerem. W następnym podrozdziale już do nich wrócimy.

Zanim jednak to nastąpi, radzę rozwiązać na koniec kilka zadań, które sprawdzą twoją wiedzę i umiejętności nabyte podczas lektury tego podrozdziału.

Zadanie 7[106]
1. Wyprowadź tabelkę dwudziestu pierwszych liczb systemu jedenastkowego.
2. Rozkoduj do systemu dziesiętnego liczbę $(GG.AG)_{18}$.
3. Ile będzie wynosiła reszta z dzielenia 5555 : 66 ?
4. Zakoduj dowolną metodą liczbę 2003.1214 w systemie czwórkowym z dokładnością do 10 cyfr po przecinku i rozkoduj ją z powrotem na system dziesiętny. Czy została zachowana dokładność? Po czym to można stwierdzić?

System binarny

Rozpoczynając kolejny podrozdział wracamy do tematu komputerów i programowania. Jak zapewne wiesz, komputer posługuje się systemem **dwójkowym**, czyli **binarnym**. Do zapisywania wszelkich informacji używa więc tylko dwóch cyfr: 0 i 1.
Poznawszy teorię dowolnych systemów liczbowych, skupimy się teraz na tych naprawdę ważnych z naszego punktu widzenia.

Zanim jednak to nastąpi, tym razem wyjątkowo – już na wstępie – proponuję rozwiązanie kilku zadań. Pozwolą nam one trochę „wczuć się w klimat" :)

Zadanie 8[107]
1. Wyprowadź tabelkę kilku pierwszych liczb systemu dwójkowego.
2. Ile będzie wynosiła reszta z dzielenia 25 : 2 ? Jaki jest prosty sposób na wyznaczanie takiej reszty?
3. Zakoduj liczbę 128 w systemie dwójkowym.
4. Rozkoduj liczbę $(010101100.1100)_2$

[105] 1. $(102.0200...)_3$; 2. $(20.2223...)_5$; 3. $(0.0069...)_{16}$; 4. $(2725.1133...)_9$
[106] 1) 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, 10, 11, 12, 13, 14, 15, 16, 17, 18; 2) 288+16+0.5555...+0.0493 = 304.6048; 3) 11; 4) $(133103.0133011001)_4$; dokładność została zachowana
[107] 1) 0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001 itd.; 2) 25%2=1, Sprawdzamy, czy liczba jest parzysta. 3) 10000000; 4) 172.75, zera na początku i na końcu są dla zmyły :)

Teoria

Na początek, jak zwykle, muszę podać trochę teorii. Na szczęście tym razem nie będzie wzorów. W zamian proponuję zapoznanie się z definicjami kilku (bardziej lub mniej związanych z omawianym tematem) pojęć. Chciałbym, żebyś poznał ich znaczenie i nauczył się je rozróżniać, ponieważ bardzo często są one mylone albo używane niepoprawnie.

Informacja – to konstatacja stanu rzeczy, wiadomość, powiadamianie społeczeństwa w sposób zobiektywizowany za pomocą środków masowego przekazu albo (co dla nas najbardziej odpowiednie) obiekt abstrakcyjny, który w postaci zakodowanej może być przechowywany, przesyłany, przetwarzany i użyty do sterowania.

Dane – to informacje wyrażone w pewnym języku. W informatyce są obiektami, na których operują programy.

W praktyce informacje można zdefiniować jako dane wraz ze sposobem ich interpretowania. Pojęcia te dotyczą nie tylko komputerów. Kiedy urzędnik przegląda tabelki z liczbami mówimy, że przegląda jakieś „dane liczbowe". Tymczasem dla niego te liczby niosą pewne „informacje".
„Dane" jest więc pojęciem ogólniejszym, który obejmuje informacje bez znanego lub istotnego w danym kontekście znaczenia ich treści. Niniejszy tekst niesie pewne informacje, ale kiedy nagrasz go na CD-ROM pośród innych plików powiesz, że jest tam „tyle a tyle megabajtów danych".

Kodowanie – to zapisywanie informacji w określony sposób.
Szyfrowanie – to kodowanie informacji w taki sposób, aby były nieczytelne dla osób niepowołanych, a więc utajnione.

Szyfrowanie jest więc tylko specjalnym rodzajem kodowania. Kodowaniem możemy nazwać każde zapisywanie informacji w jakiejś postaci, choćby tekstu w języku polskim. Każdy sposób zapisu możemy nazwać kodem. Mieliśmy już do czynienia z kodowaniem liczb w różnych systemach (czyli inaczej ich zapisywaniem) i bynajmniej nie robiliśmy tego po to, aby kodowane liczby stały się nieczytelne :)

Szum to sygnał, który nie niesie żadnych informacji.

Krótka podróż w czasie

Przenieśmy się teraz na chwilę do przeszłości celem zrozumienia, dlaczego właśnie taki, a nie inny system liczbowy jest podstawą całej informatyki.

Dawno, dawno temu odkryto elektryczność i zaczęto budować różne urządzenia. Prąd płynął sobie w przewodach – raz to mniejszy, innym razem większy – przenosząc sygnały radiowe, dźwiękowe czy obraz telewizyjny. Takie urządzenia nazywamy **analogowymi**.

Aż tu nagle stanęło przed maszynami trudne zadanie wykonywania obliczeń. Szybko okazało się, że w matematyce nie może być (tak jak jest np. w radiu) żadnych szumów ani trzasków. Liczby są liczbami i muszą pozostać dokładne.

Dlatego ktoś kiedyś wpadł na genialny pomysł, by w każdej chwili w przewodzie mógł być przenoszony tylko jeden z dwóch możliwych stanów: prąd nie płynie albo płynie pewien z góry ustalony, nie ma napięcia albo jest pewne określone napięcie, napięcie jest dodatnie albo ujemne itp. Te dwa stany można reprezentować przez dwie cyfry systemu binarnego: 0 oraz 1. Tak powstały urządzenia **cyfrowe**.

Często dopiero zmiana stanu jest informacją. Np. podczas budowy cyfrowego urządzenia elektronicznego można przyjąć taki kod, że stan identyczny z poprzednim oznacza 0, a stan odwrotny do poprzedniego oznacza 1.

No dobrze, ale właściwie co takiego genialnego jest w ograniczeniu się tylko do dwóch stanów i dlaczego wybrano właśnie dwa, a nie np. trzy albo dziesięć? Dzięki temu osiągnięto m.in. dwie istotne cechy urządzeń cyfrowych:

- **Prostota** – elementy wykonujące operacje numeryczne na dwóch możliwych stanach budować jest najprościej.
- **Wierność kopii** – przesyłanie oraz kopiowanie danych nie powoduje utraty jakości (w przypadku sygnałów, np. dźwięku) ani dokładności (w przypadku liczb).

System binarny

Jeśli wykonałeś ostatnie zadanie zauważyłeś może, że operowanie na liczbach w systemie dwójkowym jest dużo prostsze, niż w wypadku innych systemów.
Aby jeszcze lepiej pokazać ten fakt, wykonajmy razem dwa przekształcenia.

Zakodujemy w systemie binarnym liczbę 1984.

1984 : 2	0
992 : 2	0
496 : 2	0
248 : 2	0
124 : 2	0
62 : 2	0
31 : 2	1
15 : 2	1
7 : 2	1
3 : 2	1
1 : 2	1
0	

Tabela 22. Kodowanie liczby w systemie binarnym.

A więc 1984 = $(11111000000)_2$.

Jest przy tym trochę więcej pisania, ale za to przy odrobinie wprawy dzielenie dowolnie dużych liczb przez 2 można wykonywać w pamięci, a o reszcie z tego dzielenia (równej zawsze 0 albo 1) świadczy parzystość dzielonej liczby.

Teraz rozkodujmy liczbę $(1011.0101)_2$.

$$2^3 + 2^1 + 2^0 + 2^{-2} + 2^{-4} = 8 + 2 + 1 + 0.5 + 0.0625 = 11.5625$$

Jak widać, nie trzeba pisać przed każdą rozpisywaną cyfrą odpowiednio 0* lub 1*. Wystarczy tylko spisać te potęgi dwójki, którym odpowiada cyfra 1 i pominąć te, którym odpowiada cyfra 0.

Zadanie 9[108]

1. Zakoduj liczbę 255 w systemie dwójkowym.
2. Zakoduj liczbę 0.5 w systemie dwójkowym.

[108] 1) $(11111111)_2$; 2) $(0.1)_2$; 3) 83; 4) 60

3. Rozkoduj liczbę $(01010011)_2$.
4. Rozkoduj liczbę $(111100)_2$.

Czy potrafisz wykonywać większość potrzebnych operacji w pamięci?

Dodawanie i odejmowanie

Czeka nas teraz kolejna powtórka z pierwszych klas szkoły podstawowej. Przypomnimy sobie bardzo dokładnie, jak wykonywało się dodawanie i odejmowanie „pod kreskę". Przypomnienie zrobimy na normalnych liczbach w systemie dziesiętnym, by następnie nauczyć się tych samych operacji dla liczb binarnych.

Dodawanie i odejmowanie liczb dziesiętnych

Jako pierwsze wykonajmy proste dodawanie dwóch liczb: 163 + 82 = 245.

	1	6	3
+		8	2
=	2	4	5

Tabela 23. Dodawanie pod kreskę.

Zaczynając od prawej strony dodajemy 3+2=5. Następnie dodajemy 6+8=14. Występuje tu tzw. przepełnienie. W takiej sytuacji jako wynik danej kolumny zapisujemy ostatnią cyfrę sumy (czyli 4), a poprzednią przenosimy na kolejną kolumnę na lewo. Stąd 1+1=2.

Teraz zajmiemy się rzeczą trochę trudniejszą – odejmowaniem. Odejmiemy 701326 – 29254 = 672072.

	7	0	1	3	2	6
-		2	9	2	5	4
=	6	7	2	0	7	2

Tabela 24. Odejmowanie pod kreskę.

Znowu zaczynając od prawej strony odejmujemy 6-4=2. Potem próbujemy odjąć 2-5. Ponieważ nie da się wykonać tego na liczbach dodatnich, dokonujemy tzw. pożyczki – pożyczamy jednostkę z liczby następnej (3 zamieni się w 2). Ta jednostka po przejściu do naszej kolumny zamienia się w dziesiątkę (to chyba oczywiste, dlaczego właśnie dziesiątka? :) i stąd 10+2-5 = 12-5 = 7. Z trójki po pożyczeniu została dwójka, a 2-2=0.

Dalej sytuacja jest jeszcze bardziej skomplikowana. Znowu musimy dokonać pożyczki, bo nie da się odjąć 1-9. Tym razem jednak nie ma od kogo pożyczyć w następnej kolumnie – stoi tam 0! Pożyczamy więc od stojącej dwie kolumny dalej siódemki. Pożyczona jednostka zamienia się w poprzedniej kolumnie (tej nad zerem) w dziesiątkę. Z tej dziesiątki dalej pożyczamy jednostkę, która zamienia się w kolejną dziesiątkę. Z siódemki została więc szóstka, zamiast zera jest dziewięć, a my możemy wreszcie policzyć 11-9=2.
Dalej jest już prosto, o ile pamiętamy, co gdzie zostało. 9-2=7, a 6-0=0.

Mam nadzieję, że przypomniałeś sobie sposób wykonywania dodawania i odejmowania pod kreskę oraz w pełni rozumiesz, jak to się robi. Szczególnie dużo trudności sprawiają pożyczki podczas odejmowania, dlatego na to szczególnie uczulam.

Dodawanie i odejmowanie liczb binarnych

Pora przejść na system dwójkowy. Dodawali będziemy zera i jedynki, ale 1+1=2. Jak tą dwójkę zapisywać? Zapisywać nigdzie jej nie trzeba. Ona będzie występowała tylko w

pożyczkach i przeniesieniach, a jej kodowanie jako 2 lub jako $(10)_2$ to kwestia mało ważna.

Od tej chwili darujemy sobie czasami zapisywanie liczb w nawiasach i z indeksem pamiętając, że zajmujemy się systemem dwójkowym.

Dodajmy dwie liczby binarne: 101011 + 01000 = 110011.

	1	0	1	0	1
+		0	1	0	0
=	1	1	0	0	1

Tabela 25. Dodawanie liczb binarnych.

W zasadzie nie ma tutaj żadnej wielkiej filozofii. 1+0=1, 0+0=0. Dopiero w trzeciej (od prawej strony) kolumnie występuje przeniesienie: 1+1=2, czyli $(10)_2$. Dlatego w tej kolumnie zapisujemy 0, a w następnej 1. W końcu 1 + domyślne 0 = 1.

Z odejmowaniem też jest podobnie. Odejmijmy 1001011 – 010110 = 110101.

	1	0	0	1	0	1	1
-		0	1	0	1	1	0
=		1	1	0	1	0	1

Tabela 26. Odejmowanie liczb binarnych.

Po kolei: 1-0=1, 1-1=0. Dalej nie możemy odjąć 0-1, dokonujemy więc pożyczki sąsiedniej jedynki. Tam zostaje zero, a pożyczona jedynka zamienia się na (no – zgadnij! :) dwójkę. Stąd 2-1=1. Z jedynki w czwartej kolumnie zostało zero. 0-0=0.

Dalej znowu musimy pożyczyć. Ponieważ obok nie ma nikogo skłonnego do wypożyczenia potrzebnej nam jedynki, szukamy nieco dalej. Tamta jedynka z ostatniej kolumny pożycza nam swoją jedyną jedynkę zostawiając sobie zero. Jedynka przechodzi do kolumny przedostatniej stając się dwójką, z której dalej pożyczamy jedynkę. W kolumnie przedostatniej zostaje jedynka, a my możemy wreszcie odjąć 2-1=1.

Pamiętając o tym, co zostało w dwóch ostatnich kolumnach, kończymy działanie wykonując 1-0=1 oraz 0 – domyślne 0 = 0, którego też nie musimy zapisać.

W ten oto sposób opanowaliśmy umiejętność wykonywania podstawowych operacji arytmetycznych na liczbach dwójkowych, czyli na tych słynnych komputerowych zerach i jedynkach :)
Teraz, jak się zapewne domyślasz, pora na…

Ćwiczenia

Zadanie 10[109]
Oblicz:

1. 111 + 111
2. 11001010 + 10101100
3. 111111 – 1101
4. 10000 – 1101

Zamień liczby z jednego z podpunktów na dziesiętne i sprawdź, czy otrzymałeś prawidłowy wynik.

[109] 1) Wskazówka: $3=(11)_2$, 1110; 2) 101110110; 3) 110010; 4) 11

System ósemkowy i szesnastkowy

W całej swej „fajności” system binarny ma jedną wielką wadę, którą z całą pewnością zdążyłeś już zauważyć. Mianowicie liczby w tym systemie są po prostu długie. Do zapisania każdej liczby potrzeba wielu cyfr – dużo więcej, niż w systemach o większej podstawie.
W sumie nie ma w tym niczego dziwnego – w końcu to jest system o najmniejszej możliwej podstawie. Czy nie da się jednak czegoś na to poradzić?

Rozwiązaniem są dwa inne systemy liczbowe, które również mają duże znaczenie w informatyce. Są to system ósemkowy oraz przede wszystkim system szesnastkowy.

Dlaczego właśnie one są takie ważne? Nietrudno zauważyć, że 8 i 16 to odpowiednio trzecia i czwarta potęga dwójki. Co z tego wynika?

Okazuje się, że każdym trzem cyfrom systemu binarnego odpowiada jedna cyfra systemu ósemkowego, a każdym czterem cyfrom systemu binarnego odpowiada jedna cyfra systemu szesnastkowego.

System dziesiętny oraz większość pozostałych nie posiada tej cennej właściwości. Zapewne dlatego, że ich podstawy nie są potęgami dwójki.

Dzięki temu można sporządzić tabelkę wszystkich cyfr danego systemu i ich binarnych odpowiedników oraz używać jej do prostej zamiany dowolnie długich liczb! Utworzenie takiej tabelki z pewnością nie sprawiłoby ci problemu. Oto ona:

0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Tabela 27. Binarne odpowiedniki cyfr szesnastkowych.

Możemy teraz przeliczyć liczbę 00110101 na system szesnastkowy. W tym celu grupujemy cyfry po cztery, a następnie korzystając z tabelki zamieniamy je na cyfry szesnastkowe.

$$00110101 = 0111\ 0101 = (75)_{16}$$

A teraz odwrotnie:

$$(ABCD)_{16} = 1010\ 1011\ 1100\ 1101 = 1010101111001101$$

Bardzo proste! Rodzi się tylko jednak pytanie: czy musisz tą tabelkę znać na pamięć? W zasadzie wypadałoby znać, ale w rzeczywistości nie trzeba jej wkuwać.

Jeśli pilnie rozwiązywałeś wszystkie powyższe zadania, powinieneś umieć szybko wyprowadzić sobie każdą potrzebną liczbę licząc po kolei liczby dwójkowe. Spójrz jeszcze raz na te zera i jedynki w powyższej tabelce i spróbuj zauważyć pewne prawidłowości w ich rozkładzie. Każdy może znaleźć swój sposób na jej zapamiętanie.

Z czasem nabędziesz wprawy i większość operacji wykonasz zawsze w pamięci. Pomoże w tym pamiętanie wag kolejnych cyfr w systemie binarnym. Oczywiście – znajomość na

pamięć kolejnych potęg dwójki jest **obowiązkowa** dla każdego programisty!!!

Oto najważniejsze z nich (tych większych nie musisz wkuwać, ale przynajmniej się z nimi „opatrz" :)
1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192 itd.
2^{15} = 32 768 (ok. trzydzieści dwa tysiące)
2^{16} = 65 536 (ok. sześćdziesiąt pięć tysięcy)
2^{31} = 2 147 483 648 (ok. dwa miliardy)
2^{32} = 4 294 967 296 (ok. cztery miliardy)

Ja wcale nie przesadzam. Te potęgi dwójki i system szesnastkowy są naprawdę aż tak ważne w informatyce i jako programista, nawet używający języków najwyższego poziomu, musisz sprawnie się nimi posługiwać!

Na zakończenie wspomnę jeszcze o przykładowych zastosowaniach:

- Do zapisywania kolorów w Win32API i DirectX używa się liczb szesnastkowych.
- Do zapisywania adresów komórek pamięci używa się liczb szesnastkowych.
- Do zapisywania atrybutów plików w systemie Linux i na serwerach FTP używa się liczb ósemkowych.

…oraz wymienię kilka ważnych wartości dla 8-cyfrowych liczb binarnych, a także ich odpowiedniki dziesiętne i szesnastkowe (także wypadałoby się ich nauczyć :/)

- $00000000 = (00)_{16} = 0$
- $01000000 = (40)_{16} = 64$
- $01111111 = (7F)_{16} = 127$
- $10000000 = (80)_{16} = 128$
- $11000000 = (C0)_{16} = 192$
- $11111111 = (FF)_{16} = 256$

System dwójkowy to, jak już wiesz, inaczej system binarny (w skrócie *bin*).
System ósemkowy to inaczej system oktalny (w skrócie *oct*).
Nasz normalny system dziesiętny to inaczej system decymalny (w skrócie *dec*).
System szesnastkowy to inaczej system heksadecymalny (w skrócie *hex*).

Zadanie 11

1. Wyprowadź na kartce 16 pierwszych liczb binarnych.
2. Dopisz do nich odpowiadające cyfry szesnastkowe.
3. Postaraj się wypisać z pamięci jak najwięcej kolejnych potęg dwójki.
4. Podziel 256:2, 256:4, 256:8 itp. Wyniki zamień na liczby binarne i szesnastkowe.
5. Zastanów się, jak do zapisywania jakich informacji wystarczy, a dla jakich za mała jest liczba binarna 8-, 16-, 32-cyfrowa?

Zadanie 12[110]
Zamień:

1. liczbę $(776)_8$ na system binarny
2. liczbę $(B01A)_{16}$ na system binarny
3. liczbę 11010010110 na system szesnastkowy
4. liczbę 205 na system binarny, ósemkowy i szesnastkowy

Zastanów się jeszcze raz nad metodami, których użyłeś. Jak wiele operacji udało ci się przeprowadzić w pamięci? Jak szybko dokonujesz przekształceń? Czego powinieneś jeszcze się douczyć?

[110] 1) 111 111 110; 2) 1011 0000 0001 1010; 3) Wskazówka: liczbę trzeba uzupełnić z lewej strony zerem. $(696)_{16}$; 4) 11001101 = $(315)_8$ = $(CD)_{16}$

Bonus

Na zakończenie będzie mały bonus. Pokażę teraz jeszcze jeden, być może najprostszy sposób przeliczania liczb dwójkowych na dziesiętne w obydwie strony.

Każdą liczbę można w dokładnie jeden sposób przedstawić jako kombinację wag cyfr danego systemu (tutaj – dwójkowego) pomnożonych przez cyfrę (u nas to nie ma znaczenia – cyfrą jest 0 albo 1).
Przykładowo liczbę 205 z poprzedniego zadania można zamienić w taki sposób:

- W liczbie 205 liczba 128 (waga ósmej cyfry) mieści się raz. Piszemy wobec tego 1. 205-128=77.
- W liczbie 77 liczba 64 (waga siódmej cyfry) mieści się raz. Piszemy 1. 77-64=13.
- W liczbie 13 liczba 32 (waga kolejnej cyfry) nie mieści się ani razu. Piszemy 0.
- Liczba 16 też się w niej nie mieści – piszemy kolejne 0.
- Liczba 8 mieści się raz – piszemy jedynkę. 13-8=5.
- W liczbie 5 liczba 4 (waga trzeciej cyfry) mieści się raz. Piszemy 1. 5-4=1.
- W liczbie 1 liczba 2 (waga drugiej cyfry) nie mieści się. Piszemy zero.
- Wreszcie zostaje nam jedynka (waga pierwszej cyfry), która mieści się w liczbie 1 dokładnie 1 raz – piszemy kończącą jedynkę.

W ten sposób otrzymujemy: 205 = 11001101.

W drugą stronę też możemy łatwo przeliczać. Warunkiem jest znajomość potęg dwójki. Widząc liczbę 11010011, sumujemy wagi tych cyfr, przy których stoi jedynka. Zresztą... chyba była już o tym mowa kilka stron wcześniej :)

1 + 2 + 16 + 64 + 128 = 211

Zadanie 13[111]
Którą potęgą dwójki jest liczba:
1. 16
2. 64
3. 256
4. 1024

Czy potrafisz podać odpowiedzi od razu?

Jednostki informacji

Komputer przechowuje dane w postaci liczb binarnych. Zachodzi potrzeba mierzenia ilości tych danych. Powstały w tym celu specjalne jednostki. Zacznijmy ich omawianie od początku.

Pojedyncza cyfra systemu dwójkowego – mogąca przechowywać informację o jednym z dwóch możliwych stanów – przyjmująca wartości oznaczane jako 0 albo 1, będąca najmniejszą i niepodzielną jednostką informacji cyfrowej – to **bit**.
1 b (mała litera „b") /bit/

Jeden bit wystarczy do zapisania np. informacji o płci osoby w bazie danych. Przykładowo 0 mogłoby oznaczać kobietę, a 1 mężczyznę (tylko bez obrazy, miłe panie ;) To jednak stanowczo za mało do zapisywania większości spotykanych informacji.

[111] 1) 4; 2) 6; 3) 8; 4) 10

Bity grupuje się. 8 bitów tworzy **bajt**. Bajt jest podstawową jednostką informacji cyfrowej.
1 B = 8 b (duże „B”) /bajt/

Jak nietrudno obliczyć, jeden bajt może znajdować się w jednym z 2^8=256 stanów. To wystarczająco dużo, by zapamiętać np. wiek osoby w bazie danych. Nikt raczej nie dożywa wieku większego, niż 255 lat :)

To również wystarczająco dużo, by każdemu stanowi (zamiast liczby) przyporządkować pewien znak. Oprócz dużych i małych liter, cyfr, znaków przestankowych, wszystkich znaków alfanumerycznych znajdujących się na klawiaturze, kilku znaków sterujących i innych znalazłoby się jeszcze miejsce dla różnych dziwnych symboli.

Podobnie jak w wypadku jednostek fizycznych, możemy tworzyć wielokrotności jednostek informacji.

Małe przypomnienie z fizyki:

podwielokrotność	***przedrostek***	***symbol***	***wielokrotność***	***przedrostek***	***symbol***
10^{-18}	atto	a	10^{1}	deka	da
10^{-15}	femto	f	10^{2}	hekto	h
10^{-12}	piko	p	10^{3}	kilo	k
10^{-9}	nano	n	10^{6}	mega	M
10^{-6}	mikro	μ	10^{9}	giga	G
10^{-3}	mili	m	10^{12}	tera	T
10^{-2}	centy	c	10^{15}	peta	P
10^{-1}	decy	d	10^{18}	eksa	E

Tabela 28. Podwielokrotności i wielokrotności jednostek podstawowych.

Ponieważ bit jest niepodzielny, nie istnieją podwielokrotności jednostek informacji. Istnieją jednak wielokrotności.

Tkwi tu jednak pewna różnica. Wielokrotności jednostek w fizyce są potęgami dziesiątki (wybieranymi, dla większych wielokrotności, co 3). Oczywiście ma to związek z podstawą naszego systemu. Czy w informatyce nie powinniśmy zatem brać potęg dwójki?

Faktycznie, w powszechnym użyciu są wielokrotności jednostek informacji będące potęgami dwójki. Pewną analogię do wielokrotności jednostek fizycznych pozwala zachować właściwość mówiąca, że $2^{10} = 1024 \approx 1000$.

- 1 kB = 2^{10} B = 1024 B ≈ tysiąc (małe „k”) /kilobajt/
- 1 MB = 2^{20} B = 1024 kB = 1048576 B ≈ milion (duże „M”) /megabajt/
- 1 GB = 2^{30} B = 1024 MB = 1073741824 B ≈ miliard (duże „G”) /gigabajt/
- 1 TB = 2^{40} B = 1024 GB = 1099511627776 B ≈ bilion (duże „T”) /terabajt/
- 1 PB = 2^{50} B = 1024 GB = 1125899906842624 B ≈ biliard (duże „P”) /petabajt/
- 1 EB = 2^{60} B = 1024 PB = 1152921504606846976 B ≈ trylion (duże „E”) /eksabajt/

Pojawia się teraz pytanie: „Jak dużo to jest?”. Spróbuję na nie odpowiedzieć podając typowe rozmiary kilku przykładowych rodzajów danych:

- **Tekst** – zależnie od długości jeden dokument zajmuje kilkadziesiąt lub kilkaset kB.
- **Grafika** – zależnie od wielkości i formatu zapisu jeden obrazek zajmuje kilkadziesiąt lub kilkaset kB, a nawet ponad 1 MB.

- **Muzyka** – zależnie od długości jeden utwór zajmuje kilka MB. Jedna minuta muzyki w formacie MP3 to ok. 1 MB.
- **Programy** – w zależności od wielkości pełne wersje programów zajmują od kilku do kilkuset MB.
- **Film** – w zależności od długości i jakości kodowania jeden film zajmuje od kilkuset MB do ponad 1 GB.

Podam też pojemności przykładowych nośników danych:

- **Dyskietka 3.5"** – 1.44 MB
- **Pamięć typu flash** – zależnie od ceny kilkadziesiąt lub kilkaset MB, a nawet ponad 1 GB
- **CD-ROM** – 650 lub 700 MB
- **DVD-ROM** – w zależności od rodzaju od kilku do kilkunastu GB.
- **Dysk twardy** – kilkadziesiąt lub ponad 100 GB.
- **Wszystkie dane udostępniane w sieci P2P** – kilka PB.

Idealiści chcieli wylansować nowe nazwy dla wielokrotności bajta – „kibibajt", „mebibajt", „gibibajt", „tebibajt" itd. oraz oznaczenia KiB, MiB, GiB itd., natomiast przez kilobajt, megabajt, gigabajt itd. chcieli oznaczać wielokrotności będące potęgami dziesiątki, nie dwójki – tak jak jest w fizyce.

Przyzwyczajenia wzięły jednak górę nad teoriami i oprócz nielicznych programów, które nazywają jednostki informacji w ten oryginalny sposób, wszyscy mówią i piszą o kilobajtach i megabajtach mając na myśli potęgi dwójki.

Niektórzy oznaczają kilobajty przez duże K. Ma to podkreślać inne znaczenie tej wielokrotności, niż tradycyjnego „kilo". Nie ma chyba jednak uzasadnienia dla takiego wybiórczego odróżniania i dlatego osobiście zalecałbym używanie normalnych kilobajtów przez małe „k", a także megabajtów, gigabajtów itd. jako potęg dwójki.

Trzeba pamiętać, że 1024 to nie jest 1000 i nie zawsze można pominąć tą na pozór niewielką różnicę. Przykładowo uzbierawszy 700000000 B danych chcemy nagrać CD-ROM. Tymczasem w rzeczywistości jest to tylko 667 MB i zmieszczą się nam na płycie jeszcze 33 MB!

Dlatego trzeba uważać na tą różnicę i zawsze pamiętać, w jakiej wielokrotności wyrażony jest rozmiar danych. Jeśli używasz Total Commandera i pokazuje on wielkość plików w bajtach, zawsze kliknij prawym klawiszem na plik i wybierz *Właściwości*, aby zobaczyć jego faktyczny rozmiar w odpowiedniej wielokrotności.

Uważaj też na wielkości podawane przez różnych autorów i producentów, np. rozmiary plików czy pojemności dysków twardych. Niektórzy cwaniacy podają liczbę miliardów bajtów, zamiast prawdziwych gigabajtów.

Oprócz rozmiaru danych można jeszcze mierzyć szybkość ich przesyłania (np. przez Internet). Wielkość taką wyrażamy w kilobajtach lub też w kilobitach na sekundę, co zapisujemy odpowiednio jako: kBps lub kB/s oraz kbps lub kb/s.

1 kB/s = 8 kbps

Zadanie 14

1. Sprawdź, jak duży jest katalog główny twojego systemu operacyjnego.
2. Sprawdź, jak duże różnice występują w wielkościach katalogów z różnymi zainstalowanymi programami.
3. Jaki duży jest katalog z twoimi dokumentami?
4. Jak duże są pojedyncze pliki różnego rodzaju w katalogu z twoimi dokumentami?

5. Spróbuj policzyć, ile danych mniej więcej posiadasz zgromadzonych na wszystkich CD-ROMach? W jakiej wielokrotności wyrazisz tą wielkość?
6. Ile mogłyby zajmować wszystkie napisane przez ciebie programy zebrane w jednym miejscu?
7. Przemyśl: Czy rozmiar danych zawsze idzie w parze z ich wartością materialną albo moralną? Jeśli nie, to co o nim decyduje?

Liczby naturalne

Zaczynamy nareszcie omawianie spraw prawdziwie komputerowych. Rozpoczniemy od dokładnego opisania sposobu, w jaki komputer przechowuje w pamięci najprostsze możliwe liczby – liczby naturalne.

Liczby naturalne, jak pamiętamy ze szkoły, to liczby całkowite nieujemne, a więc 0, 1, 2, 3, 4, 5, 6, …, 100 itd.

Reprezentacja liczby naturalnej

W sposobie przechowywania liczby naturalnej w pamięci komputera nie ma żadnej wielkiej filozofii. Przechowywana jest dokładnie tak, jak zapisalibyśmy ją w systemie dwójkowym. Każdy bit stanowi pojedynczą cyfrę w tym systemie.

Poświęcając na przechowywanie liczby określoną ilość miejsca w pamięci ograniczamy zakres, jaki może przyjmować ta liczba. Jest to inaczej ilość możliwych kombinacji bitów. Przykładowo dla 1 bajta (8 bitów) możliwych liczb będzie:

$2^8 = 256$

Przyjmując za pierwszą liczbę 0, otrzymujemy zakres od 0 do 255. Zakresem jest więc zawsze liczba możliwych kombinacji bitów (czyli 2 do potęgi równej liczbie bitów) pomniejszona o jeden.

Można wyrysować tabelkę z bitami i odpowiadającymi im wagami. Wszystko to wygląda dokładnie tak samo, jak podczas omawiania zwykłych liczb binarnych.

bit	7	6	5	4	3	2	1	0
waga	128	64	32	16	8	4	2	1

Tabela 29. Budowa liczby naturalnej.

Kolejne bity ponumerowałem (od 0) jedynie dla czytelności. Wagi są, jak widać, kolejnymi potęgami dwójki.

Przykładowo liczbę 00110001 możemy rozkodować sumując wagi tych bitów, którym odpowiada jedynka.

32 + 16 + 1 = 49

Aby zakodować liczbę wybieramy te wagi, których suma będzie równa danej liczbie.

55 = 32 + 16 + 4 + 2 + 1 = 00110111

Zadanie 15[112]

1. Ilu bitów potrzeba do zapisywania liczb z zakresu 0…65535?
2. Rozkoduj jednobajtową liczbę 11111111.
3. Zakoduj liczbę 257 na 10 bitach. Czy ten przykład jest dla ciebie prosty?
4. Ilu bajtów potrzeba do zapisywania liczb z zakresu 239…255?

Liczby naturalne w programowaniu

Poznamy teraz praktyczne sposoby implementacji liczb naturalnych w języku C++.

Typy liczb naturalnych

Przyjęto nazwy na kilka typowych rodzajów liczb naturalnych różniących się ilością wykorzystywanych bajtów (i co za tym idzie – zakresem). Mają one swoje odpowiedniki pośród typów danych w języku C++. Każdy z nich ma po dwie nazwy – jedną standardową i drugą (krótszą) zadeklarowaną w plikach nagłówkowych Windows.

Po pierwsze, mamy jeden bajt (ang. *byte*), czyli 8 bitów. Odpowiada mu typ `unsigned char`, albo inaczej `BYTE`. Jego zakres to 0…255.

Po drugie, może być typ dwubajtowy, czyli 16-bitowy. Dwa bajty nazywane są słowem (ang. *word*). Stąd nazwy typów: `unsigned short` albo `WORD`. Zakres takiej liczby to 0…65 535 (65 tysięcy).

Dalej mamy liczbę 32-bitową, czyli zajmującą 4 bajty pamięci. Nazywana bywa ona podwójnym słowem (ang. *double word*). Odpowiadające typy w języku C++ to `unsigned long` oraz `DWORD`. Zakres wynosi 0…4 294 967 295 (4 miliardy).

Jest też typ, którego rozmiar zależy od tego, czy używany kompilator jest 16-, czy 32-bitowy. Nazywa się `unsigned int` albo `UINT`. W praktyce jednak kompilatory we współczesnych systemach operacyjnych (w tym Windows i Linux) są 32-bitowe, a więc `UINT` jest tak naprawdę tożsamy z `DWORD` – ich rozmiary w pamięci i zakresy są jednakowe.

Od czasu do czasu słyszy się głosy, jakoby pojęcie „słowa” (ang. *word*) oznaczało ten właśnie rozmiar zależny od platformy. W rzeczywistości jednak określenie to pozostało synonimem dwóch bajtów (16 bitów).

To jeszcze nie koniec. Dla tych, którym nie wystarcza zakres 4 miliardów, Microsoft przygotował w swoim kompilatorze dodatkowy typ: `unsigned __int64`. Jak sama nazwa wskazuje, liczby tego typu zajmują 64 bity, czyli 8 bajtów. Ich zakres to 0…18 446 744 073 709 551 615 (18 trylionów).

Dodatkowe słowo `unsigned` przed każdym z typów oznacza „bez znaku” i ma podkreślać, że chodzi nam o liczby naturalne (zawsze dodatnie). O liczbach ze znakiem (+ lub -), czyli o liczbach całkowitych, będzie mowa w następnym podrozdziale.

To nie pomyłka, że typ `char` występuje tu w roli typu liczbowego. Mimo docelowego przeznaczenia do przechowywania znaków tekstowych jest to właściwy typ liczby 8-bitowej, jako że C++ nie wprowadza rozróżnienia pomiędzy taką liczbą a znakiem. Będzie o tym mowa dokładnie w ostatnim podrozdziale.

[112] 1) 16b; 2) 255; 3) 0100000001; 4) Potrzebujemy 255-239=16 różnych kombinacji; $16=2^4$; potrzeba więc 4b = 0.5B; nikt nie powiedział, że zakres musi być od zera, ani że nie można mówić o połowie bajta :P

Zapisywanie wartości liczb naturalnych

Po zdefiniowaniu zmiennej danego typu, wcześniej czy później należałoby przypisać jej konkretną wartość liczbową. Język C++ pozwala na zapisywanie wartości (są to tzw. stałe dosłowne) w trzech systemach. Nie ma pośród nich systemu binarnego – liczby w tym systemie byłyby zbyt długie, a my przecież znamy inne, fajniejsze systemy :)

Można napisać liczbę naturalną ot tak po prostu. Zostanie ona potraktowana dosłownie jako liczba w systemie dziesiętnym i zadziała dla każdego z wyżej wymienionych typów, o ile nie przekracza jego zakresu.

```
BYTE n1 = 255;
WORD n2 = 7;
DWORD n3 = 0;
```

Jeśli zapisywaną liczbę poprzedzimy zerem, zostanie potraktowana jako liczba w systemie ósemkowym, np.:

```
BYTE n1 = 0377;
WORD n2 = 07;
DWORD n3 = 00;
```

Z kolei aby zapisać liczbę w naszym upragnionym i najpożyteczniejszym systemie szesnastkowym ;) należy poprzedzić ją znakami `0x` (zero i iks). Wtedy już można jej dalszą część zaczynać od dowolnej liczby zer bez żadnych konsekwencji. Litery A…F stosowane w roli cyfr mogą być zarówno duże, jak i małe.

```
BYTE n1 = 0xFF;
WORD n2 = 0x07;
DWORD n3 = 0x0;
```

Na to poprzedzanie liczby zerem trzeba uważać. Czasami chciałoby się wyrównać liczby różnej długości do jednej kolumny. Pamiętaj, by zawsze wstawiać w wolnych miejscach spacje, a nie zera. Inaczej liczba zostanie potraktowana jako zapisana w systemie ósemkowym i może mieć inną niż oczekiwana wartość!

Można też dodać na końcu liczby dużą literę `L`, by wymusić typ `long` oraz `u`, by podkreślić brak znaku (czyli że jest to liczba naturalna, a nie ogólnie całkowita). Połączenie tych dwóch liter – czyli dodanie na końcu liczby `uL`, wymusza jej potraktowanie jako wartości typu `unsigned long`. W praktyce rzadko (jeśli w ogóle) zachodzi potrzeba używania tych przyrostków.
W przypadku takich przyrostków wyjątkowo nie ma znaczenia wielkość liter.

Działania na liczbach naturalnych

Liczby naturalne można dodawać (+), odejmować (-), mnożyć (*) i dzielić (/). Operator dzielenia zastosowany na dwóch liczbach naturalnych da wynik również naturalny, a reszta z dzielenia zostanie obcięta.

Zadanie 16[113]
Jakie wartości będą miały zmienne po zainicjalizowaniu:

```
BYTE  n1 = 2+2*2;
```

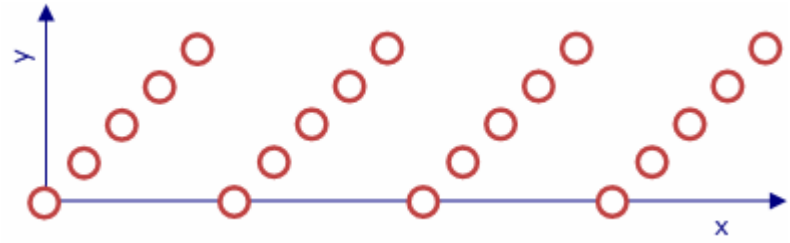
[113] n1 = 2 + (2*2) = 2+4 = 6; n2 = $(FF)_{16}$ / $(F)_{16}$ + 2 = 255 / 15 + 2 = 17 + 2 = 19; n3 = 10 + $(10)_8$ + $(10)_{16}$ + $(10)_{16}$ = 10 + 8 + 16 + 16 = 50; n4 = [$(A)_{16}$ / $(2)_8$ + 100] * $(3)_8$ = (10 / 2 + 100) * 3 = (5 + 100) * 3 = 105 * 3 = 315 > 255, Błąd: liczba poza zakresem!

```
WORD  n2 = 0xff / 0x0F + 2;
DWORD n3 = 10 + 010 + 0x10 + 0x010;
BYTE  n4 = (0x0A / 02 + 100) * 03;
```

Reszta z dzielenia

Skoro dzielenie w zbiorze liczb naturalnych obcina resztę, potrzebne jest działanie zwracające tą resztę. Działanie takie istnieje i oznacza się je w C++ symbolem procenta: `%`. Dzięki swoim ciekawym cechom oraz ogromnemu zastosowaniu w programowaniu zasługuje na poświęcenie mu osobnego punktu.

Wstępne wiadomości o reszcie z dzielenia były już wprowadzone wcześniej. Teraz zajmiemy się nim jeszcze bardziej szczegółowo. Możemy nawet sporządzić wykres tej funkcji:



Wykres 3. Wykres funkcji naturalnej y = x % 5

Z rysunku można wywnioskować, że funkcja f(x) = x % c jest okresowa o okresie c i przyjmuje kolejne wartości naturalne z zakresu 0…c-1.

Zliczanie

Można to wykorzystać przy wielu okazjach. Jedną z najczęściej spotykanych jest wszelkiego rodzaju zliczanie. Wyobraź sobie, że pewien licznik odlicza w górę zwiększając wartość pewnej zmiennej `L` o jeden. Jeśli chcesz, by jakaś akcja była wykonywana tylko co szósty cykl licznika, napisz:

```
if (L % 6 == 0)
   RobCos();
```

W taki wypadku pierwsze wywołanie funkcji nastąpi już na samym początku – kiedy zmienna `L` jest równa 0. Możesz to zmienić porównując otrzymaną resztę z dzielenia z wartością większą od zera, np.:

```
if (L % 6 == 1)
   RobCos();
```

Wtedy funkcja `RobCos()` wykona się dla `L` = 1, 7, 13, 19, 25 itd.

Losowanie liczb naturalnych

Reszta z dzielenia może przydać się także do generowania liczb pseudolosowych. Służąca do tego funkcja `rand()` ma taką niemiłą cechę, że zawsze zwraca liczbę naturalną z zakresu 0…`RAND_MAX`. Wartość tej stałej wynosi u mnie `0x7fff`.
Jak wobec tego wylosować liczbę naturalną z innego zakresu?

Wystarczy w tym celu wykorzystać właściwość operatora `%` mówiącą o „zawijaniu się” jej wykresu po osiągnięciu wartości maksymalnej. Możemy skonstruować taką funkcję:

```
// Losuje liczbę z zakresu 0...max-1
inline UINT my_rand(UINT max)
{
   return rand() % max;
}
```

Warto zauważyć, że tylko dla stosunkowo małych wartości maksymalnych prawdopodobieństwo rozkładu wartości losowanych będzie w miarę równomierne.

Tak naprawdę, działanie `%` nie jest niezbędne. Można je sobie skonstruować za pomocą wyrażenia: `(x - y *(x / y )`.
`/` jest tutaj dzieleniem całkowitym.

Operatory bitowe

Oprócz operacji na pojedynczych wartościach logicznych, działania z algebry Boole'a mają w języku C++ także inne odpowiedniki. Są nimi operatory bitowe – takie, które wykonują podaną operację logiczną na wszystkich odpowiadających sobie bitach podanych wartości.

Najlepiej będzie pokazać to na przykładzie. Pierwszym operatorem bitowym, jaki poznamy, będzie operator negacji bitowej oznaczany symbolem ~ (tzw. tylda).

~(00100111) = 11011000

	0	0	1	0	0	1	1	1
~	1	1	0	1	1	0	0	0

Tabela 30. Przykład negacji bitowej.

Jak widać, każdy z bitów został zanegowany.

Operatorami dwuargumentowymi są operator sumy bitowej (alternatywy) oznaczany za pomocą | (taka pionowa kreska) oraz operator iloczynu bitowego (koniunkcji) oznaczany jako &. Wykonywane przez nie operacje na poszczególnych bitach są analogiczne, jak w algebrze Boole'a.

00100111 | 11001010 = 11101111

	0	0	1	0	0	1	1	1
|	1	1	0	0	1	0	1	0
=	1	1	1	0	1	1	1	1

Tabela 31. Przykład sumy bitowej.

00100111 & 11001010 = 00000010

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

Tabela 32. Przykład iloczynu bitowego.

Istnieją jeszcze operatory przesunięcia bitowego. Ich użycie umożliwia przesunięcie całej wartości o określoną liczbę bitów w lewo lub w prawo. Powstałe po przesunięciu miejsce jest wypełnione zerami. Bity „wypychane" poza komórkę pamięci są bezpowrotnie tracone.

Przykład:

00000101 << 4 = 01010000

Operatory przesunięcia bitowego mają ogromne zastosowanie do konstruowania wartości z kilku elementów, np.:

(1 << 3) | (0 << 2) | (0 << 1) | (1 << 0) = (1 << 3) | 1 = 1000 | 1 = 1001

Nie wolno pomylić operatorów bitowych z odpowiadającymi im operatorami logicznymi. Trzeba uważać na tą różnicę tym bardziej, że są one podobne w zapisie i łatwo tutaj o pomyłkę. Tymczasem wartości takich wyrażeń będą zupełnie różne od oczekiwanych.

„**Lub**": | - bitowe, || - logiczne
„**I**": & - bitowe, && - logiczne

|| i && operują na całych liczbach, a | i & na pojedynczych bitach.

Zadanie 17[114]
Oblicz i podaj wynik w systemie binarnym, dziesiętnym oraz szesnastkowym:

```
1. 0x0f | 99
2. (05 << 4) / 4
3. 010 & 0x10 & 10
4. ((0x0f << 3) | (0xf0 >> 4) - 100) * (257 & 24)
```

Xor

Pozostał nam do omówienia jeszcze jeden operator bitowy – operator różnicy symetrycznej zwany też „xor" (ang. *exclusive or* – wyłącznie lub). Jego wynikiem jest w danym bicie jedynka wtedy i tylko wtedy, kiedy dokładnie jeden z porównywanych bitów jest 1 – nie żaden ani nie obydwa.

Oznacza się go w C++ symbolem ^ (ptaszek). Należy zapamiętać, że ten znak to właśnie *xor*, a nie, jak to się czasami oznacza, podnoszenie liczby do potęgi. W C++ nie ma operatora potęgowania.

Co takiego daje nam ten operator? Nie byłoby w nim niczego użytecznego, gdyby nie niezwykłe właściwości działania, które on wykonuje. Oprócz tego, że (podobnie jak wszystkie pozostałe operatory bitowe) jest ono przemienne, dla każdych x i y zachodzi także własność:

x ^ y ^ y = x

Innymi słowy, po dwukrotnym „przexorowaniu" liczby przez tą samą wartość otrzymujemy daną liczbę wyjściową.

Szyfrowanie

Można pokusić się o napisanie na tej podstawie prostego algorytmu szyfrującego. Zasada jego działania będzie następująca:

	J	a	k	i	e	ś	_	c	o	ś
^	N	i	c	N	i	c	N	i	c	N

[114] 1) 1111 | 1100011 = 1101111 = $(111)_{10}$ = 0x6F; 2) (101 << 4) / 4 = 1010000 / 4 = 80 / 4 = 20 = 0x14 = 10100; 3) 1000 & 10000 & 1010 = 0; 4) ((1111 << 3) | (11110000 >> 4) – 100) * (100000001 & 11000) = (1111000 | 1111 – 100) * 0 = 0

=	X	X	X	X	X	X	X	X	X	X

Tabela 33. Szyfrowanie za pomocą xor.

Każdą komórką tabeli jest tym razem znak, a więc cały bajt. Wykonując xor każdego znaku przez hasło otrzymujemy pewien szyfrogram, czyli znaki oznaczone tutaj przez XXXXXXXXXX.

„Jakieś_coś” ^ „Nic” = XXXXXXXXXX

Hasło zostaje „zawinięte”, czyli powtórzone wiele razy. Takie powtórzenie można zaimplementować (jak mam nadzieję już się domyślasz) za pomocą operatora reszty z dzielenia. Oto funkcja:

```
std::string SzyfrowanieXor(std::string a_Text, std::string a_Haslo)
{
   std::string Wynik;
   for (size_t i = 0; i < a_Text.size(); i++)
         Wynik += a_Text[i] ^ a_Haslo[i % a_Haslo.size()];
   return Wynik;
}
```

Najciekawsze w tym wszystkim jest to, że ta sama funkcja służy do szyfrowania i deszyfrowania tekstu (lub, odpowiednio, zaszyfrowanego tekstu) przez podane hasło. Wynika to oczywiście z własności, którą zaprezentowałem na początku.

	X	X	X	X	X	X	X	X	X	X
^	N	i	c	N	i	c	N	i	c	N
=	J	a	k	i	e	ś	_	c	o	ś

Tabela 34. Deszyfrowanie za pomocą xor.

XXXXXXXXXX ^ „Nic” = „Jakieś_coś”

Suma kontrolna

Drugim ciekawym zastosowaniem różnicy symetrycznej jest obliczanie sum kontrolnych, czyli wartości skojarzonych z pewnymi danymi i pozwalających zweryfikować ich poprawność.

Na przykład jeśli piszesz program do kompresji, możesz w swoim formacie pliku przewidzieć miejsce na sumę kontrolną obliczoną z kompresowanych danych. Podczas dekompresji obliczysz sumę jeszcze raz i jeśli nie zgodzi się z tą zapisaną, to znaczy że dane zostały uszkodzone. Należy wtedy wyświetlić błąd.

Aby obliczyć jednobajtową sumę kontrolną, wystarczy „przexorować” wszystkie bajty danych. Oto przykładowa funkcja:

```
BYTE SumaKontrolna(void* a_Dane, size_t a_Rozmiar)
{
   BYTE Suma = 0;
   for (size_t i = 0; i < a_Rozmiar; i++)
         Suma = Suma ^ static_cast<BYTE*>(a_Dane)[i];
   return Suma;
}
```

Przedstawione metody szyfrowania i obliczania sumy kontrolnej nie są najlepsze czy najbezpieczniejsze. W powszechnym użyciu są dużo lepsze, ale i bardziej skomplikowane. Wszystko zależy od tego, do jakiego zastosowania potrzebujesz danego algorytmu oraz

czy masz ochotę na napisanie go samemu :)
Te przykłady miały tylko pokazać zastosowanie operatora ^.

Flagi bitowe

Mimo małych możliwości, czasami potrzebne jest przechowywanie danych na jednym bicie – czyli wartości logicznej typu prawda/fałsz. Do operowania na pojedynczej wartości tego rodzaju doskonale nadaje się typ logiczny.

Gorzej, kiedy zachodzi potrzeba utworzenia, przesłania czy wykorzystania całego zestawu takich flag bitowych. Operowanie na osobnych zmiennych typu `bool` nie byłoby ani wygodne, ani oszczędne. Dlatego do takich zastosowań wykorzystuje się pojedyncze bity liczb naturalnych.
Nauczmy się operacji na takich flagach bitowych!

Zanim jednak przejdziemy do omawiania tematu, pokażę popularny przykład użycia takich flag. Mają one zastosowanie chociażby w jednym z parametrów znanej wszystkim funkcji `MessageBox()`:

```
MessageBox (0, "Hello world!", "App", MB_OK | MB_ICONEXCLAMATION);
```

Jako ostatniego parametru powyższa funkcja oczekuje zestawu informacji różnego rodzaju, m.in. jakie chcemy mieć przyciski i jaką ikonkę w tworzonym okienku z komunikatem. Chociaż niektóre możliwości wykluczają się (trzeba podać tylko jedną z nich), to wszystko są wartości logiczne w rodzaju tak/nie.

Zastanówmy się, jak takie flagi skonstruować. Z powyższego przykładu widać już, że ich użycie polega na połączeniu wybranych flag operatorem bitowym „lub”. Można się domyślać, że każda z takich flag to stała o nazwie rozpoczynającej się od tego samego przedrostka i o takiej wartości, w której jedynką jest tylko jeden bit – w każdej z nich inny.

Możemy już spróbować zadeklarować sobie jakieś przykładowe flagi:

```
const DWORD BLE_FLAGA1 = 0x00000001uL;
const DWORD BLE_FLAGA2 = 0x00000002uL;
const DWORD BLE_FLAGA3 = 0x00000004uL;
const DWORD BLE_FLAGA4 = 0x00000008uL;
const DWORD BLE_FLAGA5 = 0x00000010uL;
```

Wartości trzeba tak dobrać, aby po przeliczeniu na system binarny odpowiadały jedynkom w kolejnych bitach liczby (czyli kolejne potęgi dwójki).

Przypisanie wartości:

```
DWORD dwWartosc = BLE_FLAGA1 | BLE_FLAGA4;
```

…zainicjalizuje zmienną `dwWartosc` wartością:

```
0001 | 1000 = 1001 = 9
```

Jednak nie wartość liczbowa jest tutaj istotna, ale właśnie kombinacja pojedynczych bitów. Dzięki takiemu podejściu możesz „upychać” wiele informacji typu logicznego (do 32) w pojedynczej 32-bitowej liczbie naturalnej.

Teraz trzeba jakoś odczytywać tak upakowane bity. Posłużymy się w tym celu drugim operatorem bitowym – „i”. Filtrując za jego pomocą kombinację flag bitowych przez

pojedynczą flagę otrzymujemy, zależnie od wartości sprawdzanego bitu w tej wartości – tą flagę lub 0.

```
if ((dwWartosc & BLE_FLAGA4) == BLE_FLAGA4)
   std::cout << "Flaga 4 jest ustawiona." << std::endl;
else
   std::cout << "Flaga 4 nie jest ustawiona." << std::endl;
```

Jak widać, nie jest to trudne. Ma za to ogromne zastosowanie w programowaniu i jest wykorzystywane w różnych API. Dlatego trzeba temat flag bitowych dobrze zrozumieć i nauczyć się ich używać.

Liczby całkowite

Nauczyliśmy się wszystkiego o programowaniu liczb naturalnych. Pora rozszerzyć zakres naszego zainteresowania. Zajmiemy się teraz liczbami całkowitymi, które, jak pamiętamy ze szkoły, mogą być ujemne albo dodatnie, np.: -1000, -21, -5, 0, 3, 10, 25 itp.

Wypadałoby zacząć od opisania sposobu, w jaki komputer przechowuje tego rodzaju liczby w pamięci. Zastanówmy się, jak mógłby on to robić…

Bit znaku

Pierwsze, co przychodzi do głowy, to zarezerwowanie jednego spośród bitów liczby na przechowywanie informacji o znaku.

bit	7	6	5	4	3	2	1	0
waga	znak	64	32	16	8	4	2	1

Tabela 35. Budowa liczby całkowitej z bitem znaku.

Przykładowo można się umówić, że wartość siódmego bitu 0 oznacza znak „+”, a 1 oznacza znak „-”. Moglibyśmy wtedy zapisywać liczby w taki sposób:

```
01010011 = 83
11001101 = -77
```

Niestety, w przedstawiony sposób nie koduje się liczb całkowitych. Rozwiązanie to jest złe, ponieważ:

```
00000000 = 0
10000000 = -0
```

Jak widać, dwie możliwe kombinacje bitów odpowiadają tej samej wartości liczbowej. Jest to ogromna wada z dwóch zasadniczych powodów:

1. Jeden z możliwych stanów marnowałby się.
2. Zachodziłaby konieczność uwzględniania tej niejednoznacznej wartości w obliczeniach.

Dlatego prawdziwy sposób reprezentacji liczb całkowitych w pamięci komputera jest inny. Jego nazwa to:

Kod uzupełnień

Właściwie, jego pełna nazwa to „kod uzupełnień do dwóch”, a jego oznaczeniem jest „U2”. Kod ten opiera się na bardzo ciekawym pomyśle. Przyjrzyj się wagom, jakie przypisuje się poszczególnym bitom liczby całkowitej w tym kodzie:

bit	7	6	5	4	3	2	1	0
waga	-128	64	32	16	8	4	2	1

Tabela 36. Budowa liczby całkowitej w kodzie uzupełnień U2.

Jak widać, waga ostatniego bitu jest ujemna. Aby w pełni rozgryźć istotę sprawy musimy przypomnieć sobie, że liczby w kodzie opisanym takimi wagami liczyło się sumując wagi bitów, którym odpowiadała binarna jedynka. Popatrzmy na przykłady:

$(00000000)_{U2} = 0$
$(01010101)_{U2} = 64 + 16 + 4 + 1 = 85$
$(11110000)_{U2} = -128 + 64 + 32 + 16 = -16$
$(11000011)_{U2} = -128 + 64 + 2 + 1 = -61$

W taki sposób da się zakodować każdą liczbę z dopuszczalnego zakresu. Ano właśnie… Jaki jest zakres liczb w kodzie U2?
Aby odpowiedzieć na to pytanie, spróbujmy znaleźć najmniejszą i największą liczbę, jaką da się zakodować w ten sposób. Będą to odpowiednio:

$(10000000)_{U2} = -128$
$(01111111)_{U2} = 127$

Tak więc dopuszczalny zakres to –128…127.

Odwrotny kod uzupełnień

Można też wymyślić sobie kod uzupełnień o wszystkich wagach ujemnych, a ostatniej dodatniej.

bit	7	6	5	4	3	2	1	0
waga	128	-64	-32	-16	-8	-4	-2	-1

Tabela 37. Inny kod uzupełnień.

Jak można zauważyć, jego zakresem będzie –127…128. Liczby zapisane w takim kodzie uzupełnień są oczywiście niekompatybilne z liczbami zapisanymi w tym pierwszym.

Prawdziwym kodem U2 – tym stosowanym w komputerze – jest ten pierwszy, w którym wszystkie wagi są dodatnie, ostatnia jest ujemna, a zakres jest na minusie o jeden większy niż na plusie.

Zadanie 18[115]
Zaprojektuj kod uzupełnień o podanym zakresie. Ile bitów potrzeba? Narysuj do każdego tabelkę z wagami.

1. –16…15
2. –15…16
3. –1024…1023

[115] 1) 5b, -16,8,4,2,1; 2) 5b, 16,-8,-4,-2,-1; 3) 11b, -1024,512,256,128,64,32,16,8,4,2,1

Kodowanie liczb w U2

Aby zapisać liczbę w kodzie uzupełnień wpisujemy jedynki w bitach o takich wagach, aby ich suma była równa danej liczbie. Przypomnijmy tabelkę z kodem, którego będziemy używali:

bit	7	6	5	4	3	2	1	0
waga	-128	64	32	16	8	4	2	1

Tabela 38. Kod U2, którego używamy.

Przykłady kodowania liczb:

$33 = 32 + 1 = (00100001)_{U2}$
$115 = 64 + 32 + 16 + 2 + 1 = (01110011)_{U2}$

Liczby ujemne zapisuje się analogicznie. Trzeba tylko ustawić ostatni bit na 1 (aby w ogóle otrzymać liczbę ujemną) a następnie wybierać takie bity dodatnie, które zwiększą nam wartość do pożądanej.

$-44 = -128 + 64 + 16 + 4 = (11010100)_{U2}$
$-2 = -128 + 64 + 32 + 16 + 8 + 4 + 2 = (11111110)_{U2}$

Teraz twoja kolej :)

Zadanie 19[116]
Zapisz w powyższym kodzie U2 liczby:
1. 120
2. −120
3. −1
4. 64

Dodawanie i odejmowanie w U2

Tutaj będzie niespodzianka. Okazuje się, że mimo swojej nieco dziwnej budowy liczby w kodzie uzupełnień dodaje się i odejmuje dokładnie tak samo, jak zwykłe liczby binarne! Trzeba przy tym jednak ignorować wszelkie pożyczki oraz przepełnienia poza zakres.

Rozpatrzmy przykłady:

	0	1	0	0	1	0	1	0
+	1	1	1	1	1	0	1	1
=	0	1	0	0	0	1	0	1

Tabela 39. Dodawanie liczb w kodzie U2.

Łatwo zauważyć, że nastąpiło tutaj przepełnienie. Jak się jednak okazuje, mimo tego otrzymany wynik jest poprawny!

$74 + (-5) = 74 - 5 = 69$

Teraz zobaczmy, jak wygląda odejmowanie:

	0	1	0	0	1	0	1	0

[116] 1. $(01111000)_{U2}$; 2. $(10001000)_{U2}$; 3. $(11111111)_{U2}$; 4. $(01000000)_{U2}$

-	1	1	1	1	1	0	1	1
=	0	1	0	0	1	1	1	1

Tabela 40. Odejmowanie w kodzie U2.

Trzeba było dokonać pożyczki od nieistniejącej jedynki. Mimo tego udało się poprawnie obliczyć:

74 – (-5) = 75 + 5 = 79

Jasne jest, że nie zawsze wynik będzie poprawny. Obliczana liczba może się znaleźć poza zakresem i wtedy otrzymamy błędny wynik.

Zadanie 20[117]
Oblicz za pomocą kodu uzupełnień:
1. 2 + 2
2. –6 + (-6)
3. 127 + (-127)
4. 10 – 20

Odwracanie liczby

Oprócz dodawania i odejmowania, z liczbą całkowitą można zrobić jeszcze jedną rzecz – można ją odwrócić. Odwrócenie to inaczej zmiana znaku. Liczbą odwrotną do 6 jest liczba –6, a liczbą odwrotną do –6 jest liczba 6. Odwracanie nazywamy zanegowaniem.

Aby odwrócić liczbę zapisaną w kodzie U2, trzeba wykonać dwie czynności:

1. Zanegować wszystkie bity liczby (negacja bitowa).
2. Do wyniku dodać 1 (normalne dodawanie pod kreskę).

Oto przykład odwracania liczby -78:

	1	0	1	1	0	0	1	0
~	0	1	0	0	1	1	0	1

Tabela 41. Odwracanie liczby - etap 1 z 2 (odwracanie bitów)

	0	1	0	0	1	1	0	1
+	0	0	0	0	0	0	0	1
=	0	1	0	0	1	1	1	0

Tabela 42. Odwracanie liczby - etap 2 z 2 (dodawanie jedynki)

Jak łatwo policzyć, wyszło 78.

Ta możliwość odwracania to dobra wiadomość. Jeśli nie radzisz sobie z kodowaniem liczb ujemnych poprzez gromadzenie potrzebnych wag, możesz normalnie zakodować liczbę dodatnią, a następnie ją odwrócić!

[117] 1. $(00000010)_{U2}+(00000010)_{U2}=(00000100)\ )_{U2}$; 2. $(11111010)_{U2}+(11111010)_{U2}=(11110100)_{U2}$; 3. $(01111111)_{U2}+(10000001)_{U2}=(00000000)_{U2}$; 4. $(00001010)_{U2}-(00010100)_{U2}=(11110110)_{U2}$

Liczby całkowite w programowaniu

Nadeszła pora na poznanie praktycznej implementacji liczb całkowitych w języku C++. Oto przegląd dostępnych typów danych:

- Typ jednobajtowy (8-bitowy) to `signed char`, albo po prostu `char`. Jego zakres to –128…127.
- Typ dwubajtowy (16-bitowy) to `short`. Jego zakres to –32 768…32 767 (±32 tysiące).
- Typ czterobajtowy (32-bitowy) nazywa się `long`. Jego zakresem są liczby –2 147 483 648…2 147 483 647 (±2 miliardy).
- Dodatkowym ośmiobajtowym (64-bitowym) typem jest `__int64` o zakresie -9 223 372 036 854 775 808…9 223 372 036 854 775 807 (±9 trylionów).
- Typem o długości zależnej od używanego kompilatora jest typ `int`. W praktyce w systemach Windows i Linux jest on równoważny typowi 32-bitowemu. To właśnie tego typu używa się najczęściej jako zwyczajnego typu dla liczb całkowitych.

W rzeczywistości możliwych nazw typów jest dużo więcej. Po słowie `short` i `long` można postawić `int`, a przez analogię to typu `__int64` pozostałe mają także nazwy: `__int8`, `__int16` i `__int32`. Analogicznie do typów naturalnych, przed każdym z przedstawionych można też postawić słowo `signed`.

Pozostał do omówienia już tylko sposób zapisywania liczb całkowitych. Jednakże praktycznie nie ma tu czego opisywać. Liczbę można poprzedzić znakiem + lub -. Znak + znaczy tyle samo, co gdyby go tam w ogóle nie było – sygnalizuje, że liczba jest dodatnia.

Liczby rzeczywiste

Wiemy już bardzo dużo o tym, jak komputer radzi sobie z dodatnimi i ujemnymi liczbami całkowitymi. Jednak to nie wystarczy. W obliczeniach zachodzi czasami potrzeba posłużenia się liczbami posiadającymi część ułamkową. Takie liczby nazywamy liczbami rzeczywistymi.

Mówiąc ściślej, liczby rzeczywiste to wszystkie liczby położone na osi liczbowej. Także te, których nie umiemy dokładnie zapisać, np. ludolfina (znana jako π), liczba Nepera (znana jako e) czy pierwiastek z dwóch (takie liczby nazywamy niewymiernymi).

Jak można przypuszczać, komputer nie posługuje się zawsze precyzyjnymi liczbami wykonując operacje na nich tak jak uczeń na matematyce, np.:

$$\frac{10\pi+13}{3}=\frac{10}{3}\pi+\frac{13}{3}$$

Niektóre zaawansowane programy matematyczne to potrafią. Jednak normalnie komputer posługuje się konkretnymi wartościami liczbowymi i na nich wykonuje obliczenia tak, jak my robimy to na lekcjach fizyki. Z konieczności liczby te mają dokładność ograniczoną do pewnej ilości miejsc po przecinku. Liczby niewymierne (jak wspomniana π czy e) są reprezentowane przez wartości przybliżone, np.:

$$\frac{10\pi+13}{3}=\frac{10\cdot 3.14+13}{3}=\frac{13.4+13}{3}=\frac{26.4}{3}=8.8$$

Kod stałoprzecinkowy

Zastanówmy się, w jaki sposób komputer mógłby przechowywać w pamięci liczby rzeczywiste. Aby znaleźć odpowiedź na to pytanie przyjrzyjmy się bliżej metodzie, jakiej my używamy na co dzień.

Zapisując liczbę rzeczywistą, piszemy np. tak:

1984.0415

Kilka cyfr tworzy normalną część całkowitą liczby, dalej występuje przecinek (albo – jak to jest w programowaniu – kropka) i wreszcie kolejne cyfry, które tworzą część ułamkową. Cała liczba zapisana jest w naszym normalnym systemie dziesiętnym.

Gdyby tak zapisywać liczby w taki sam sposób, ale w systemie binarnym, można byłoby bezpośrednio przechowywać je w pamięci komputera! Po prostu – pewna ilość bitów uznawana byłaby za cyfry dwójkowe przed przecinkiem, a pozostałe bity za cyfry po przecinku.

Taki kod – z umieszczonym na stałe przecinkiem – to **kod stałoprzecinkowy** albo **stałopozycyjny**.

Dla przykładu rozważmy liczbę zapisaną w kodzie 16-bitowym. Darujemy sobie dokładne rozpisanie sposobu jej rozkodowania, bo nie jest to tutaj najważniejsze.

(11111111.11111111) = 255 + 255/256 = 255.99609375

Musisz wiedzieć, że w taki sposób **nie** koduje się w komputerze liczb rzeczywistych. Kod stałoprzecinkowy ma wiele wad:

- mały zakres liczb możliwych do zakodowania
- mała dokładność (precyzja) części ułamkowej
- trudny do oszacowania błąd obliczeń

Mała powtórka z fizyki

Skoro taki sposób jest nienajlepszy – to jak wygląda ten lepszy? Niestety sprawa, do której zmierzamy, jest dość skomplikowana. Do jej zrozumienia potrzebna będzie cała wiedza opisana powyżej (w tym kodowanie liczb naturalnych i kodowanie liczb całkowitych w kodzie uzupełnień). Zanim przejdziemy do sedna sprawy, musimy też po raz kolejny zrobić małą powtórkę ze szkolnej wiedzy.

Przypomnijmy sobie, w jaki sposób zapisuje się liczby na przedmiocie najbliższym naszym aktualnym rozważaniom – czyli na fizyce. Liczby mogą być dokładne, np. 100 m. Mogą być też przybliżone do kilku miejsc po przecinku, np. 0.3333 A.

Często zdarza się, że liczba jest bardzo duża albo bardzo mała. Stosuje się w takim wypadku odpowiednie przedrostki wielokrotności, które powtórzyliśmy już sobie niedawno. Przykłady: 22 μF, 3 km.

Do obliczeń trzeba jednak sprowadzić wielkości do jednostek podstawowych. Można wtedy napisać tak: 0.000022 F, 3000 m, ale do wyrażania liczb bardzo dużych i bardzo małych używa się na lekcjach fizyki takiego zapisu:

$22*10^{-6}$ F, $3*10^{3}$ m

Przeanalizujmy to dokładnie. Liczba zapisana w taki sposób składa się z dwóch części. Pierwszą jest „liczba właściwa", a drugą liczba 10 podniesione do jakiejś potęgi. Nietrudno zauważyć, że potęga –6 odpowiada przedrostkowi mikro (μ), a potęga 3 odpowiada przedrostkowi kilo (k). Dziesiątka to, jak można się domyślić, po prostu podstawa naszego systemu.

Tak naprawdę, liczba przed znakiem mnożenia to kodowana liczba po tzw. normalizacji. Np.:

$123000000 = 1.23*10^8$

Umawiamy się, że znormalizowana liczba musi mieć jedną cyfrę przed przecinkiem i pozostałe po przecinku. W tym celu **przesuwamy przecinek** o odpowiednią liczbę miejsc w prawo albo w lewo. Ta liczba miejsc, o jakie przesunęliśmy przecinek, to jest właśnie wykładnik potęgi.
W przypadku przesuwania przecinka w lewo potęga jest (jak widać) dodatnia, a w wypadku przesuwania przecinka w prawo byłaby ujemna.

Na zakończenie tej powtórki zobaczmy jeszcze, jak można rozkodować tak zapisaną liczbę. Jako przykład weźmy: $4.79*10^{-5}$.

Można ją zwyczajnie wyliczyć:

$4.79*10^{-5} = 4.79*1/10^5 = 4.79*0.00001 = 0.0000479$

ale można też po prostu przesunąć przecinek o 5 pozycji w lewo:

000004.79 > 00000.479 > 0000.0479 > 000.00479 > 00.000479 > 0.0000479

Kod zmiennoprzecinkowy FP2

Zostawmy już ten niekoderski system dziesiątkowy i wróćmy do naszego ulubionego zapisu binarnego :) Można wyobrazić sobie przeniesienie wszystkiego, co zostało napisane wyżej, bezpośrednio z systemu dziesiętnego na binarny. Jedyną różnicą będzie liczba 2 zamiast 10 jako podstawa potęgi.

No to teraz, niestety, zobaczymy kolejny przerażający wzór :O

$$L = (-1)^s \cdot m \cdot N^c$$

N to podstawa systemu liczbowego.
s to bit znaku.
- 0 oznacza znak +, bo $(-1)^0 = 1$
- 1 oznacza znak -, bo $(-1)^1 = -1$

c (**cecha**) to wykładnik potęgi, czyli informacja, o ile miejsc przesuwamy przecinek. Może być dodatnia albo ujemna. Jest zapisana w kodzie uzupełnień U2.
m (**mantysa**) to znormalizowana liczba, zapisana jako binarna liczba naturalna.

Taki kod z cechą i mantysą, to **kod zmiennoprzecinkowy** albo **zmiennopozycyjny** (jego oznaczeniem jest **FP2** – od ang. *floating point*). Jego nazwa wzięła się stąd, że cecha może przesuwać przecinek mantysy. Pozwala to na zapisywanie bardzo dużych i bardzo małych liczb.
To właśnie w taki sposób komputer koduje liczby rzeczywiste!

Oprócz dwóch nowopoznanych słówek – „mantysa” i „cecha” – pokazany wzór razem z objaśnieniem powinien nam pomóc w zakodowaniu i rozkodowaniu przykładowej liczby. Do liczby zapisanej w kodzie FP2 zawsze trzeba podać, ile bitów zajętych jest przez mantysę, a ile przez cechę. Oto budowa przykładowego kodu FP2:

	znak	***cecha***					***mantysa***									
bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
waga	s	-16	8	4	2	1	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}

Tabela 43. Budowa przykładowego kodu FP2.

Sprawa wydaje się bardzo trudna i taka jest w istocie. Ale nie trzeba się bać na zapas. Spróbujemy teraz powoli zakodować, a potem rozkodować liczbę w przedstawionym systemie, a wszystko stanie się jasne :)

Kodowanie

Chcemy zapisać liczbę 1984.0415 w kodzie FP2 o budowie przedstawionej w powyższej tabelce. W tym celu wykonujemy kolejne kroki:

Po pierwsze, ustalamy znak. Liczba jest dodatnia, a więc:

$s = 0$

Gdyby była ujemna, wartością bitu byłoby 1, a dalej rozpatrywalibyśmy już liczbę przeciwną – czyli pozbawioną tego minusa (dodatnią).

Teraz musimy zapisać tą liczbę w normalnym systemie binarnym. Posłużymy się poznanym wcześniej algorytmem Hornera. Jako docelową liczbę bitów przyjmujemy 11 – o jeden więcej, niż może przechować mantysa (zaraz się okaże, dlaczego właśnie tak).

$1984.0415 * 2^{11} = 4\,063\,316.992 \approx 4\,063\,317$

4 063 317 : 2	1
2 031 658 : 2	0
1 015 829 : 2	1
507 914 : 2	0
253 957 : 2	1
126 978 : 2	0
63 489 : 2	1
31 744 : 2	0
15 872 : 2	0
7 936 : 2	0
3 968 : 2	0
1 984 : 2	0
992 : 2	0
496 : 2	0
248 : 2	0
124 : 2	0
62 : 2	0
31 : 2	1
15 : 2	1
7 : 2	1
3 : 2	1
1 : 2	1
0	

Tabela 44. Kodowanie liczby algorytmem Hornera.

A więc mamy:

1984.0415 = 11111000000.00001010101

OK, liczba została przeliczona. Teraz możemy zająć się właściwym kodowaniem w FP2. Dokonujemy normalizacji – czyli przesuwamy przecinek tak, aby przed przecinkiem znajdowała się tylko jedna niezerowa cyfra.

Wychodzi coś takiego:

1.11110000000001010101

Zauważ, że jedyną możliwą niezerową cyfrą w systemie dwójkowym jest 1. Skoro przed przecinkiem zawsze stoi pojedyncza jedynka, możemy zapamiętać, że ona tam jest i oszczędzić jednego bitu nie zapisując jej. Dlatego ją pogrubiłem.

Można też spotkać taki kod, w którym wszystkie znaki znajdują się za przecinkiem – tzn. przecinek przesuwany jest tak, aby bezpośrednio za nim znalazła się pierwsza jedynka. Kolejne wagi mantysy miałyby wtedy postać 2^{-2}, 2^{-3}, 2^{-4} itd.
O ile się nie mylę, komputer w rzeczywistości używa jednak tego kodu z jedynką przed przecinkiem.

Ostatecznie mantysę utworzą kolejne cyfry spisane od przecinka dotąd, dokąd zmieszczą się w przyjętej długości mantysy (u nas 10 bitów). Gdybyśmy mieli mniej cyfr, niż jest potrzebne, mantysę uzupełnia się zerami z prawej strony.

m = 1111000000

Teraz zajmiemy się cechą. Pamiętamy, że przecinek przesunęliśmy o 10 miejsc w lewo. Jako cechę trzeba więc będzie zapisać liczbę 10 w kodzie U2, którego budowę można wyczytać z tabelki z budową kodu zmiennoprzecinkowego, w jakim kodujemy.

Trzeba odróżnić ujemną wartość cechy od ujemnej wartości mantysy. Tutaj obydwie te liczby są dodatnie, ale równie dobrze mogłyby być ujemne. Ujemna mantysa oznacza, że kodowana liczba jest ujemna. Ujemna cecha oznacza, że podczas normalizowania przesuwamy przecinek w prawo.

Ostatecznie cecha wygląda tak:

$c = (01010)_{U2}$

Jak widać, mantysę zapisuje się w postaci liczby naturalnej i osobnego bitu znaku, a cechę za pomocą kodu uzupełnień U2. Co więcej, na schemacie budowy kodu FP2 widać, że znak mantysy jest od samej mantysy oddzielony cechą.

Mamy już wszystkie części liczby. Możemy ją zapisać w pełnej okazałości:

$1984.0415 = (0\ 01010\ 1111000000)_{FP2}$

Wiem, wiem, to nie było proste :) Warto wrócić w tej chwili to teorii i do tego przerażającego wzoru (może wyda się teraz już odrobinę mniej straszny?), a potem jeszcze raz dokładnie przeanalizować powyższy sposób kodowania liczby.

Dekodowanie

Rozkodujemy teraz tą liczbę z powrotem. Posłuży do tego przedstawiony niedawno wzór. Rozpatrując kolejne elementy, otrzymujemy:

s = 0, a więc mamy: $(-1)^0$

m = 1111000000, a więc mamy: $2^{-1} + 2^{-2} + 2^{-3} + 2^{-4}$

c = $(01010)_{U2}$ = 10, a więc mamy 2^{10}

Składając to razem zgodnie z wzorem, otrzymujemy do policzenia takie coś:

$(-1)^0 * (\mathbf{2^0} + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4}) * 2^{10} =$

Pogrubiony składnik odpowiada jedynce przed przecinkiem, którą zapamiętaliśmy i nie zapisaliśmy w zakodowanej liczbie, oszczędzając jeden bit. Nie wolno o niej zapomnieć podczas rozkodowania liczby.
Uważaj na opuszczenie tego składnika – to bardzo częsty błąd!!!

Kiedy to policzymy, wyjdzie nam od razu rozkodowana liczba. A więc do dzieła!

$(-1)^0$ = 1, a więc ten element możemy pominąć.

Można byłoby teraz żmudnie podnosić te dwójki do tych ujemnych potęg, ale jest lepsze wyjście. Możemy do każdego z wykładników potęg w nawiasie dodać wykładnik tej potęgi za nawiasem. Ostatecznie można liczyć dalej tak:

$2^{10} + 2^9 + 2^8 + 2^7 + 2^6 = 1024 + 512 + 256 + 128 + 64 = 1984$

Wyszło mniej więcej tyle, ile powinno było wyjść. Ale chyba czegoś tu brakuje :) Czyżbyśmy o czymś zapomnieli? Może nie zrobiliśmy czegoś ważnego i dlatego obcięło nam część ułamkową liczby?

Otóż nie! Wszystko jest OK. Po prostu zakodowanie liczby w takim a nie innym systemie FP2, za pomocą określonej ilości bitów przeznaczonych na cechę i na mantysę spowodowało ograniczenie dokładności do iluś miejsc po przecinku (tym już przesuniętym). Stąd utrata dalszej części liczby.

Po głębszym zastanowieniu można wywnioskować, że precyzja liczby zależy od ilości znaków przeznaczonych na mantysę, a zakres liczby od ilości znaków przeznaczonych na cechę.

Wartości specjalne

Standard kodu FP2 przewiduje dodatkowo wartości specjalne:

1. Maksymalna wartość cechy przy zerowej wartości mantysy daje w zależności od bitu znaku mantysy wartość zwaną –INF lub +INF (oznaczającą odpowiednio -∞ i +∞).
2. Maksymalna wartość cechy przy jakiejkolwiek niezerowej wartości mantysy to tzw. NaN (ang. *Not a Number*), czyli wartość, która nie jest poprawną liczbą.

Ćwiczenia

Całe to kodowanie liczb rzeczywistych wygląda na bardzo trudne i niewdzięczne. Zapewniam cię jednak, że przy odpowiednim podejściu może być naprawdę doskonałą

zabawą!
Osobiście lubię czasem zakodować sobie jakąś liczbę w jakimś wymyślonym kodzie FP2, potem ją rozkodować i zobaczyć, co z niej zostało :D

Zadanie 21[118]

1. Rozkoduj liczbę $(1\ 11111\ 1111111111)_{FP2}$.
2. Zakoduj liczbę 9999 w kodzie FP2.
3. Zakoduj liczbę w −0.2 w kodzie FP2 w 3 bitach cechy i 4 bitach mantysy.
4. Zakoduj w kodzie FP2 liczbę 0.333333…, potem ją rozkoduj i na podstawie wyniku oszacuj dokładność używanego w tym dokumencie 16-bitowego formatu.

Liczby rzeczywiste w programowaniu

Tradycyjnie już, poznawszy szczegóły przechowywania danych na poziomie pojedynczych bitów, przechodzimy do omawiania rzeczy praktycznych.

Typy zmiennoprzecinkowe

Najpierw poznamy typy danych, które pozwalają programować operacje na liczbach zmiennoprzecinkowych.

Pierwszym z typów zmiennoprzecinkowych w C++ jest typ `float`. Zajmuje 32 bity (4 bajty), z których na cechę przeznaczonych jest 8, a na mantysę 23. Jego zakres wynosi $-1.175494351*10^{-38}$…$3.402823466*10^{38}$, a jego dokładność to 6…7 znaczących cyfr. Nazywany bywa typem zmiennoprzecinkowym o pojedynczej precyzji. To właśnie on jest używany najczęściej – jego precyzja wystarcza do zdecydowanej większości zastosowań.

Kolejnym typem jest `double`. Zmienne tego typu zajmują 64 bity (8 bajtów). Cecha zajmuje 11 bitów, a mantysa 52. Jego zakres wynosi $2.2250738585072014*10^{-308}$…$1.7976931348623158*10^{308}$, jego dokładność to 15…16 cyfr. Nazywany bywa (jak sama jego angielska nazwa wskazuje :) typem o podwójnej precyzji.

Wprowadzone przeze mnie w tym tekście pojęcia liczb naturalnych, całkowitych czy rzeczywistych są zaczerpnięte z matematyki. W praktyce do nazywania typów danych programista posługuje się nazwami: **liczby całkowite bez znaku**, **liczby całkowite ze znakiem** (x-bitowe) oraz **liczby zmiennoprzecinkowe** (pojedynczej lub podwójnej precyzji).

Zapisywanie wartości zmiennoprzecinkowych

Domyślnie całkowita wartość liczbowa traktowana jest jako wartość jednego z typów całkowitych. W praktyce często kompilator „domyśla się”, że chodzi o typ zmiennoprzecinkowy i nie zgłasza ostrzeżenia. Zawsze warto jednak podkreślić typ.

Aby pokazać wszelkie możliwe sposoby zapisu liczb zmiennoprzecinkowych, najlepiej będzie rozpatrzyć przykładowy kod:

```
float a = 2;        // 1
float b = 2.;       // 2
float c = 2.0;      // 3
float x = 2.0f;     // 4
float y = -10.5;    // 5
```

[118] 1. $-(1+2+4+8+16+32+64+128+256+512+\mathbf{1024})/2048 = -2036/2048 = -0.9941…$; 2. $(0\ 01101\ 0011100001)_{FP2}$; 3. $(1\ 101\ 1001)_{FP2}$; 4. $(0\ 11110\ 0101010101)_{FP2} \Rightarrow (1+4+16+64+256+\mathbf{1024})/4096 = 1365/4069 = 0.33325…$; dokładność wynosi w tym przypadku ok. 3 cyfr dziesiętnych po przecinku.

```
float z = -5.2e-3; // 6
```

1. Tutaj podana zostaje wartość całkowita, która przy mniejszym lub większym proteście kompilatora zostanie automatycznie potraktowana jako zmiennoprzecinkowa.
2. Można zapisać kropkę nie wpisując po niej kolejnych cyfr. Przyznasz chyba jednak, że nienajlepiej to wygląda? :)
3. Liczbę można wpisać normalnie jako część przed przecinkiem, kropkę (która, jak pamiętamy, w programowaniu robi za przecinek :) oraz cyfry po przecinku.
4. Wartość zawierająca przecinek domyślnie traktowana jest jako stała dosłowna typu `double`. Aby wymusić potraktowanie jej jako wartość typu `float`, należy postawić na jej końcu literkę `f`.
5. Tutaj nie ma niczego nowego. Chciałem tylko pokazać przez to, że liczba może być ujemna, a po przecinku mogą znajdować się cyfry inne niż zero (co, mam nadzieję, wydaje się oczywiste :)
6. To jest alternatywny sposób zapisu liczb zmiennoprzecinkowych – wygodny, jeśli chodzi o liczby bardzo duże oraz bardzo małe. Nazywa się to **notacja naukowa** (ang. *scientific notation*) i polega na tym, że piszemy mantysę, dalej literkę `e` oraz cechę.

Pozostańmy jeszcze przez chwilę przy ostatnim przykładzie. Litera `e` nie ma niczego wspólnego ze stałą równa 2.72… To tylko takie oznaczenie (które pochodzi od ang. *exponent*, czyli wykładnik), a cały ten zapis znaczy tyle co:

$-5.2 * 10^{-3} = -0.0052$

Wartość zapisana w notacji naukowej domyślnie jest typu `double` i podczas pokazanego wyżej przypisania kompilator zgłosi ostrzeżenie. Nic nie stoi jednak na przeszkodzie, żeby na końcu tak zapisanej wartości także postawić magiczną literkę `f` :)

Ogólnie warto, abyś wypracował sobie (z uwzględnieniem natury używanego przez ciebie kompilatora i jego ostrzeżeń) standard dotyczący zaznaczania lub nie zaznaczania typu przy stałych dosłownych za pomocą przyrostków `u`, `L` czy `f`. Ja osobiście stosuję tylko `f`.

Operatory zmiennoprzecinkowe

Liczby rzeczywiste, podobnie jak całkowite, można dodawać, odejmować, mnożyć i dzielić. Nie wymaga to chyba dłuższego komentarza… A może jednak? :) Okazuje się, że jest o czym pisać.

Po pierwsze: do liczb rzeczywistych nie da się stosować operatora reszty z dzielenia `%`.

P drugie, operator dzielenia / wykonuje dzielenie rzeczywiste wtedy, kiedy przynajmniej jeden z jego argumentów jest rzeczywisty. Jego rezultatem jest wtedy także liczba rzeczywista. W przeciwnym wypadku dokonuje dzielenia całkowitego z obcięciem reszty i jego wynik jest także całkowity.

```
float a = 10 / 3;
float b = 10.0f / 3;
float c = 10.0f / 3.0f;
```

W przedstawionym przykładzie zmienna `b` i `c` będzie przechowywała wartość 3.3333…, natomiast wartością zmiennej `a` będzie liczba 3. W jej przypadku nastąpiło dzielenie całkowite oraz konwersja zwróconej wartości całkowitej na liczbę zmiennoprzecinkową.

Jak widać, należy zawsze bardzo uważać na typ argumentów biorących udział w dzieleniu. Przy okazji uczulam też na możliwość wystąpienia błędu dzielenia przez 0. Przed nią także trzeba się zabezpieczać.

Funkcje matematyczne

Skoro już jesteśmy przy liczbach rzeczywistych, pozwolę sobie opisać kilka funkcji matematycznych. Być może nie wszystkie są ci znane i nie wszystkich będziesz często używał. Zapewniam jednak, że większość z nich wypada znać i jest naprawdę użyteczna w programowaniu.

Do **potęgowania** nie ma w języku C++ (podobnie jak w większości języków programowania) operatora takiego, jak do dodawania czy mnożenia. Podnoszenie do kwadratu można sobie łatwo zrobić pisząc `(x*x)`. Analogicznie sześcian (trzecia potęga) to będzie: `(x*x*x)`. Do podnoszenia liczby do dowolnej potęgi rzeczywistej służy funkcja `pow(x, y)`.

Pierwiastek kwadratowy oblicza funkcja `sqrt(x)`. Dowolny inny można sobie zrobić pisząc `pow(x, 1.0/y)`.

Wartość bezwzględną (moduł) z liczby x oblicza funkcja `fabs(x)`. Istnieje też jej odpowiednik do liczb całkowitych: `abs(n)`.

Nie wszystkie **funkcje trygonometryczne** reprezentowane są w C++ przez funkcje o takich samych nazwach, jak nazwy tych funkcji w matematyce. Sinus oblicza funkcja `sin(x)`, a cosinus: `cos(x)`. Tangens, który oznacza się przez tg, obliczany jest przez funkcję `tan(x)`. Cotangensa w ogóle nie ma, ale znając podstawowe wzory trygonometryczne można go łatwo wyliczyć w taki sposób: `(1.0/tan(x))`. Trzeba tylko uważać, żeby nie wykonać dzielenia przez 0.

Funkcje cyklometryczne wyglądają w C++ podobnie do trygonometrycznych. Są to odpowiednio funkcje: `asin(x)`, `acos(x)` i `atan(x)`. Arcus cotangensa znowuż nie ma, ale można go wyliczyć za pomocą wyrażenia: `(M_PI/2.0-atan(x))`.

Jeszcze jedną grupą podobnych funkcji są **funkcje hiperboliczne**. Odpowiadają im w C++ odpowiednio: `sinh(x)`, `cosh(x)` i `tanh(x)`. Cotangensa hiperbolicznego nie ma, a wyliczyć go można ze wzoru: `((exp(x)+exp(-x))/(exp(x)-exp(-x)))`.

Do obliczania **logarytmów** służy kilka funkcji różniących się podstawą. `log10(x)` oblicza logarytm dziesiętny (o podstawie 10), a `log(x)` oblicza logarytm naturalny (o podstawie e).

Warto jeszcze wspomnieć o funkcji `exp(x)` podnoszącej stałą *e* do podanej potęgi (jest to funkcja tzw. eksponencjalna). Oczywiście różnych funkcji matematycznych jest dużo więcej. Po szczegóły odsyłam do opisu bibliotek *cmath* oraz *math.h* w dokumentacji.

Wszystkie opisane tu funkcje pochodzą z biblioteki *math.h*. Musisz ją włączyć do modułu, w którym używasz tych funkcji.

Przedstawione funkcje operują na liczbach typu `double`. Każda ma jednak swój odpowiednik dla typu `float`, którego nazwa kończy się na „f”, np.: `sinf(x)`, `log10f(x)`.

Stałe liczbowe

Od czasu do czasu zachodzi potrzeba użycia w pisanym programie jednej z „magicznych” liczb, jak liczba π czy *e*. Można je sobie zdefiniować samemu, ale okazuje się, że są one wpisane także do standardowych nagłówków.

Biblioteka *math.h* zawiera wiele użytecznych stałych, w tym stałe o nazwach `M_PI` i `M_E`.

Jeśli używasz DirectX i włączasz do kodu nagłówek rozszerzenia D3DX, masz też do dyspozycji stałą typu `float` o nazwie `D3DX_PI`.

Losowanie liczb

Podobnie jak przy okazji omawiania liczb naturalnych, także teraz zastanowimy się nad otrzymywaniem liczb pseudolosowych za pomocą niezbyt wygodnej funkcji `rand()`. Moje rozwiązanie wygląda tak:

```
inline float myrandf()
{
   return static_cast<float>(rand()) / static_cast<float>(RAND_MAX);
}
```

Przedstawiona funkcja zwraca losową liczbę z zakresu 0.0…1.0 włącznie dzieląc otrzymaną liczbę całkowitą przez jej maksymalny zakres.

Konwersja z liczbami całkowitymi

Czasami trzeba zamienić liczbę rzeczywistą na całkowitą lub odwrotnie. Oczywiście podczas konwersji z liczby zmiennoprzecinkowej na całkowitą reszta zostaje obcięta.

Kompilator potrafi przeprowadzić taką konwersję automatycznie.

```
float f = 10.5f;
int i = f;
f = i;
```

Wyświetla jednak przy tym ostrzeżenia, że może zostać utracona dokładność. Żeby oszczędzić sobie czytania, lepiej stosować operatory rzutowania, np.:

```
float f = 10.5f;
int i = static_cast<int>(f);
f = static_cast<float>(i);
```

Znaki

Rozpoczynamy ostatni podrozdział. Zajmiemy się w nim sposobem, w jaki komputer przechowuje w pamięci tekst.

Chociaż komputery zostały stworzone do liczenia, od zawsze posługiwały się tekstem. Nie chodzi tylko o możliwość pisania na komputerze wypracowań i referatów :) Jeszcze zanim pojawiły się pierwsze systemy okienkowe, ludzie wydawali komputerom polecenia za pomocą konsoli – czyli tekstowego wiersza poleceń.

Na rozgrzewkę proponuje wykonanie następującego ciekawego zadanka:

Zadanie 22[119]

1. Przyjrzyj się uważnie klawiaturze i popatrz na znaki, które możesz za jej pomocą wprowadzać. Jakich symboli brakuje, a przydałyby się? Które znaki są niepotrzebne i rzadko ich używasz?
2. Zdobądź i obejrzyj tablicę znaków ASCII. Które z tych znaków rozpoznajesz jako możliwe do wpisania z klawiatury? Które ci się podobają? Które uważasz za bezużyteczne?
3. Zaprojektuj własną tablicę znaków składającą się z 64 pozycji (tablica 8x8). Które znaki wybierzesz uważając je za najistotniejsze i dlaczego? Ilu bitów potrzeba do zapisania jednego takiego znaku?

ASCII

Aby zakodować tekst, trzeba każdej możliwej kombinacji bitów przyporządkować pewien znak. Najpopularniejszym standardem kodowania znaków jest kod ASCII. Jeden znak w tym kodzie zajmuje 1 bajt (8 bitów). Standard ten obowiązuje już od czasów systemu DOS aż po dziś dzień.

Tak naprawdę, sam standard ASCII wykorzystuje 7 bitów. Oznacza to, że dostępnych jest 128 różnych kombinacji bitów, czyli można zapisać 128 różnych znaków. Czy to dużo, czy mało? To zależy do czego… Taka ilość spokojnie wystarczyła, by pierwszym 32 znakom (odpowiadającym zakodowanym w systemie binarnym liczbom naturalnym 0…31) przypisać pewne specjalne kody sterujące, a dalej zmieścić cyfry, małe i duże litery alfabetu łacińskiego oraz wszystkie znaki znajdujące się na klawiaturze.

Na przykład:

- 32 = 0x20 = 040 oznacza spację
- 43 = 0x2B = 053 oznacza znak „+”
- 57 = 0x39 = 071 oznacza cyfrę „9”
- 97 = 0x61 = 0141 oznacza literę „a”

Pośród znaków ASCII wydzieloną grupę stanowią tzw. białe znaki (ang. *whitespace)* albo inaczej odstępy. Są one uznawane za znaki oddzielające pewne części tekstu. Należą do nich 4 znaki: spacja (kod 32), tabulacja (kod 9), koniec wiersza (kod 10) oraz powrót karetki (kod 13).

W wielu językach (w tym w językach programowania, np. C++ czy Pascal oraz w językach opisu, np. HTML czy XML) fakt, jakie spośród tych znaków wystąpią, w jakiej ilości i w jakiej kolejności, nie ma znaczenia – każda taka sekwencja traktowana jest jako pojedynczy odstęp. To dzięki temu możemy robić wcięcia w kodzie i swobodnie go rozmieszczać (zauważ, że wcięcie to znak końca wiersza plus pewna liczba spacji lub tabulacji).

Znaki końca wiersza

Poświęcenia dodatkowej uwagi wymaga temat znaków uznawanych za koniec wiersza (linii) w tekście. Panują w tej kwestii dwa różne standardy. W Windows koniec wiersza

[119] 1) Jakiej dziedziny dotyczą znaki, które chętnie byś dodał? Czy są to symbole matematyczne? A może jakieś inne? Jak można sobie bez nich poradzić? 2) Tablicę znaków ASCII łatwo znajdziesz w Internecie, choćby pod adresem http://www.asciitable.com/. Często była też publikowana na końcu starszych książek o programowaniu, np. o Turbo Pascalu. 3) Zapewne znalazłoby się w niej miejsce dla dużych liter alfabetu, polskich liter, cyfr, spacji (odstępu), podstawowych znaków przestankowych i kilku innych symboli. Potrzeba 6b.

zaznacza się sekwencją dwóch znaków CR (kod 13) i LF (kod 10). W Linux natomiast samym znakiem LF (kod 10).

Z kompatybilnością między tymi formatami bywa różnie. W Linux podwójny koniec wiersza najczęściej zinterpretowany zostanie prawidłowo, o ile wiersz może kończyć się odstępem i ten odstęp zostanie zignorowany.

Notatnik Windows nie odczyta poprawnie dokumentu zapisanego ze znakami końca wiersza w stylu linuxowym. Na prawidłowe jego wyświetlenie możesz za to liczyć w programie Lister wbudowanym w Total Commander.

Aby edytować i zapisywać pod Windows dokumenty ze znakami końca wiersza w stylu linuxowym, możesz użyć jednego z tekstowych edytorów HTML, np. HomeSite lub Pajączek. Trzeba tylko uaktywnić specjalną opcję w konfiguracji. Nazywa się ona najczęściej zapisywaniem znaków końca wiersza w stylu Unix.

Extended ASCII

256 możliwych kombinacji bitów w jednym bajcie to jednak za mało, by zapisać znaki specyficzne dla różnych języków świata, jak polskie „ąćęłńóśźż" czy niemieckie „umlauty" (nie mówiąc już o zupełnie innych alfabetach, jak cyrylica czy znaki chińskie).

Dlatego dodatkowe 128 znaków powstałe po użyciu ósmego bitu nie jest ujednolicone. Stworzonych zostało wiele tzw. stron kodowych (ang. *codepage*) używających tych dodatkowych znaków do kodowania liter alfabetów narodowych, a przy tym różnych symboli graficznych i innych bardziej lub mniej przydatnych.

Jest z tym niestety dużo problemów. Nawet dla samego języka polskiego powstało kilka kodów. Obecnie używane są dwa:

1. ISO-8859-2 (Latin-2)
2. Windows-1250

Ten drugi jest przez wielu potępiany za to, że został wylansowany przez Microsoft. W praktyce jednak to właśnie jego używa system Windows, a wielu miejscach Internetu jest on nie mniej popularny, niż ten pierwszy.

W wielu zastosowaniach, szczególnie w Internecie (WWW, e-mail) w nagłówku zapisywana jest nazwa standardu kodowania użytego w danym dokumencie. Pozwala to zminimalizować problemy wynikające z całego tego bałaganu.

Aby sprawdzić, czy prawidłowo działają w jakimś programie, systemie czy gdziekolwiek indziej polskie litery, wpisuje się zazwyczaj utarty tekst: „**Zażółć gęślą jaźń**". Choć nie ma on większego sensu, ma to do siebie, że będąc poprawnym gramatycznie zdaniem zawiera w sobie wszystkie polskie literki.

Unikod

Rozwój Internetu stworzył konieczność wynalezienia lepszego sposobu kodowania znaków, niż wysłużony już kod ASCII. Nawet ze swoimi stronami kodowymi ten ostatni ma wiele ograniczeń i sprawia wiele problemów. Nie można chociażby zapisać tekstu w kilku różnych językach w jednym dokumencie.

Pomyślano więc tak: Właściwie, skoro dzisiejsze dyski mają pojemności mierzone w gigabajtach, a obrazki i filmy zajmują o całe rzędy wielkości więcej miejsca niż tekst, po

co nadal ograniczać się do jednego bajta na znak? Dlaczego nie utworzyć kodu, w którym jeden znak zajmowałby, powiedzmy, 2 bajty?

Tak powstał Unikod (ang. *Unicode*, w skrócie UCS). Warto zdać sobie sprawę z faktu, że już za pomocą 2 bajtów można zakodować 2^{16} = 65536 różnych znaków! Dlatego w unikodzie znalazło się miejsce dla wszelkich użytecznych i używanych na świecie liter, symboli i znaków, a po upowszechnieniu się tego standardu nasze dzieci będą już tylko od nas słyszały historie, jakie to kiedyś były problemy w komputerze z kodowaniem znaków :)

Najpopularniejszymi odmianami unikodu są UTF-8 i UTF-16. W tej pierwszej znak może mieć różną długość. Pierwsze 128 znaków pokrywa się z tablicą ASCII i jest zapisywana za pomocą jednego bajta, natomiast znaki dodatkowe (np. polskie literki) są zapisywane za pomocą specjalnych kilkubajtowych sekwencji. Z kolei UTF-16 określa standard, w którym każdy znak zajmuje 2 bajty.

Nie będziemy się tutaj dokładnie zajmowali unikodem. Może w następnym wydaniu tego tekstu… Tymczasem musisz wiedzieć, że już dziś wiele programów i systemów operacyjnych używa go jako standardowego sposobu kodowania znaków, a programowania z użyciem unikodu warto się nauczyć.

Po szczegóły odsyłam do samego źródła – na stronę http://www.unicode.org/.

Znaki w programowaniu

Już po raz ostatni wracamy do kodu, by krótko omówić sposób obchodzenia się ze znakami w języku C++ w świetle przedstawionych wyżej faktów. Zaczniemy, jak zwykle, od opisania typów danych.

Pomijając kwestię unikodu typ jest właściwie jeden. Nazywa się on `char`. Można też używać zdefiniowanej w nagłówkach Windows nazwy `CHAR`. Zmienne tego typu reprezentują pojedynczy znak w kodzie ASCII i zajmują 1 bajt (8 bitów).

Pojedyncze znaki zapisuje się w C++ w apostrofach, np.:

```
char c = 'Ą';
```

Łańcuch znaków (ang. *string*) – inaczej po prostu tekst – reprezentuje w programowaniu tablica znaków lub wskaźnik do takiej tablicy. Łańcuchy zapisuje się w cudzysłowach. Przyjęło się, że automatycznie (w sposób niewidoczny dla programisty) koniec łańcucha oznaczany jest znakiem o kodzie 0.

```
char str1[] = "Zażółć gęślą jaźń";
char* str2 = str1;
std::cout << str2 << std::endl;
```

Wykonanie powyższego kodu pokazuje nam, że konsola Windows używa innej strony kodowej (takiej pokutującej jeszcze z czasów DOS), niż normalne okienka (Windows-1250).

Ponieważ deklarowanie dostatecznie dużych tablic i dbanie o ich długość nie należy do zajęć przyjemnych, a dynamiczna alokacja pamięci i żonglowanie wskaźnikami do bezpiecznych, twórcy języków programowania starają się zapewniać możliwość wygodniejszego operowania na łańcuchach. W C++ w skład biblioteki standardowej STL wchodzi zdefiniowany w nagłówku *string* typ `std::string`. Używa się go całkiem wygodnie, np.:

```
std::string str = "Błąd: Nie wykryto klawiatury!\n";
str += "Naciśnij [ESC], aby wyjść.";
MessageBox (0, str.c_str(), "Błąd", MB_OK | MB_ICONERROR);
```

Jak widać, łańcuchy można swobodnie przypisywać i można do nich dopisywać kolejne części. Obiekt `str` sam zajmuje się długością tekstu i jego przechowywaniem w pamięci.

Jego funkcja `c_str()` zwraca wskaźnik użyteczny wszędzie tam, gdzie funkcje (np. te z Win32API) oczekują łańcucha typu `char*`. Jest on na tyle inteligentny, że nie musisz zajmować się jego zwalnianiem.

Niektórzy twierdzą, że używanie takich automatycznych narzędzi spowalnia działanie programu i dlatego jest niedobre. Moim zdaniem nawet jeśli istnieje przez to jakaś utrata szybkości, chcąc pisać duże, poważne aplikacje trzeba zapomnieć o zajmowaniu się szczegółami tak elementarnych rzeczy jak operacje na łańcuchach.

Przy okazji widać tutaj także użycie jednego ze znaków specjalnych. Znaki takie wprowadza się w C++ w postaci ukośnika \ oraz odpowiedniej sekwencji (najczęściej jednego znaku). Oto ich lista:

- `\b` – cofacz (ang. *backspace*)
- `\f` – nowa strona (ang. *form feed*)
- `\n` – nowa linia (ang. *new line*)
- `\r` – powrót karetki (ang. *carriage return*)
- `\t` – tabulator poziomy (ang. *tabulator*)
- `\v` – tabulator pionowy (ang. *vertical tabulator*)
- `\a` – sygnał dźwiękowy (ang. *alarm*)
- `\\` - ukośnik (ang. *backslash*)
- `\'` – apostrof
- `\"` – cudzysłów
- `\0` – znak o kodzie 0 (NULL)
- `\?` – pytajnik
- `\###` – znak ASCII o kodzie podanym w miejscu „###” w systemie ósemkowym
- `\x##` - znak ASCII o kodzie podanym w miejscu „##” w systemie szesnastkowym

Znaki a liczby

Jeśli czytałeś uważnie pamiętasz zapewne, że `char` to typ znakowy i jednocześnie typ reprezentujący 8-bitową liczbę całkowitą. Jakie są tego konsekwencje? Można się domyślać, że zapisany w apostrofach znak to nic innego, jak liczba odpowiadająca jego kodowi.

Aby lepiej zilustrować ten fakt, popatrz na kod zamieniający cyfrę zapisaną jako znak ASCII na odpowiadającą jej wartość liczbową:

```
char Cyfra = '7';
int Liczba = Cyfra - '0';
std::cout << "Liczba wynosi: " << Liczba << std::endl;
```

W rozwiązaniu tym wykorzystałem fakt, że cyfry umieszczone są w tablicy ASCII kolejno od 0 do 9. Jednakże cyfrze 0 wcale nie odpowiada kod 0, tylko jakiś tam inny… Dlatego od kodu cyfry zapisanej w zmiennej odjąłem kod cyfry 0 i tak otrzymałem szukaną wartość liczbową.
Mam nadzieję, że rozumiesz, skąd to się wzięło?

Mały bonus

Na zakończenie tego ostatniego podrozdziału będzie mały bonus. Oto krótka, 3-linijkowa funkcja w C++:

```
f(){int k;float i,j,r,x,y=-16;while(puts(""),y++<15)for(x
=0;x++<79;putchar(" .:-;!/>)|&IH%*#"[k&15]))for(i=k=r=0;
j=r*r-i*i-2+x/25,i=2*r*i+y/10,j*j+i*i<11&&k++<111;r=j);}
```

Pochodzi ona ze wstępu do książki pt. *Perełki programowania gier, tom 3*. Skopiuj ją do programu, włącz nagłówek *cstdio* i zobacz, jaki będzie efekt :)

Podsumowanie

Tak oto dobiegła końca nasza podróż przez bity i bajty. Poznaliśmy sposób, w jaki komputer reprezentuje dane na najniższym poziomie, gromadzi je, zapisuje w pamięci, przesyła oraz przetwarza. To jest właśnie istota informatyki!

Wiem, że ręczne przeliczanie liczb na system siódemkowy i wiele innych rzeczy nie będzie potrzebne w praktyce programistycznej. Myślę jednak, że dla prawdziwego pasjonata programowania takie wiadomości wydają się po prostu ciekawe. Pamiętaj: brak konieczności zajmowania się pewnymi sprawami niskiego poziomu nie zwalnia od ich znajomości i rozumienia!