211: Computer Architecture
Fall 2019

Topic:

- C programming
- Data representation

Debugging C

```
#include <stdlib.h>

int main(int argc, char* argv[])
{
    int* p = malloc(10 * sizeof(int));
    for (int i = 0; i <= 10; i++)
        p[i] = i;
}
```

Debugging C

- printf(“Reached here\n”);
- -Wall
- -Werror
- -fsanitize=address
- -g
- valgrind
- gdb

Valgrind

- Run as “valgrind ./a.out” (where a.out is your binary)
- Finds invalid reads
- Finds invalid writes
- Finds leaks
- Any program you submit should have 0 of these!

Checking output

- Piping: take output from one program and give it as input to another – “./a.out | wc -l”
- Redirect output to a file: “./a.out > out”
- Compare two files: “diff myOutput correctOutput”
- See exact file contents: “hexdump -c myFile”

GDB – The GNU debugger

- Run as “gdb a.out”
- Then “run arg1 arg2 arg3”, where the args are command-line arguments for your program
- To stop before the program finishes, set a breakpoint: “break main”
- Once you hit a breakpoint, use “next” to step one line of code at a time
- Use “print x” to see the contents of a variable x
- Use “backtrace” to see the current list of functions called
- Use “frame X” to select frame X from this list

Classic Memory Bugs

Memory management is one of the biggest differences between C and Java

Here are some classic bugs that might afflict you

Bug - # 1

The classic `scanf` bug

```
scanf("%d", val);
```

Bug - # 2

Reading Uninitialized Memory

- Assuming that heap data is initialized to zero

```
/* return y = Ax */
int *matvec(int **A, int *x) {
  int *y = malloc(N*sizeof(int));
  int i, j;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      y[i] += A[i][j]*x[j];
  return y;
}
```

Bug - # 3

Overwriting Memory

- Allocating the (possibly) wrong sized object

```
int **p;
p = malloc(N*sizeof(int));
for (i=0; i<N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

Bug - # 4

Overwriting Memory

- Off-by-one error

```
int **p;
p = malloc(N*sizeof(int *));
for (i=0; i<=N; i++) {
  p[i] = malloc(M*sizeof(int));
}
```

Bug - # 5

Overwriting Memory

- Misunderstanding pointer arithmetic

```
int *search(int *p, int val) {
  while (*p && *p != val)
     p += sizeof(int);
  return p;
}
```

Bug - # 6

Referencing Nonexistent Variables

- Forgetting that local variables disappear when a function returns

```
int *foo () {
  int val;
  return &val;
}
```

Bug - # 7

Freeing Blocks Multiple Times

```
x = malloc(N*sizeof(int));
<manipulate x>
free(x);

y = malloc(M*sizeof(int));
<manipulate y>
free(x);
```

Bug - # 8

Referencing Freed Blocks

```
x = malloc(N*sizeof(int));
<manipulate x>
free(x);
...
y = malloc(M*sizeof(int));
for (i=0; i<M; i++)
  y[i] = x[i]++;
```

Bug - # 9

Failing to Free Blocks (Memory Leaks)

- Slow, long-term killer

```
foo() {
   int *x = malloc(N*sizeof(int));
   ...
   return;
}
```

Bug - # 10

Failing to Free Blocks (Memory Leaks)

- Freeing only part of a data structure

```
struct list { int val; struct list *next;};
foo() {
  struct list *head = malloc(sizeof(struct list));
  head->val = 0;
  head->next = NULL;
  <create and manipulate the rest of the list>
  ...
  free(head);
  return;
}
```

What Do Computers Do?

Manipulate stored information

Information is data

- How is it represented?

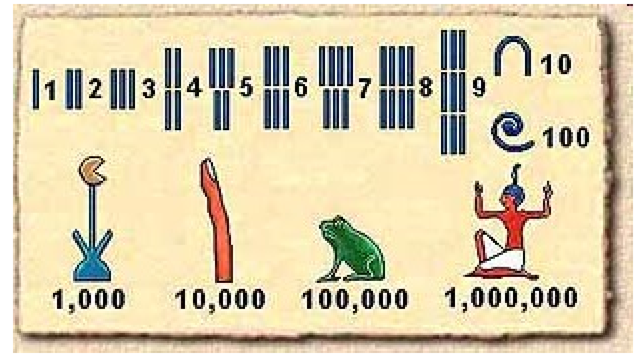
Basic information: numbers

Human beings have represented numbers throughout history

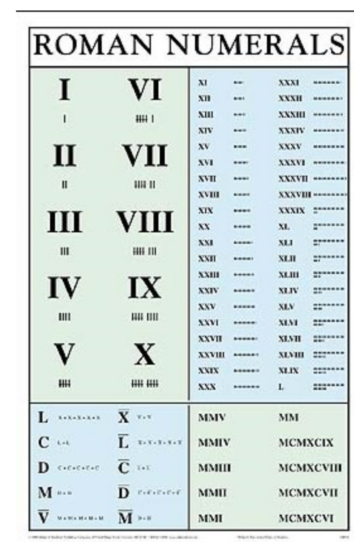
- Egyptian number system
- Roman numeral

Typically decimal

- Natural for humans



Discoveregypt.com



Number System

Comprises of

- Set of numbers or elements
- Operations on them
- Rules that define properties of operations

Need to assign value to numbers

Let us take decimal

- Base 10
- Numbers are written as $d_n \ldots d_2 d_1 d_0$
- Each digit is in [0-9]
- Value of a number is interpreted as $\sum_{i=0}^{n} d_i \times 10^i$

Binary Numbers

Base 2 ⇒ each digit is 0 or 1

Numbers are written as $d_n \ldots d_2 d_1 d_0$

Value of number is computed as $\sum_{i=0}^{n} d_i \times 2^i$

Binary representation is used in computers

- Easy to represent by switches (on/off)
- Manipulation by digital logic in hardware

Written form is long and inconvenient to deal with

Hexadecimal Numbers

Base 16

Each digit can be one of 16 different values

- Symbols = {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}

First 10 symbols (0 through 9) are same as decimal

- A=10,B=11,C=12, D=13, E=14, F=15

Numbers are written as $d_n \ldots d_2 d_1 d_0$

Value = $\sum_{i=0}^{n} d_i \times 16^i$

Octal Numbers

Base 8 ⇒ each digit is in [0-7]

Numbers are written as $d_n \ldots d_2 d_1 d_0$

Value of number is computed as $\sum_{i=0}^{n} d_i \times 8^i$

Converting Hex to Binary

Each hexadecimal digit can be represented by 4 binary digits

- Why?

0x2A8C (hex) = 0b0010101010001100 (binary)

- 0xC = 12 x 16^0 = 8 + 4 = (1 x 2^3) + (1 x 2^2) = 0b1100
- 0x80 = 8 x 16^1 = 2^3 x 2^4 = 2^7 = 0b 10000000
- And so on …

So, to convert hex to binary, just convert each digit and concatenate

What about octal to binary?

Converting Binary to Hex

Do the reverse

- Group each set of 4 digits and change to corresponding digit in hex
- Go from right to left

Example 1011011110011100

- 0b1011011110011100 = 0xB79C

What about binary to octal?

Decimal to Binary

What's the largest p, q, r … such that

- $n = 2^p + r_1$, where $r_1 < 2^p$
- $n - 2^p = 2^q + r_2$, where $r_2 < 2^q$
- $n – (2^p + 2^q) = 2^r + r_3$, where $r_3 < 2^r$
- …

The above means that

- $n = (1 \times 2^p) + (1 \times 2^q) + (1 \times 2^r) + \ldots + r_m$, where $r_m = n \% 2$
- Can you see why this now allows n to be easily written in binary form?

Example: convert 21 to binary

- $21 = 2^4 + 5$, $5 = 2^2 + 1 \Rightarrow 21 = 0b10101$

Decimal to Binary and Back

How to do the conversion algorithmically?

What about binary to decimal?

What about decimal to hex? Hex to decimal?

Decimal to octal? Octal to decimal?

Hex to octal? Octal to Hex?

Decimal and Binary fractions

In decimal, digits to the right of radix point have value $1/10^i$ for each digit in the i^{th} place

- 0.25 is 2/10 + 5/100

Similarly, in binary, digits to the right of radix point have value $1/2^i$ for each i^{th} place

- Just the base is different

8.625 is 1000.101

- .625 = 6/10 + 2/100 + 5/1000 = 1/2 + 1/8

How to convert?

Decimal to Binary Example

```
Algorithm

Number = decimalFraction
while (number > 0) {
    number = number*2
    if (number >=1) {
        Output 1;
        number = number-1
    }
    else {
        Output 0
    }
}
```

Why does it work?

Example: 0.625 to binary

- ANS: 0.101
  - 0.625*2 = 1.25
  - output 1
  - 0.25*2 =0.5
  - output 0
  - 0.5*2 = 1
  - output 1
  - Exit

C shift operations

- x << n – shift x left by n bits
- x >> n – shift x right by n bits
- A left shift = multiplying by 2
- A right shift = dividing by 2 (and discarding remainders)

- 0x01 << 1 == 0x02
- 25 << 2 == 100
- 25 >> 1 == 12

C mask operations

- x & m – do a bitwise AND operation
- x | m – do a bitwise OR operation
- x ^ m – do a bitwise XOR operation
- ~x – do a bitwise NOT operation

- 0x01 & 0x01 = 0x01
- 0x11 | 0x44 = 0x55
- 0x11 | 0x14 = 0x15
- 0x55 ^ 0x53 = 0x06

C shift and mask

- Given the integer 0x11223344, grab the 3rd byte (0x33)

- int x = 0x11223344;
- int y = (x >> 8) & 0xff;