CSAPP

Lukasz Kopyto

March 14, 2024

2 Rozdzial 2. Reprezentacja i Manipulacja Danymi.

[start = 2]

2.1 Przechowywanie informacji

Komputery uzywaja 8-bitowych blokow nazywanych bajtami, jako najmniejsze adresowane jednostki pamieci. Program na poziomie maszynowym widzi pamiec jako bardzo duza tablice bajtow, zwana pamiecia wirtualna(ang. virtual memory). Kazdy bajt pamieci ma swoj unikatowy numer, nazywany adresem. Zbior wszystkich mozliwych adresow jest nazwany jako przestrzen adresow wirtualnych(virtual address space).

2.1.1 Notacja szesnastkowa/hexadecymalna

Tabelka 2.2. Trzeba ja miec w glowie.

Dziesietnie	Binarnie	Szesnastkowo
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Zwykle zapisujemy liczbe szesnastkowa z przedrostkiem 0x.

Na przyklad wezmy liczbe zapisana w systemie szesnatkowym: 0x173A4C. Chcemy ja zamienic na system binarny. Bierzemy kazda cyfre z liczby w zapisie szesnastkowym i zamieniamy ja na odpowiadajacy kwartet binarny.

Dla liczby 0x173A4C wyglada to tak: $1 \rightarrow 0001$

$7 \rightarrow 0111$

$3 \rightarrow 0011$

$A \rightarrow 1010$

$4 \rightarrow 0100$

$C \rightarrow 1100$

Po ustawieniu kolejno kwartetow, otrzymujemy liczbe w zapisie binarnym:

$$000101110011101001001100_2$$

Dla odwrotnej konwersji dziala to analogicznie. Dzielimy liczbe w zapisie binarnym na kwartety, od najmniej znaczacego bitu. Po czym zamieniamy kwartety na odpowiadajace im cyfry w zapisie szesnastkowym.

Cwiczenie 2.1

Dokonaj ponizszych konwersji

- 0x25B9D2 to binary:

  Kazda cyfre szesnastkowo zamieniamy na 'kwartet binarny':

  $$0x25B9D2 \rightarrow 001001011011100111010010$$

- binary 1010111001001001 to hexadecimal:

  Z liczby binarnej wydzielamy kwartety, poczawszy od LSB(least significent bit-najmniej znaczacy bit). Jesli liczba bitow nie jest podzielna przez 4 to dopelniamy zerami:

  $$1010111001001001 \rightarrow 1010111001001001 \rightarrow 0xAE49$$

- 0xA8B3D to binary:

  $$0xA8B3D \rightarrow 10101000101100111101$$

- binary 1100100010110110010110 to hexadecimal

  $$1100100010110110010110 \rightarrow 001100100010110110010110 \rightarrow 0x322D95$$

Kiedy x jest potega dwojki tzn $x = 2^n$ mozna latwo to przepisac na wartosc szesnastkowa pamietajac, ze binarnie x to bedzie 1 i n zer. Wiec dla n zapisanego w formie $i + 4j$ gdzie $0 \leq i \leq 3$, mozemy zapisac x w formie szesnastkowej biorac za pierwsza cyfre szesnastkowa 1 gdy i = 0, 2 gdy i = 1, 4 gdy i = 2 oraz 8 gdy i = 3 dodajac pozniej j zer(szesnastkowych). Jako przyklad spojrzmy na $x = 2048 = 2^{11}$. Mamy tu $n = 3 + 2 * 4$, zatem x szesnastkowo to 0x800.

Cwiczenie 2.2

Uzupelnij ponizsza tabelke:

n	$2^n(decimal)$	$2^n(hexadecimal)$
5	32	$2^{1+1*4} = 0x20$
23	2^{23} duzo	$2^{23} = 2^{3+5*4} = 0x800000$
15	32768	$2^{15} = 2^{3+3*4} = 0x8000$
12	4096	$2^{12} = 2^{0+3*4} = 0x1000$
6	64	$2^6 = 2^{2+1*4} = 0x40$
8	256	$2^8 = 2^{0+2*4} = 0x100$

2.2 Rozmiary danych

Kazdy komputer posiada jakis rozmiar slowa maszynowego, ktory okresla nominalny rozmiar wskaznika na dane. Poniewaz adres wirtualny jest zakodowany poprzez to slowo, najwazniejszym parametrem systemu zdeterminowanym poprzez rozmiar slowa maszynowego jest maxymalny rozmiar przestrzeni adresow wirtualnych. Zatem dla maszyny z w-bitowym rozmiarem slowa maszynowego, adresy wirtualne sa z zakresu od 0 do $2^w - 1$, co daje programowi dostep do co najwyzej 2^w bajtow.

Wiekszosc 64-bitowych maszyn moze korzystac z programow skompilowanych dla 32-bitowych maszyn. Jest to forma wstecznej kompatybilnosci.

Jezyk C wspiera roznorakie formaty danych dla calkowitych jak i zmiennoprzecinkowych danych. Tabelka 2.3 pokazuje typowoa liczbe bajtow zarezerwowanych dla roznych typow danych w C. W pozniejszym rozdziale zostanie przerobiona to co jest zagwarantowane przez standard C a co jest typowe. Dokladna liczba bajtow dla poszczegolnych typow danych zalezy od tego jak program zostal skompilowany.

Tabelka 2.3: Typowe rozmiary w bajtach standardowych typow danych w C

Liczba bajtow zarezerwowanych zalezy od tego jak program zostal skompilowany. Ponizsza tabelka pokazuje wartosci typowe dla 32 i 64 bitowych programow.

C Signed	C Unsigned	32-bit	64- bit
[signed] char	unsigned char	1	1
short	unsigned short	2	2
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char*		4	8
float		4	4
double		8	8

Aby uniknac polegania na "typowych" rozmiarach danych i roznych ustawieniach kompilatora, standard ISO C99 wprowadzil nowa klase typow danych, gdzie rozmiar danych jest ustalonyy bezwzgledu na ustawienia kompilatora oraz maszyny. Wsrod nich sa miedzy innymi int32_t oraz int64_t, ktore maja dokladnie 4 oraz 8 bajtow kolejno.

Wiekoszosc typow danych jest domyslnie ze znakiem, chyba ze poprzedzona slowem zastrzezonym unsigned lub specjalna forma deklaracji dla ustwionych typow(uint32_t etc.). Wyjatkiem jest jednak typ char. Pomimo iz wiekszosc kompilatorow i maszyn traktuje char jako wartosc ze znakiem, standard C nie gwarantuje tego. Zamiast tego, aby zagwarantowac to aby char byl 1-bajtowym typem ze znakiem, programisci powinni uzywac deklaracji [signed] tak jak zostalo to pokazane w tabelce 2.3. W wielu przypadkach program to olewa czy char jest signed czy unsigned, ale trzeba miec to na uwadze.

Jezyk C pozwala na rozne sposoby deklarowac ten sam typ oraz wprowadzac lub nie opcjonalne slowa kluczowe. Jako przyklad ponizsze deklarace maja to samo znaczenie:

- unsigned long
- unsigned long int
- long unsigned
- long unsigned int

Standardy jezyka C ustalaja dolne granice wartosci roznych typow danych(pozniej to bedzie dokladniej przerobione). Trzeba miec na uwadze, ze nie ustalaja pne gornych granic(z wyjatkiem typow z ustawionym rozmiarem).

Miedzy 1980 a 2010 32-bitowe maszyny z 32-bitowymi prgramami byly dominujace. Wiele programow zostalo napisanych z zalozeniem, ze slowo maszynowe ma 32 bity. Z zmiana na 64-bity, pojawilo sie wiele problemow. Na przyklad, wiele programistow zakladalo, ze obiekt typu int, moze byc uzywany do przechowywania wskaznika. To dziala dobrze dla wiekszosci 32-bitowych programow, ale bedzie to prowadzilo do problemow w 64-bitowych programach.

2.3 Adresowanie oraz uporzadkowanie bajtow (Addressing and Byte Ordering)

Wielo bajtowy obiekt jest przechowywany jako ciag bajtow z adresem bedacym najnizszym adresem z uzytych bajtow. Naprzyklad zalozmy ze zmienna x jest typu int i ma adres 0x100; tzn wartoscia wyrazenia adresu & x jest 0x100. Wtedy, zakladajac ze int ma 32-bitowa reprezentacje, 4 bajty na ktorych jest zapoisany x, beda przechowywane pod adresami 0x100, 0x101, 0x102, 0x103.

Zeby uporzadkowac bajty reprezentujace jakis obiekt, sa dwie znane konwencje. Rozwazmy w-bitowa liczbe calkowita, ktora ma reprezentacje w bitach $[x_{w-1}, x_{w-2}, \ldots, x_1, x_0]$, gdzie x_{w-1} jest najbardziej znaczacym bitem(MSB most significant bit) oraz x_0 bedacym najmniej znaczacym bitem(LSB least significat bit). Zakladajac ze w jest wielokrotnoscia 8, bity zmiennej x moga byc pogupowane jako bajty z najbardziej znaczacym bajtem majacym bity $[x_{w-1}, x_{w-2}, \ldots, x_{w-8}]$ oraz najmniej znaczacym bajtem majacym bity $[x_7, x_6, \ldots, x_0]$ oraz pozostalymi bajtami majacymi kolejne bity ze srodka. Konwencja ta nazywa sie **big endian**. Konwencja odwrotna, w ktorej najmniej znaczacy bity sa z przodu nazywa sie **little endian**. Zobaczmy to na przykladzie.

Zalozmy ze zmienna x typu int o adresie 0x100 ma wartosc szesnastkowa 0x01234567. Uporzadkowanie bajtow o adresach od 0x100 do 0x103 zalezy od typu maszyny:

Big endian:

...	0x100	0x101	0x102	0x103	...
...	01	23	45	67	...

oraz:

Little endian:

...	0x100	0x101	0x102	0x103	...
...	67	45	23	01	...

Zauwazmy ze w slowie 0x01234567 high-order bit ma wartosc szesnastkowa 0x01, a low-order bit ma wartosc 0x67

Uporzadkowanie bitow czasami jest problemem.

Jednym z przykladow takich problemow jest wtedy kiedy dane w formie binarnej sa przesylane w sieci pomiedzy roznymi maszynami. Problemem jest odbior danych w zapisanych w formacie big endian przez maszyne, ktora operajue w formacie little endian i vice versa. Zeby uniknac tych problemow kod napisany dla takich aplikacji musi przestrzegac pewnych konwencji. Przyjrzymy sie temu w rozdziale 11.

Drugim waznym miejscem gdzie porzadek bitow jest kluczowy, jest wtedy gdy patrzymy na ciagi bajtow reprezentujacych liczby calkowite(integer data). To sie zdarza czesto gdy patrzymy na kod maszynow programow. Jako przyklad spojrzmy na wiersz pochodzacy z pliku ktory zwraca w formie tekstu kod maszynowy dla procesora Intel x86-64:

4004d3: 01 05 43 0b 20 00 add %eax,0x200b43(%rip)

Wiersz ten zostal wygenerowany przez disassembler, narzedzie sluzace do okreslenia ciagu instrukcji reprezentowanych przez plik wykonywalny(exeucatble file). Wiersz ten mowi nam, ze szesnastkowa wartosc ciagu bitow: 01 05 43 0b 20 00 bitowa reprezentacja instrukcji ktora dodaje jedno slowo danych do wartosci przechowywanej pod adresem obliczonym przez dodanie 0x200b43 do aktualnej wartosci 'program counter'- licznik programu?, adresu nastepnej instrukcji do wykonania.

Jesli wezmiemy ostatnie 4 bajty z ciagu 43 0b 20 00 i zapiszemy je odwrotnie, dostaniemy 00 20 0b 43. Ignorujac poczatkowe zera otrzymamy wartosc 0x200b43, czyli wartosc liczbowa po prawet stronie instrukcji. Wystepowanie bajtow w odwrotnej kolejnosci jest czestym zjawiskiem przy czytaniu kodu maszynowego wygenerowanego dla little-endian maszyny, takiej jak ta powyzej. Naturalny sposob na zapisanie bajtow jest taki, ze najnizej numerowany bit jest po lewej stronie a najwyzej numerowany jest po prawej. Jest to sprzeczne z normalnym zapisemliczb, gdze najbardziej znaczaca cyfra jest po lewej a najmniej znaczaca po prawej.

Listin 2.4(ponizej) pokazuje kod w C, ktory uzywa rzutowania aby dobrac sie i wyswietlic bajtowa reprezentacje roznych obiektow w programie. Uzywamy typedef aby zdefiniowac typ byte_pointer jako wskaznik na obiekt typu unsigned char. Wskaznik ten odnosi sie do ciagu bajtow gdzie kazdy bajt jest interpretowany jako nieujemna liczba calkowita. show_bytes bierze jako pierwszy argument adres ciagu bajtow, reprezentowany jako byte_pointer, na drugim miejscnu jest "licznik bajtow". Jest on zdefiniowany jako obiekt typu size_t- preferowany typ do reprezentowania rozmiarow roznych struktur i obiektow(sizeof go zwraca). show_bytes drukuje nam pojedyncze bajty w reprezentacji szesnastkowe-¿ wyswietla 2 cyfry, ktore nalezy interpretowac jako wartosc szesnastkowa-¿ kazda z nich ma 4 bity-¿ razem daja 2*4 bity = 8bitow = 1 bajt

Funkcje show_int, show_float, show_pointer pokazuja nam jak uzywac funkcji show_bytes aby wyswietlic bajtowe reprezentacje obiektow programu w C typu int, float oraz pointer(wskaznik, void *). Zauwazmy ze przekazujemy funkcji show_bytes wskaznik &x argumentu ktory przekazalismy, nastepnie rzutujemy ten wskaznik na typ unsigned char *. Rzutowanie mowi kompilatorowi(i nam) aby interpretowac ten wskaznikk jako ciag bajtow a nie jako obiekt typu ktory pierwotnie przekazalismy do funkcji. Wskaznik ten, bedzie adresem "najnizszego" bajtu obiektu ktory przekazalismy.

Procedury te uzywaja wbudowanej w C funkcji sizeof aby okreslic liczbe bajtow uzywanych przez obiekt. W ogolnosci sizeof(T) zwraca liczbe bajtow potrzebnych aby przechowac obiekt typu T. Uzywajac sizeof zamiast sztywnej liczby bajtow sprawiamy ze nasz program jest w jakims stopniu przenosny.

2.4 Listing

Kod sluzacy do wyswietlenia bajtowej reprezentacji obiektow programu.

Uzywa rzutowania aby "obejsc" system typow.

```
#include <stdio.h>

typedef unsigned char *byte_pointer;

void show_bytes(byte_pointer start, size_t len)
{
    int i;
    for (i = 0; i < len; i++)
        printf(" %.2x", start[i]);
    printf("\n");

}
```

```
void show_int(int x)
{
    show_bytes((byte_pointer) &x, sizeof(int));
}

void show_float(float x)
{
    show_bytes((byte_pointer) &x, sizeof(float));
}

void show_poiner(void *x)
{
    show_bytes((bytes_pointer) &x, sizeof(void *x));
}
```

2.5 Listing

Kod ponizej wyswietla bajtowa reprezentacje przykladowych obiektow danych typow.

```
void test_show_bytes(int val)
{
    int ival = val;
    float fval = (float) ival;
    int *pval = &ival;
    show_int(ival);
    show_float(fval);
    show_pointer(pval);
}
```

Jak odpalimy ten program a dokladniej jak odpalimy funkcje show_bytes(x), gdzie x to int x = 12345, dostaniemy takie cos na wyjsciu:

Wartosc	Typ	Bajty(szesnastkowo)
12345	int	39 30 00 00
12345	float	?
12345	int *	?