# 1 Base Conversion

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

| Pow | $2^p$ | $16 * p$ |
|---|---|---|
| 0 | 1 | 0 |
| 1 | 2 | 16 |
| 2 | 4 | 32 |
| 3 | 8 | 48 |
| 4 | 16 | 64 |
| 5 | 32 | 80 |
| 6 | 64 | 96 |
| 7 | 128 | 112 |
| 8 | 256 | 128 |
| 9 | 512 | 144 |
| 10 | 1024 | 160 |
| 11 | 2048 | 176 |
| 12 | 4096 | 192 |
| 13 | 8192 | 208 |
| 14 | 16384 | 224 |
| 15 | 32768 | 240 |

- Numbers are theoretical and are representations of numerals
- Numerals have infinite preceding zeroes
- **Overflow**: When a set number of bits cannot represent the result of add (sub, mul, or div)
- $16^3 = 4096, 16^4 = 65536$
- $2^{-3} = 0.125, 2^{-4} = 0.0625, 2^{-5} = 0.03125$

# 2 Integer Representations

- **Unsigned**: $[0, 2^n - 1] = [0, 255]$
- **Sign and Magnitude**: $[-(2^{n-1} - 1), 2^{n-1} - 1]$
  - Leftmost sign is int (0 is +, 1 is -)
  - **Problem 1**: Making numbers smaller than 0 makes the number bigger in binary representation
  - **Problem 2**: There are 2 zeroes (sign bit does not matter)
  - **Problem 3**: Complicated circuitry
- **One's complement**: $[-(2^{n-1} - 1), 2^{n-1} - 1] = [-127, 127]$
  - Flip all the bits for negative numbers
  - **Problem**: there are still two 0s (0000 and 1111)
  - **Solution**: shift everything over left by 1
- **Two's complement**: $[-(2^{n-1}), 2^{n-1} - 1] = [-128, 127]$
  - Flip all bits and add 1
    - Or multiply leftmost bit by $-1$
  - All 1s represents -1
  - $2^{N-1}$ non-negatives, $2^{N-1}$ negatives
  - There are 1 more negative than positive
  - Cannot overflow when adding numbers of opposite sign
  - All neg numbers have bigger representation than pos
- **Bias encoding**: $[0 + \text{bias}, 2^n - 1 + \text{bias}]$
  - Add a bias to bring the center of the range down
  - Standard Bias for N bits: $-(2^{N-1} - 1)$

# 3 C Basics

- **C Pre-Processor** (CPP): Executes lines with `#`
  - Does a text-replace for macros
  - Functions would be called multiple times
- Compiling can be done in parallel ( `make -j` )
- **Amdahl's Law**: Linking is a sequential bottleneck
- C is function oriented
- **Execution**: Compile with `gcc file.c`
- **Memory storage**: Handled manually ( `malloc(), free()` )
- **Constants**: `#define, const`
- **Variable declaration**: Beginning of block (C99 same as Java)
- ANSI C was updated to C99 aka C9x
  - Added variable-length arrays, int types and booleans
- C11, C18: Multi-threading, unicode, removed `gets()`
- Passing args into main: `int main(int argc, char *argv[])`
  - `argc` contains the number of strings on command line
    - Executable counts as one, plus one for each argument
  - `argv` is a pointer to an arr with the args as strings

## 3.1 Types

- **Booleans**: `0, NULL, false` are False, everything else True
- **Integers**: Preferrable to use `intN_t` and `uintN_t`
- **Constants**: `const int days_in week`
- **Enums**: `enum color RED, GREEN, BLUE`
- **Typedef**: `typedef uint8_t BYTE;` Custom types
- **Struct**: Custom structured groups of variables

```
typedef struct{
    int length_in_seconds;
    int year_recorded;
} SONG;
```

# 4 Pointers

- Variables have undefined values if not initialized in declaration
- **Heisenbug**: Difficult to reproduce/randomly change
- **Bohrbug**: Repeatable bugs
- **Syntax**:
  - `int *p;` p is a pointer
  - `p = &y;` assign the address of y to p
  - `z = *p;` assign value at address in p to z
  - `*p = z;` assign the value of z to what p is pointing to
- Must initialize pointers before dereferencing them
- Generic pointer: `void*`
- Pointer to func: `int (*fn) (void*, void*) = &foo`
  - Call the func: `(*fn)(x, y)` or `foo(x,y)`
- **Struct arrow notation**: `int h = paddr->x;`
  - Equivalent to `int h = (*paddr).x;`
- **Word Alignment**: Can only access 4 byte boundaries
- **Handle**: pointer to a pointer `**h` (needed for arrays)

## 4.1 Pointers vs Arrays

- `char *string` and `char string[]` are almost identical
  - Cannot increment an array pointer
  - Array address = itself: `&arr` equiv to `arr`
  - String literals are read-only (cannot edit chars)
    - `char *str = "hello"`, cannot do `char[2] = 'y'`
    - Malloc first (include `\0`) to do `char[2] = 'y'`
  - Don't need explicit array size for initializers
    - `int a[] = 1, 2;` , `char s[] = "hey";`
    - `char s[]` is same as `char[] s`
  - For arrays, must allocate strlen+1 for null terminator
- Declared arrays are only allocated in local scope
- **Pointer arithmetic**: `pointer + n` adds `n * sizeof(type)`
  - Get ith element of arr: `arr[i]` , `*(arr + i)`
- Array does not know own length, use `int ARRAY_SIZE`
  - Exception is strings, the null terminator ( `\0` ) signifies end

# 5 Dynamic Memory Allocation

- `sizeof(type)` returns size in bytes ( `sizeof(char) = 1` )
- Mem Allocation: `malloc(size)` returns `(void*)`
  - Initialize with all 0: `calloc(num_items, sizeof(type))`
  - Resize: `realloc(ptr, size)` returns new pointer
  - Casting (not required): `(char*) malloc(sizeof(char))`
  - Returns `NULL` if request cannot be granted
  - For strings, `malloc(strlen(string) + 1))` b/c `\0`
- Dynamically allocated space must be `free(ptr)`
- **Memory leak**: If memory is not freed and the program exits
- Array names are not variables
  - `*a` : first value of array
  - `a` : pointer of array
  - `&a` : pointer of array (just like `a` )
- Variable decl. allocates memory, but struct decl. does not
- `strcpy(destination, original)`

```
struct Node {
    char *value;
    struct Node *next;
};
typedef struct Node *List;
// List is pointer to struct Node

List ListNew(void) { return NULL; }
//Create new empty list
```

## 5.1 Memory Locations Summary

- **Local variables**: stack
- **Function params**: stack
- **Result of malloc**: heap
- **Static variables**: static
- **Global variables**: static
- **Strings**:
  - static ( `char* s = "str"` )
  - stack ( `char[4] s = "str"` ) or ( `char s[4] = "str"` )
- **Constants**:
  - stack (function scope `const` )
  - static (global scope `const` )
  - code (hardcoded values)
- **Functions**: code
- **#define values**: code
- **Machine instructions**: code

## 5.2 Memory Location Details

- **Stack**: LIFO, grows downward
  - Stores local vars, params, instruction addresses, local `const`
  - Contiguous blocks of memory, stack pointer points to top
  - Stack frame tossed off when procedure ends (but not erased)
  - Stack pointer is just moved up
  - Big stack from deep recursion/fractals
- **Heap**: Space from `malloc` , grows upward
  - Result of `malloc()`
  - Not contiguous memory
  - Want fast `malloc()` and `free` but avoid fragmentation
  - Memory blocks have a header with size and pointer
  - Free blocks are kept in a circular linked list
    - Pointer field unused in allocated blocks
  - `malloc()` : search free list for a block large enough
    - **best-fit**: choose smallest block
    - **first-fit**: choose first block
    - **next-fit**: choose smallest, resuming from last search
  - `free()` : Check if blocks adjacent to freed block are free
    - Coalesce into 1 bigger free block, otherwise add new freed block to free list
  - `free` is very quick, `malloc` searches the full list worst case
  - Big heap from big mallocs without function calls
- **Static data**: Global scope `static` , Does not grow/shrink
  - Static variables (anywhere), global variables (incl. const)
- **Code**: Loaded when program starts. Does not grow/shrink
  - Includes functions, values of `#define` , machine instructions
- `char*` lives in static, `char[]` lives in stack
  - Array is made on the stack and filled with chars

## 5.3 Common Errors

- **Dangling reference** - pointer pointing to freed memory
  - Leads to probable **Segmentation Fault**
- **Memory leak** - memory not freed but cannot be accessed
  - Losing the ptr by incrementing it
  - Not freeing before returning
- **Seg Fault** - Read/writing to memory you don't have access to
- **Bus Error** - Bad word alignment

# 6 Bitwise Operators

- Operators: AND, OR, XOR, NOT: `&, |, ^, ~`
- OR turns bits on, AND turns bits off, XOR flips
- Use Two's Complement to leave all MSBs untouched
- **Logical Left shift**: `x << k` add `k` 0s on the right
  - Equivalent to $x * 2^k$
- **Logical Right shift**: `x >> k` remove `k` bits from the right
  - Equivalent to $\lfloor x / 2^k \rfloor$
- **Applications**:
  - Get $n^{\text{th}}$ bit of x: `(x >> n) & 1`
  - Set $n^{\text{th}}$ bit of &x to v: `*x = (*x & ~ (1 << n)) | (v << n)`
  - Flip $n^{\text{th}}$ bit of &x: `*x = *x ^ (1 << n)`
- **Postfix**: Highest priority. `*p++ = *(p++)`
  - Useful for checking `NULL` and incrementing
  - Dereferences THEN increments
- **Prefix**: Equal priority to dereference. `++*p = ++(*p)`
  - Has right to left associativity (like `*, &`
- Order of operations:
  - `while (*dst++ = *src++);`

1) Assign value at src to dst
2) Check if value at dst (assigned from src) is true (nonzero)
3) Increment dst, src