Zawartość:

- rozmiary danych

- typy całkowite (zakresy, przepełnienie, przekręcenie zakresu)

- typy zmiennoprzecinkowe

- typ znakowy

- przeliczanie systemów znakowych

Rozmiary danych:

1 bit – przyjmuje wartości 1 albo 0

1 bajt – 8 bitów

1 kilobajt – 1024 bajty

1 megabajt – 1024 megabajty itd.

Każdy bajt w pamięci operacyjnej ma swój adres. W systemie 32 bitowym maksymalna liczba możliwa do zapisania to (2^32)-1 co odpowiada dokładnie 4GB adresów. Stąd ograniczenie do 4GB ramu w komputerach 32 bitowych. Komputery 64 bitowe również mają ograniczenie wbrew powszechnej opinii lecz jest ono tak duże, że ludzie twierdzą, że go nie ma.

(2^32)-1 to 4GB

(2^33)-1 to 8GB

…

(2^36)-1 to 64GB

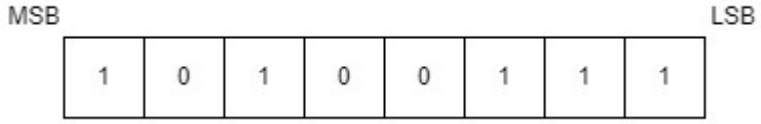
…

(2^42)-1 to 4TB

itd

Pojęcie MSB oraz LSB:

Definicja dla bitów:



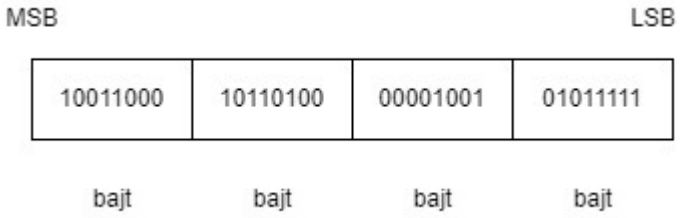
MSB – Most Significant Bit (najbardziej znaczący bit)

LSB – Least Significant Bit (najmniej znaczący bit)

Definicja dla bajtów:

Powyższe dwa określenia często stosowane są także w kontekście bajtów wielobajtowej wartości. Np. unsigned int ma 4 bajty, więc dolny bajt będzie LSB, a górny MSB.

Liczba 2 561 935 711 zapisana w zmiennej unsigned int ma następującą postać binarną:



Jak widać MSB to najstarszy bajt.

Przeliczanie systemów:

BIN – system binarny/dwójkowy

DEC – system dziesiętny/decymalny

OCT – system ósemkowy/oktagonalny

HEX – system szesnastkowy/hexagonalny

DEC >>> BIN

12 : 2 = 6 reszta 0

6 : 2 = 3 reszta 0

3 : 2 = 1 reszta 1

1 : 2 = 0 reszta 1

0

Czytamy od dołu do góry. Liczba 12 DEC to 1100 BIN.

BIN >>> DEC

Zamiana 0b1100 BIN na DEC.

0*2^0 + 0*2^1 + 1*2^2 + 1^2^3 = 0 + 0 + 4 + 8 = 12

Inna metoda:

Załóżmy, że chcemy zamienić liczbę 0b10100111 na system dziesiętny:

Każda pozycja idąc od prawej do lewej ma dwa razy większą wagę niż poprzednia, pierwsza po prawej ma wagę 1.

Waga:	128	64	32	16	8	4	2	1
Bit:	1	0	1	0	0	1	1	1

Wystarczy tylko dodać wagi gdzie bit jest równy 1: 128+32+4+2+1=167

DEC >> HEX

Zamiana liczby 1583 DEC na HEX:

1583 : 16 = 98 reszta 15 (F)

98 : 16 = 6 reszta 2

6 : 16 = 0 reszta 6

1583 DEC to 62F HEX

HEX >> DEC

Zamiana 62F HEX na DEC

15(F)*16^0 + 2*16^1 + 6*16^2 = 15 + 32 + 1536 = 1583

BIN >> HEX oraz HEX >> BIN

Załóżmy, że chcemy przeliczyć liczbę 0x8A na system dwójkowy:

Nybel – jedna cyfra w systemie szesnastkowym. Powyżej mamy dwa nyble 8 oraz A. Każdy nybel to 4 bity, więc wystarczy przeliczyć każdy nybel na system dwójowy i skleić wynik.

Nybel: 8 A

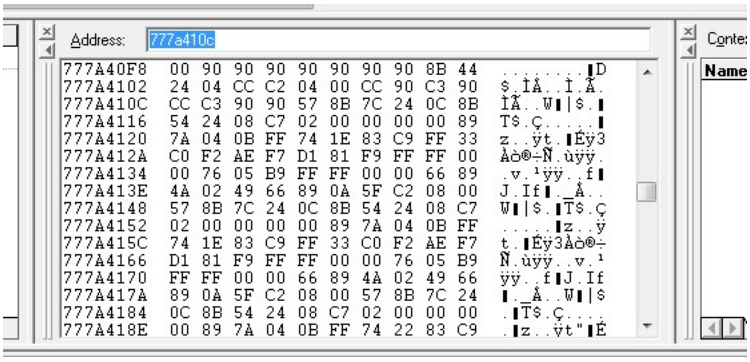
Wartość dwójkowa: 0b1000 0b1010

Wynik:0b10001010

W drugą stronę działa to analogicznie. Dzielimy na grupy po 4 bity (od prawej strony zaczynając oczywiście)

Przeliczanie systemów, jest bardzo ważne w programowaniu (np. w programowaniu elektroniki) oraz w sieciach (liczenie adresacji).

W informatyce często stosuje się zapis szesnastkowy, ponieważ lepiej widać w nim podział na bajty. Bajt to po prostu dwie cyfry/nyble.



Typy danych:

Typ danych to tak naprawdę informacja dla języka/procesora jak ma interpretować dane zapisane w pamięci (bity bajty itd.). Tak naprawdę każdy typ danych prędzej czy później jest liczbą. Nawet kod jest liczbą od pewnego momentu.

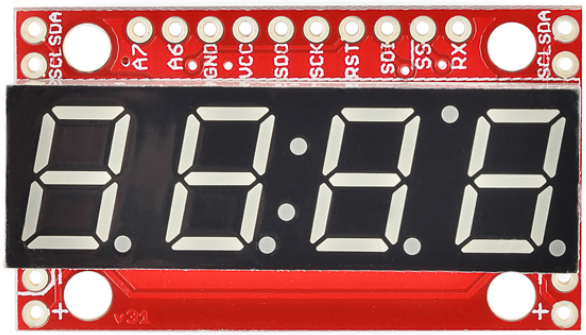
Liczby całkowite:

unsigned	signed
Char	
Zakres: 0 do 255, 0x0 do 0xFF, 0b00000000 do 0b11111111, 0 do (2^8)-1	Zakres: -128 do 127, -0x80 do 0x7F, 0b10000000 0b01111111
8 bitów na wartość	1 bit msb na znak (1 oznacza minus) i 7 bitów lsb na wartość
Short	
Zakres: 0 do 65535, 0x0 do 0xFFFF	Zakres: -32768 do 32767, 0x8000 do 0x7FFF
16 bitów na wartość	1 bit msb na znak (1 oznacza minus) i 15 bitów lsb na wartość
Int/Long	
Zakres: 0x0 do 0xFFFFFFFF	Zakres: 0x80000000 do 0x7FFFFFFF
32 bitów na wartość	1 bit msb na znak (1 oznacza minus) i 31 bitów lsb na wartość
Long Long (64 bity)	
Zakres: 0x0 do 0xFFFFFFFFFFFFFFFF	Zakres: 0x8000000000000000 do 0x7FFFFFFFFFFFFFFF
64 bitów na wartość	1 bit msb na znak (1 oznacza minus) i 63 bitów lsb na wartość

Uwaga 1: w trybie 64 bitowym nie do końca tak jest, że wszystko jest 64 bitowe

Uwaga 2: już nie ma sensu pisać zakresów w każdym systemie, podanie w hexie jest najbardziej wygodne

Co się stanie jak wpiszemy większą wartość niż dopuszcza to typ? Zostanie skrócona tak by zmieściła się w tylu bajtach ile dopuszcza dany typ. Dobrym przykładem tłumaczącym to zjawisko jest ekran 7 segmentowy puszczający np. 4 znaki:



Co się stanie jeżeli wpiszemy na niego 1234. Wyświetli się 1234. Co się stanie jak wpiszemy 12345? Wyświetli się 2345 bo 1 się już nie zmieści.

Co się stanie jak wyświetlimy 9999 i zrobimy na tej liczbie +1? Powstanie nam liczba 1000, ale wyświetli się 0000 bo 1 się nie mieści.

Przykład kodu:

unsigned char a = 0xABCD; //a przyjmie wartość 0xCD bo unsigned char ma 1 bajt

```
test.cpp > main()
1   #include <cstdio>
2
3   int main(){
4       unsigned char a = 0xABCD;
5       printf("%X\r\n",a);
6   }

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER  .NET INTERACTIVE

PS C:\Users\siedl\Desktop\Trash> g++ .\test.cpp -o test.exe
.\test.cpp: In function 'int main()':
.\test.cpp:4:23: warning: unsigned conversion from 'int' to 'unsigned char' changes value from '43981' to '205' [-Woverflow]
    4 |     unsigned char a = 0xABCD;
      |                       ^~~~~~
PS C:\Users\siedl\Desktop\Trash> .\test.exe
CD
PS C:\Users\siedl\Desktop\Trash>
```

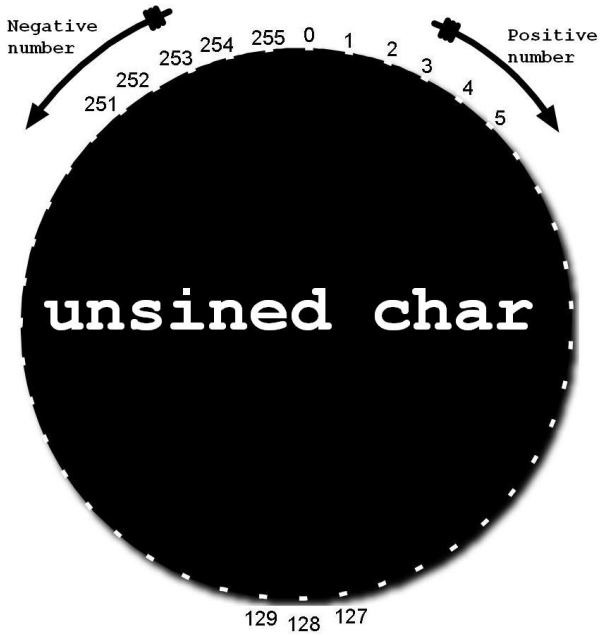
unsigned short b = 0xF0000; //b przyjmie wartość 0 bo unsigned short ma 2 bajty

```cpp
#include <cstdio>

int main(){
    unsigned short b = 0xF0000;
    printf("%X\r\n",b);
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER  .NET INTERACTIVE

PS C:\Users\siedl\Desktop\Trash> g++ .\test.cpp -o test.exe
.\test.cpp: In function 'int main()':
.\test.cpp:4:24: warning: unsigned conversion from 'int' to 'short unsigned int' chang
    4 |     unsigned short b = 0xF0000;
      |                        ^~~~~~~
PS C:\Users\siedl\Desktop\Trash> .\test.exe
0
PS C:\Users\siedl\Desktop\Trash>
```

Z unsigned int jest tak samo tylko tam zakres jest 32 bitowy (4 bajty). Liczby signed działają również tak samo.

Pojęcie przekręcenia zmiennej:



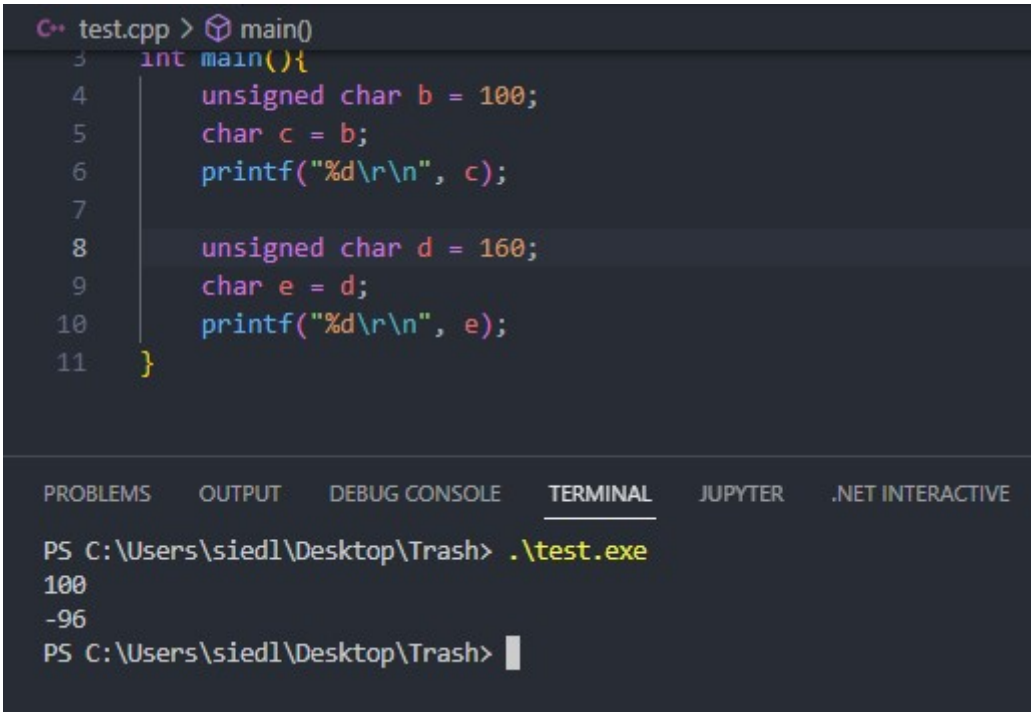
*unsigned ☺

Gdy pomniejszymy o 1 zmienną unsigned char o wartości 0 to zakres przeskoczy na swój drugi koniec. Mając uchara z wartością 255, dodając 1 przeskoczymy na 0. Każdy typ liczbowy tak ma. Oczywiście zakresy są różne.

Co się stanie jak licząc przekroczymy zakres danych? Zmienna przekręci się jak w kółko.

Dodając coś do wartości unsigned char równej 255 wróciliśmy na początek zakresu jak w kole powyżej. Odejmując przeszliśmy na koniec. Taka sama zasada panuje w większych typach short oraz int.

Mieszanie liczb ze znakiem i bez znaku:



Zakres unsigned chara to 0 do 255 a signed chara -128 do 127. Liczba 100 mieści się w obu przedziałach, ale liczba 160 już tylko w unsigned.

W iostream zdefiniowano pewne stałe, które przechowują minimum i maksimum każdego typu całkowitego:

```
//zakresy
printf("char %d %d\r\n", CHAR_MIN, CHAR_MAX); //posłużyłem się tu makrami predefiniowanymi w iostream
printf("short %d %d\r\n", SHRT_MIN, SHRT_MAX);
printf("int %d %d\r\n", INT_MIN, INT_MAX);
printf("long %ld %ld\r\n", LONG_MIN, LONG_MAX);
printf("long long %lld %lld\r\n", LLONG_MIN, LLONG_MAX);

printf("unsigned char %u %u\r\n", 0, UCHAR_MAX);
printf("unsigned short %u %u\r\n", 0, USHRT_MAX);
printf("unsigned int %u %u\r\n", 0, UINT_MAX);
printf("unsigned int %u %u\r\n", 0, ULONG_MAX);
printf("unsigned long long %llu %llu\r\n", 0LL, ULLONG_MAX);
```

Bardzo dużą błędów w oprogramowaniu bierze się, ze źle dobranego typu danych. Przez co następuje przepełnienie powodujące jakiś błąd czasem poważniejszy czasem nie.

Optymalizacja?

Jeżeli wiemy jaki zakres danych nam potrzebny dobieramy odpowiedni typ danych. Jeżeli nie wiemy to dobieramy jak największy.

Przykład z ocenami.

Zapisanie oceny 1-5 w int zajmie nam 4 bajty pamięci a w charze dokładnie 1. Oceny mieszczą się w zakresie 0-255 a nawet -128 do 127. Proste.

Liczby całkowite ujemne

Liczby całkowite ujemne kodowane są w systemie U2 czyli systemie uzupełnień do 2.

Aby zapisać np. -14 w bajcie jako liczba ujemna należy wykonać następujące kroki:

Pierwszy krok to zamiana dodatniej 14 na system binarny:

00001110 BIN = 14 DEC (wyrównane do 8bitów czyli bajta)

Drugi krok to negacja bitów:

00001110 -> 11110001

Trzeci krok to dodanie 1 to uzyskanej liczby:

11110001 + 00000001 = 11110010

-14 = 0b11110010

Typ znakowy char

Utarło się przekonanie, że typ char jest znakiem. Nie jest to prawda a przynajmniej nie do końca. Char jest liczbą o rozmiarze bajtu. Po prostu znaki alfanumeryczne są zakodowane za pomocą liczb. Np. 97 DEC to mała literka ‘a’. Pełna tablica takiego kodowania wygląda tak:

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Liczby od 0 do 31 mają specjalne funkcje. W tej grupie znajduje się znak nowej linii ‘\n’, tabulacja ‘\t’ czy powrót karetki ‘\r’.

Mając liczbę np. 97 w zmiennej typu unsigned char, możemy ją potraktować jako zwykłą liczbę 97 lub też zinterpretować ją jako ‘a’.

Oczywiście istnieją znaki powyżej liczby 127. Są to znaki diakrytyczne różnych języków oraz przeróżne symbole.

Typ zmiennoprzecinkowy float/double

Liczby zmienno przecinkowe kodowane są w systemie IEEE754.

Aby zapisać liczbę zmienno przecinkową w pamięci trzeba ją zakodować w IEEE754.

Pierwszy krok to konwersja liczby na system binarny:

14.75 DEC = 1110.11

14 : 2 = 7 r 0

7 : 2 = 3 r 1

3 : 2 = 1 r 1

1 : 2 = 0 r 1

(tutaj czytamy od dołu)

0.75 * 2 = 1.5 całości 1

0.5 * 2 = 1 całości 1

(tutaj czytamy od góry)

Drugi krok to przesunięcie przecinka tak by po jego lewej stronie była tylko jedna cyfra:

1.11011

Przesunęliśmy przecinek o 3 pozycje w lewo.

Kolejny krok to określenie trzech elementów:

znak = 0 (bo liczba jest dodatnia)

cecha = 127 + 3 (o 3 pozycje przesunięto przecinek)

mantysa = 111011 (aktualna wartość po przecinku)

Wynik:

0 130 111011

0 10000010 11011000000000000000000

01000001011011000000000000000000 = 14.75 (zakodowane w 32 bitach IEEE754)

Konwersja w drugą stronę przebiega dokładnie odwrotnie. Istnieje wersja 64bitowa (double). W niej cecha ma 10bitów (dodajemy nie 127 a 1023) a mantysa 53 bity.