Data Representation and Memory

September 5, 2019

Mentor Sessions

- Monday, Wednesday, and Thursday 7-9 pm
- Sunday 3-5 pm
- In Edmunds 105 … or Edmunds 219

Today

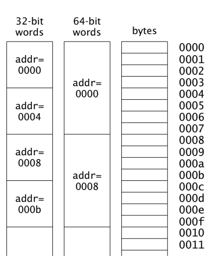
- Data, memory and the C language
  - Arrays and pointers
  - Casts
  - Typedefs
  - Structs
  - Memory Management

The C Language

- Syntax like Java: declarations, **`if`, `while`, `return`**
- Data and execution model are "closer to the machine"
  - More power and flexibility
  - More ways to make mistakes
  - Sometimes confusing relationships
  - Pointers!!
- A possible resource from CMU:
  - http://www.cs.cmu.edu/afs/cs/academic/class/15213-s16/www/recitations/c_boot_camp.pdf

Memory

- Memory is a (very large) array of bytes
- The location of a byte is its *virtual address*
- Larger words (32- or 64-bit) are stored in contiguous bytes
  - The address of a word is the address of its first byte
  - Successive addresses differ by word size



Example Data Representations

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	8	8
`long long`	8	8	8
`float`	4	4	4
`double`	8	8	8
pointer	4	8	8

Examining Data Representations

```
typedef unsigned char *pointer;

void show_bytes(pointer start, int length) {
   int i;
   for (i = 0; i < len; i++)
      printf("0x%p\t0x%.2x\n", start+i, start[i]);
}

int main() {
   int a = 15213;
   show_bytes ((pointer)&a, sizeof(int));
   return 0;
}
```

Print directives
%p: pointer
%x: hexadecimal

Output

```
0x7fff5fbffa1c 0x6d
0x7fff5fbffa1d 0x3b
0x7fff5fbffa1e 0x00
0x7fff5fbffa1f 0x00
```


Examining Data: The Full Program

```
#include <stdio.h>

typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len) {
  size_t i;
  for (i = 0; i < len; i++)
    printf("%p\t0x%.2x\n", start+i, start[i]);
  printf("\n");
}

int main() {
  int a = 15213;
  printf("int a = 15213;\n");
  printf("int a = 0x%.8x;\n",a);
  show_bytes((pointer) &a, sizeof(int));
  return 0;
}
```


Examining Data: The Result

```
pom-itb-cs2:tmp 16$ gcc -o showbytes showbytes.c

pom-itb-cs2:tmp 17$ ./showbytes
int a = 15213;
int a = 0x00003b6d;
0x7fff5e1f7b48   0x6d
0x7fff5e1f7b49   0x3b
0x7fff5e1f7b4a   0x00
0x7fff5e1f7b4b   0x00

pom-itb-cs2:tmp 18$
```


Examining Data Representations, continued

```
0x7fff5fbffa1c 0x6d
0x7fff5fbffa1d 0x3b
0x7fff5fbffa1e 0x00
0x7fff5fbffa1f 0x00
```

a = 15213 = 0x 00 00 3B 6D

x86 processors are *little endian,* with the least significant byte at the lowest address.

Some other processors are *big endian,* with the *most* significant byte at the lowest address.

Reading "Byte-reversed" listings

- A debugger converts binary machine code into assembly language. Here is a fragment with an embedded constant.

Address	Instruction Code	Assembly Rendition
8048365:	5b	pop %ebx
8048366:	81 c3 ab 12 00 00	add $0x12ab,%ebx
804836c:	83 bb 28 00 00 00 00	cmpl $0x0,0x28(%ebx)

- x86 instructions are between 1 and 15 bytes long. Other processors have fixed-length instructions

Unsigned and Signed Integers

- Use w-bit words; w can be 8, 16, 32, or 64
- The bit sequence $b_{w-1} \ldots b_1\ b_0$ represents an integer

	unsigned	signed
value	$\sum_{i=0}^{w-1} b_i 2^i$	$-b_{w-1}2^{w-1} + \sum_{i=0}^{w-2} b_i 2^i$
smallest	0	-2^{w-1}
largest	$2^w - 1$	$2^{w-1} - 1$

- Important!! "signed" does not mean "negative"

Example: Three-bit integers

unsigned		signed
111	7	
110	6	
101	5	
100	4	
011	3	011
010	2	010
001	1	001
000	0	000
	−1	111
	−2	110
	−3	101
	−4	100

- The high-order bit is the *sign bit.*
- The largest unsigned value is $11\ldots1$, UMax.
- The signed value for -1 is always $11\ldots1$.
- Signed values range between TMin and TMax.

This representation of signed values is called *two's complement*.

Arithmetic, Part 1

- Usual addition and subtraction
  - Like you learned in second grade, only binary
  - Same for unsigned and signed
  - … but error conditions differ
- To negate a signed value: complement the bits and add 1

Reason: $x + \sim x = 11\ldots1 = -1$, so $x + (\sim x + 1) = 0$

Flags

- A flag is a one-bit value: 1 is "set" and 0 is "unset"
- Flags record conditions of previous arithmetic operations
  - **C**: The carry-out flag from the last bit; indicates unsigned overflow
  - **Z**: Set if the result is zero
  - **N**: The sign bit of the result; indicates a negative signed result
  - **V**: Indicates if the result, interpreted as a signed value, is erroneous. For addition, this means that the signs of the operands agree and the result has a different sign

Arithmetic, Part 2

- Comparisons: <, <=, ==, !=, >=, >
  - Return "logical values, 0 or 1
  - Computation relies on subtraction and flags
  - Different for unsigned and signed
- Multiplication
  - Product can be two words long; it may be truncated to one word
  - Different for unsigned and signed
- Division
  - Produces quotient and remainder, one word each
  - Different for unsigned and signed
  - In x86, the (signed) remainder has the same sign as the numerator

Casting Types in C

- "Casting" means changing the type of a value

```
sometype x;
othertype y;

x = y;     // type error!

x = (sometype) y;
```

- Sometimes it means "interpret these bits in a different way"
  - Unsigned to/from signed
- Other times it means "convert these bits to the same value in a different representation"
  - Shorter integer types to/from longer
  - Integer types to/from floating point

Integer Types in C

- All integer types (char, short, int, long) can be prefixed with unsigned
- Constants are, by default, signed. Unsigned constants have the suffix U
- Casting between unsigned and signed changes the interpretation, but not the bits
- Implicit casting occurs in assignments and parameter lists. In mixed expressions, signed values are implicitly cast to unsigned
  - Source of many errors!

Casting Exercises

Word size = 32 bits

TMIN = -2,147,483,648
TMAX = 2,147,483,647

For each pair, decide whether <, =, or >

0	0U
-1	0
-1	0U
2147483647	-2147483647-1
2147483647U	-2147483647-1
-1	-2
(unsigned)-1	-2
2147483647	2147483648U
2147483647	(int)2147483648U

Integer C Puzzles

True or False?

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

```
1. x < 0   implies (x*2) < 0
2. 0 <= ux
3. x & 7 == 7  implies (x<<30) < 0
4. ux > -1
5. x > y   implies -x < -y
6. x * x >= 0
7. x >= 0   implies  -x <= 0
8. x <= 0   implies  -x >= 0
9. (x|-x)>>31 == -1
10.ux >> 3 == ux/8
11.x >> 3 == x/8
12.x & (x-1) != 0
```


Sign Extension

	5	-5
four bits	0101	1011
eight bits	00000101	11111011

- To convert a signed value to a larger number of bits, simply replicate the sign bit on the left.
  - Reason: $-b2^{w-1}$ + other stuff $= -b2^{w} + b2^{w-1}$ + other stuff

Multiplying with Shifts

C uses << and >>. The arithmetic/logical choice is made according the the operands being signed/unsigned.

Java has no unsigned integers, but it has a third shift >>> for logical right shift.

We can multiply (often faster than with the processor's multiply instruction) with shifts.

- $x \times 24 = x \times 32 - x \times 8$
  $= (x << 5) - (x << 3)$

Most compilers will generate this code automatically.

Signed Division by a Power of 2

- **x >> k** computes **x / 2^k**, rounded towards $-\infty$
- C on Intel processors rounds towards 0
  - `-11 >> 2 == -3`, but `-11/4 == -2`
- Solution: If x < 0, add 2^k-1 before shifting
  - Why does this work?

```
if (x < 0)
   x += (1 << k) - 1;
return x >> k;
```


When to Use Unsigned

- Rarely
- When doing multi-precision arithmetic, or when you need an extra bit of range … but be careful!

```
unsigned i;
for (i = cnt-2; i >= 0; i--)
    a[i] += a[i+1];
```

Pointer Arithmetic in C

- Pointers are, effectively, unsigned integers that signify addresses in memory
- Suppose `p0` and `p1` are pointers to type `T`, and `j` is an integer
  - `p0 – p1` is a signed value; it is the number of objects (not bytes!) between the two addresses
  - `p0 + p1` and `p0 * p1` are disallowed
  - `p0 + j` means `p0 + j * sizeof(T)`
  - `&x` is the address where `x` is stored
- Arrays are implemented as pointers.

Arrays

- Contiguous block of memory
- Pointer to start, then indexed by element size
  - Indices start at zero
- `ary[k]` is the same as `*(ary+k)`

Two-dimensional Arrays

- Same storage layout:

```
int a[48];      // 48 integers
int b[6][8];    // 6 rows, 8 columns
```

- "row major order"
- `b[i][j]` is the same as `b[8*i+j]`

Typedefs

- Abbreviation for complex types

```
int b[6][8];   // b is a two-dim array

typedef int b_type[6][8];
b_type b_var; // b_var is a two-dim array
```


Arrays and Pointers Combined

```
int *p[47];
```

- Array of pointers … or … pointer to an array??
- It's an array of `47` pointers
  - `p[3]` is the fourth pointer in the array `p`
  - `p[3]` is the base of an array
  - `p[3][6]` is the integer at position `6` in the array `p[3]`
    - Danger! `p[3][6]` looks the same as `a[3][6]`

What is printed?

```
int a[100];
int *p[47];

p[3] = a+12;
for (int i = 0; i < 100; i++)
    a[i] = i;

printf("%d\n", p[3][4]);
```

Structs

- Hetrogeneous records
- Example:

```
typedef struct cell {
  int value;
  struct cell *next;
} cell_t;
```

- Usage:

```
cell_t c;
c.value = 42;
c.next = NULL;
```

How many bytes are allocated for c? for p?

- Usage with pointers:

```
cell_t *p;
p->value = 42;
p->next = NULL;
```


Summary: Data in C

- Arrays and pointers
- Casts (explicit and implicit)
- Typedefs
- Structs