Agenda

Tools for C programming: C Tutor, GDB, and Valgrind

Binary and data representations

Unsigned integers

Signed integers

Addition and subtraction

Overflow

Signed extension

GDB/Valgrind Demo: bigfish.c

```
// allocate space for two int arrays
bigfish = (int *)malloc(sizeof(int)*10);
littlefish = (int *)malloc(sizeof(int)*10);
for (i=0; i < 10; i++) {
   bigfish[i] = 10+i;
   littlefish[i] = i;
}
print_array(bigfish,10, "bigfish");
print_array(littlefish,10, "littlefish");
for (i=0; i < 13; i++) {
   bigfish[i] = 66+i;
}
printf("\nafter loop:\n");
print_array(bigfish,10, "bigfish");
print_array(littlefish,10, "littlefish");
```

GDB/Valgrind Demo : bigfish.c

```
// allocate space for two int arrays
bigfish = (int *)malloc(sizeof(int)*10);
littlefish = (int *)malloc(sizeof(int)*10);
for (i=0; i < 10; i++) {
   bigfish[i] = 10+i;
   littlefish[i] = i;
}
print_array(bigfish,10, "bigfish");
print_array(littlefish,10, "littlefish");
for (i=0; i < 13; i++) {
   bigfish[i] = 66+i;
}
printf("\nafter loop:\n");
print_array(bigfish,10, "bigfish");
print_array(littlefish,10, "littlefish");
```

```
bigfish array:
 10  11  12  13  14  15  16  17  18  19
littlefish array:
  0   1   2   3   4   5   6   7   8   9

after loop:
bigfish array:
 66  67  68  69  70  71  72  73  74  75
littlefish array:
 78   1   2   3   4   5   6   7   8   9
Segmentation fault (core dumped)
```

Use gdb/valgrind to find!

GDB/Valgrind Demo: badprog.c

```
int findAndReturnMax(int *array1, int len, int max) {
   int i;
   if (!array1 || (len <=0) ) {
      return -1;
   }
   max = array1[0];
   for (i=1; i <= len; i++) {
      if (max < array1[i]) {
         max = array1[i];
      }
   }
   return 0;
}
```

```
int main(int argc, char *argv[]) {
   int arr[5] = { 17, 21, 44, 2, 60 };
   int max = arr[0];

   if ( findAndReturnMax(arr, 5, max) != 0 ) {
      printf("strange error\n");
      exit(1);
   }
   printf("max value in the array is %d\n", max);
   return 0;
}
```

What is the output of this program supposed to be?

Binary and Data Representation

Data is stored as **binary** signals

e.g. they can either be **on** or **off**

Each signal corresponds to a single **bit**

All data can be represented with bits

more complicated data -> needs more bits

Binary and data representation

Smallest unit of addressable memory is a **byte**

Memory address 0: 01010101
Memory address 1: 10101010
Memory address 2: 00001111

A byte is **8 bits**

A **word** is the default size of memory that the hardware moves around

either 32 bits or 64 bits

Why define variables? Why have types?

Variable types in C

- Different Types have different number of bytes:

```
1 byte:  char, unsigned char (no negative values)
2 bytes: short, unsigned short
4 bytes: int, unsigned int, float
8 bytes: long long, unsigned long long, double
4 or 8 bytes: long, unsigned long
```

NOTE: On a 64 bit machine, pointers are 8 bytes

Example: Memory can be interpreted in different ways depending on the context

Consider the data

0b01001100011011111101001100 (0x4C6F4C)

The above bits can mean any of the following

LOL

5,009,228

Example: ASCII

American Standard Code for Information Interchange

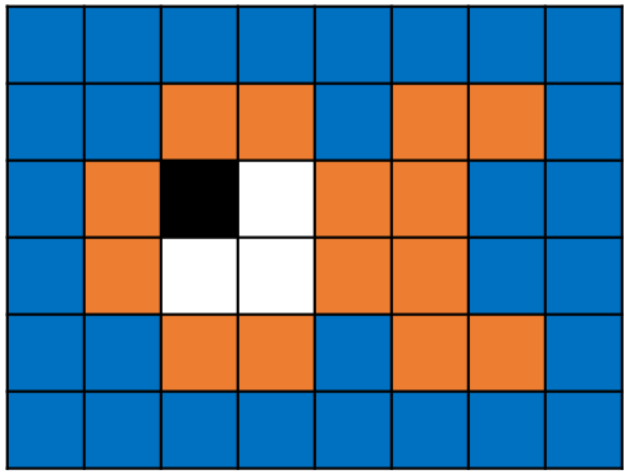
Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of 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

Example: Simple image

Binary Interpretation:

00	White	01	Orange
10	Blue	11	Black



(a)

10	10	10	10	10	10	10	10
10	10	01	01	10	01	01	10
10	01	11	00	01	01	10	10
10	01	00	00	01	01	10	10
10	10	01	01	10	01	01	10
10	10	10	10	10	10	10	10

(b)

10101010	10101010
10100101	10010110
10011100	01011010
10010000	01011010
10100101	10010110
10101010	10101010

(c)

Figure 2. The (a) image representation, (b) two-bit cell representation, and (c) byte representation of a simple fish image. *Dive Into Systems*

Number bases and unsigned integers

Recall: Decimal numbers

$$5163 = 5 * 10^3 + 1 * 10^2 + 6 * 10^1 + 3 * 10^0$$

What is the general formula?

Notation

0b (zero-b) denotes a binary number, e.g 0b 1001

0x (zero-x) denotes a hexadecimal number, e.g. 0xE3

0 (zero) denotes an octal number, e.g. 0644

d_0 denotes the lowest order bit

d_{N-1} denote the highest order bit

Other popular notation: $74_{10} = 7 * 10^1 + 4 * 10^0$

Hexadecimal

Base 16: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Compact way to represent binary numbers

Octal (base 8) is sometimes used

Dec	Bin	Hex	Dec	Bin	Hex	Dec	Bin	Hex	Dec	Bin	Hex
0			4			8			12		
1			5			9			13		
2			6			10			14		
3			7			11			15		

Exercise: Hexadecimal

Convert this binary number to hexadecimal

0b010011000110111101001100

Convert this hexadecimal number to binary

0x8045EF

Hexadecimal to decimal

Convert 0x3CD0 to decimal

Decimal to hexadecimal

Idea: Compute digits from highest order to lowest order

Example: divide by 16^4, then 16^3, then by 16^2, etc

Each step i from N-1 to 0 do

i^{th} digit = floor(input/16^i)

input = input % 16^i

Exercise: Convert 9742_{10} to hexadecimal

Digit Place	Input	Divisor	Digit (/)	Remainder (%)
d4		16^4		
d3		16^3		
d2		16^2		
D1		16^1		
d0		16^0		

Decimal to binary

Can use the same approach, but there is an easier way: **repeated division**

Idea: Repeatedly divide by 2 and check the parity

Each step i from 0 to N-1 do

i^{th} digit = input % 2

input = floor(input / 2)

Exercise: Convert 422_{10} to binary

Unsigned integers

Numbers ranging from 0 to a positive max value

Example: Represent 4 using a 4-bit unsigned integer

Example: Represent 34 using a 1 byte unsigned integer (e.g. a char)

Unsigned integer ranges

What is the largest number that can be stored in 4 bits?

What is the largest number that can be stored in 4 bytes?

Signed integers

Modern systems use a method called **two's compliment**

highest order bit encodes the sign (0 -> positive; 1 -> negative)

advantage: allows pos/neg numbers to be treated the same in hardware

Two’s compliment

Suppose we have N bits to represent a signed integer. The formula is

$$-(d_{N-1} \times 2^{N-1}) + (d_{N-2} \times 2^{N-2}) + \ldots + (d_2 \times 2^2) + (d_1 \times 2^1) + (d_0 \times 2^0)$$

^ note the leading negative sign for just the first term!

Example: What is 1001 interpreted as a *signed* integer? As an *unsigned* integer?

Exercise: Signed integers

Convert the following *signed* integer to decimal: 0b 0110

Convert the following *signed* integer to decimal: 0b 1111

Negation

To negate a two's compliment signed integer

flip the bits

add one

Example: Compute -5 as a 4-bit signed integer

Question

If we use N bits, what is the range of unsigned integers we can represent?

If we use N bits, what is the range of signed integers we can represent?

Binary addition

Works like decimal addition: we carry over values when we reach our max digit.

Example: Add 2 + 8 as 4 bit unsigned binary numbers

Binary addition

Possible outcomes when considering two binary digits and 1 carry digit

Inputs				**Outputs**	
Digit A	Digit B	Carry in		Sum	Carry out
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

Binary subtraction

Idea: X – Y is the same as X + (-Y)

Therefore, negate the second operand and then add

Example: Compute 7-2 as unsigned 4-bit integers

Unsigned Overflow

When we try to store a value too large to fit into a data type, we get **overflow.**

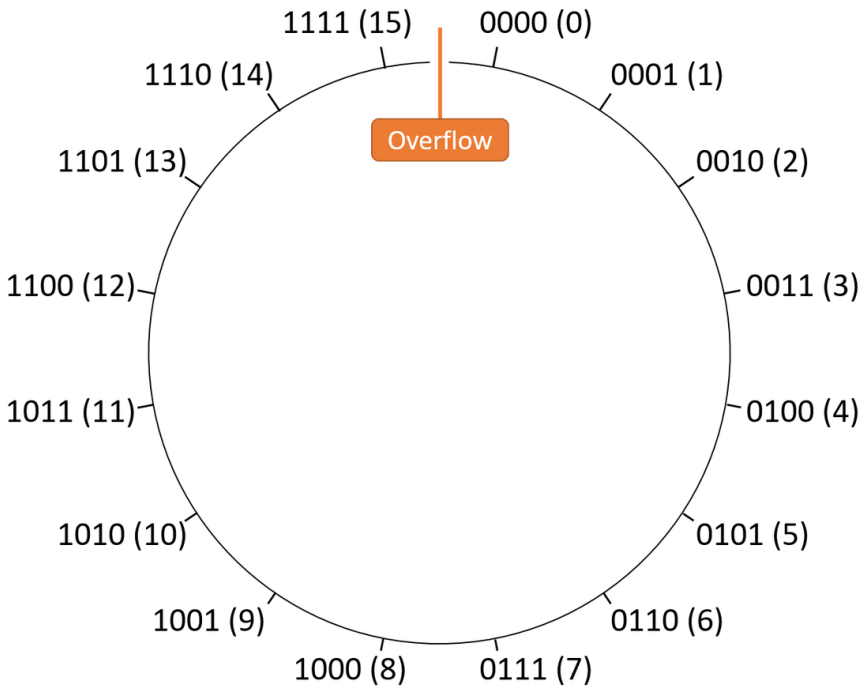


Figure 3. An arrangement of four-bit unsigned values into a modular space. All arithmetic is modular with respect to 2^4 (16).

Example: Add 12 + 7 as unsigned 4 bit numbers

Example: Add 2 - 3 as unsigned 4 bit numbers

Unsigned Overflow

Rule: If the carry-out doesn't match the carry-in, the computation has overflowed

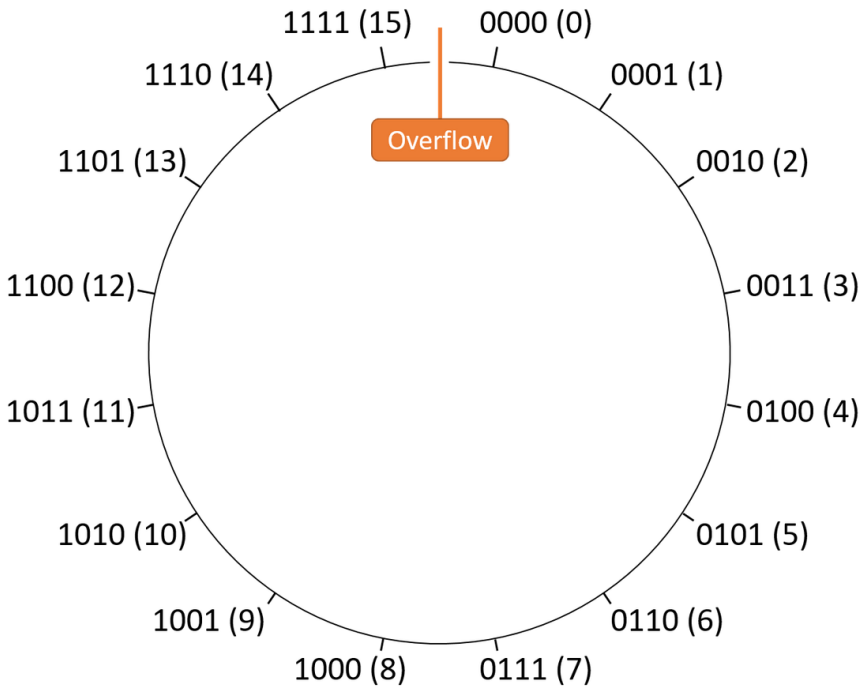


Figure 3. An arrangement of four-bit unsigned values into a modular space. All arithmetic is modular with respect to 2^4 (16).

When the carry-in = 0, we are adding and so the result should be *larger*. However, when the carry-out = 1, the result will be *smaller*.

When the carry-in = 1, we are subtracting and so we want the result to be *smaller.* However, when the carry-out = 0, the result will be *larger.*

Demo: What is the output of this program?

```
#include <stdio.h>

int main() {
   unsigned int a = 0;
   for (a = 5; a >= 0; a--) {
      printf("Message!\n");
   }
   return 0;
}
```

Signed overflow

When we try to store a value too large OR too small to fit into a data type, we get **overflow.**

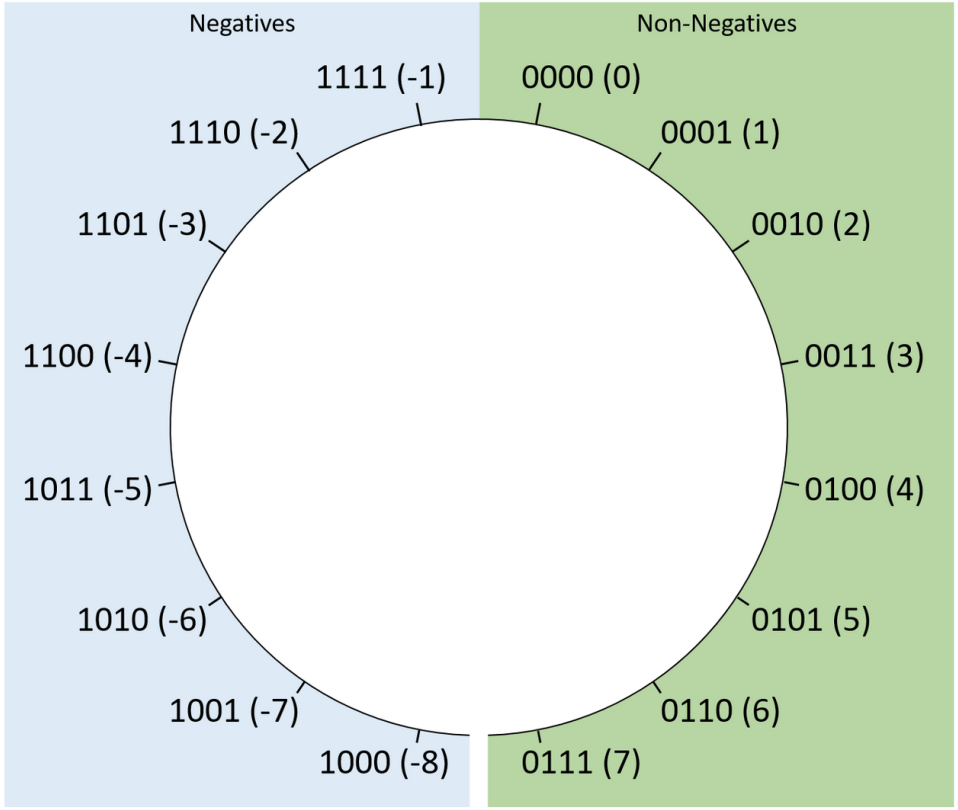


Figure 36. A logical layout of two's complement values for bit sequences of length four.

Example: Compute -6 - 3 as signed 4 bit numbers

Example: Compute 5 + 3 as signed 4 bit numbers

Signed overflow

Rule: When the operands have different sign, overflow is impossible.

When the operands have the same sign, overflow occurs when the highest order bit of the result does not match the operand.

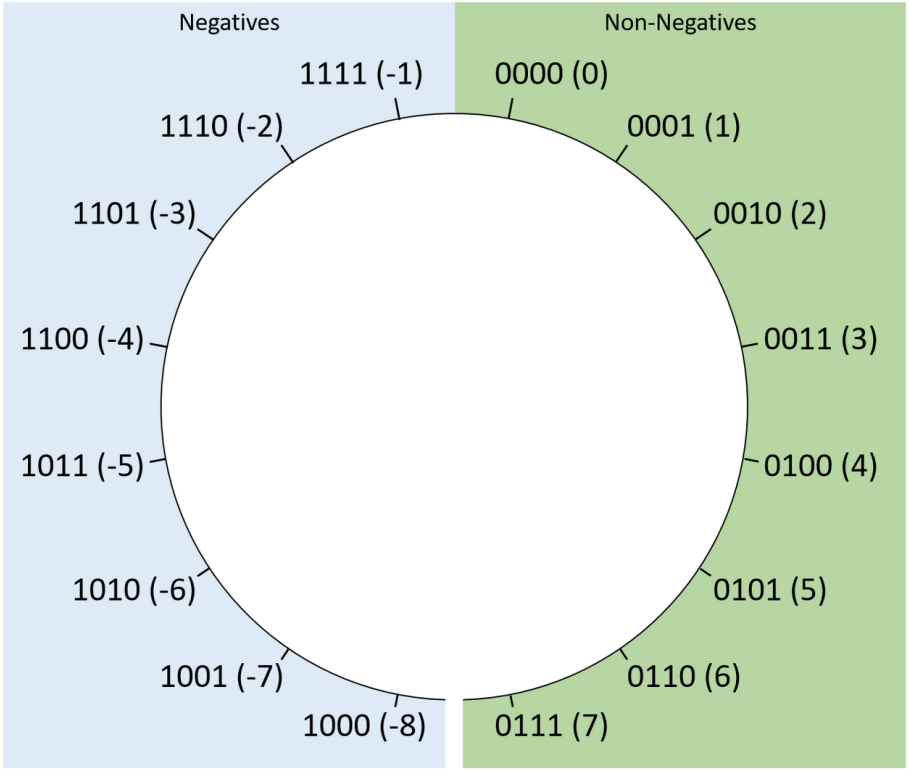


Figure 36. A logical layout of two's complement values for bit sequences of length four.

Idea: Moving towards zero is safe with signed integers.

Exercise: Compute 3 - 2 as signed 4-bit numbers

What if you used unsigned 4-bit data types instead?

Exercise: Compute 3 - 4 as signed 4-bit numbers

What if you used unsigned 4-bit data types instead?

Exercise: Add 7 + 3 as signed 4-bit numbers

What if you used unsigned 4-bit data types instead?

Signed extension

What happens when you perform an arithmetic operation on numbers with different sizes? **signed extension**

For *unsigned* values, prepend 0

For *signed* values, prepend the leftmost bit

Example: Signed extension

Add 5 and -5 as 8 bit numbers