

# CS3210: Makefiles, C, and gdb

## Tutorial 2

Instructor: Dr. Tim Andersen

# Overview

- Goal: Understand C and GDB
- Part1: C Programming
- Part2: GDB
- Part3: In-class Exercises

# Part 1: C Programming Review

- Part 1: C Programming Review
- Bitwise Operations
- Pointers
- Review of the prep quiz

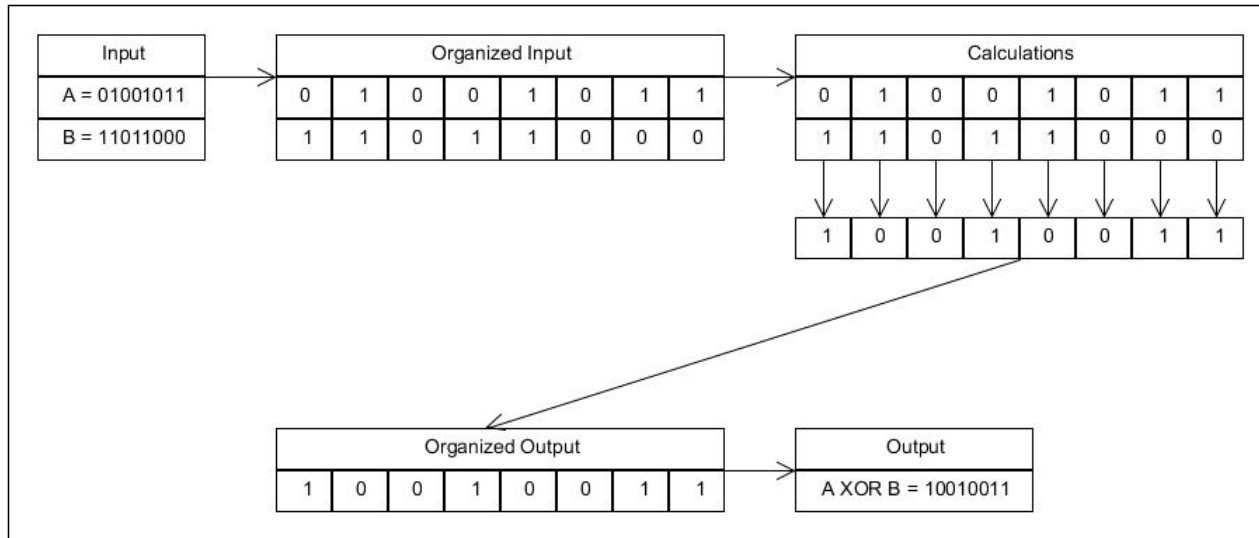
# Features of C

- Few keywords
- Structure, unions
- Macro preprocessor
- Pointers - memory, arrays
- External standard library - I/O, etc..
- Lacks (directly)
  - Exceptions, garbage-collection, OOP, polymorphism

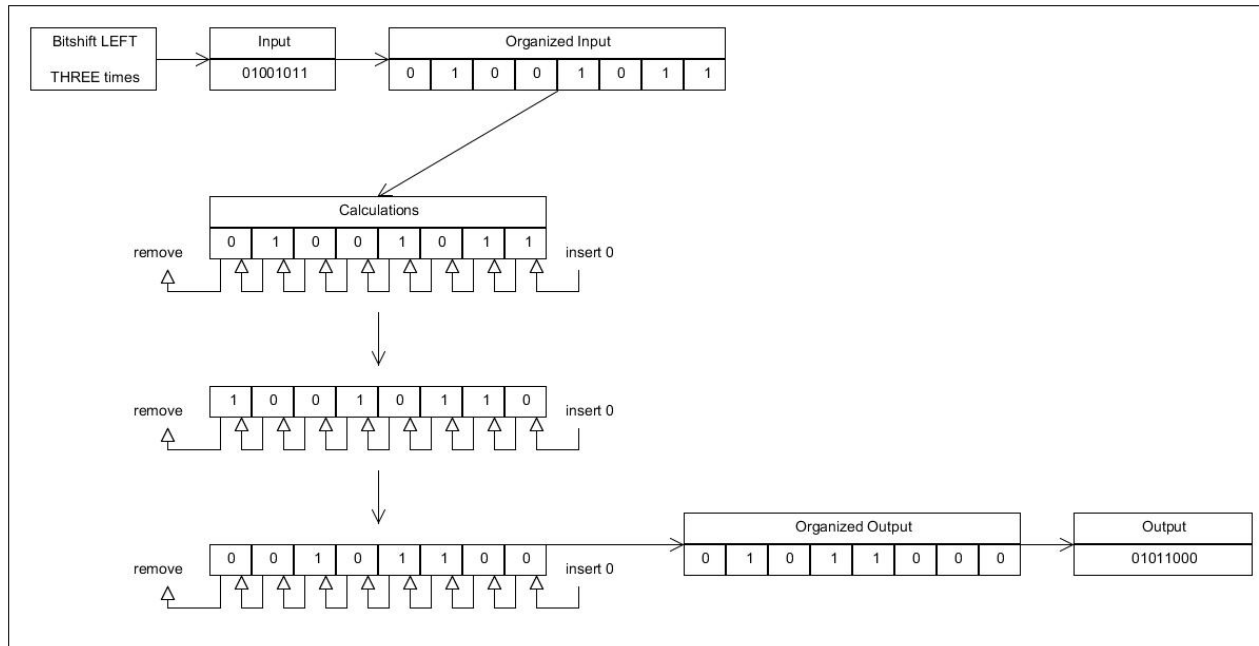
# Bitwise Operators in C

- `&` --- bitwise AND
- `|` --- bitwise inclusive OR
- `^` --- bitwise exclusive OR
- `<<` --- left shift
- `>>` --- right shift
- `~` --- one's complement(unary)

# Bitwise XOR



# Bitwise shift (left and right)



# Bitwise Operations - Example

```
unsigned getbits(unsigned x, int p, int n)
{
    return (x >> (p+1-n)) & ~(~0 << n);
}
```

- Let's say  $x=3210$ ,  $p=10$ ,  $n=4$

$p+1-n \rightarrow 10+1-4 = 7$

$1100\ 1000\ 1010 \gg 7 \rightarrow 0000\ 0001\ 1001$

$\sim(\sim 0 \ll 4) \rightarrow \sim(1111\ 1111\ 0000) \rightarrow 0000\ 0000\ 1111$

$0000\ 0001\ 1001 \& 0000\ 0000\ 1111 \rightarrow 0000\ 0000\ 1001 \rightarrow 9$



# Pointers

- Pointers are variables that contain **memory addresses** as their values
- A variable name directly references a value
- A pointer indirectly references a value
  - Referencing a value through a pointer is called indirection
- A pointer variable must be declared before it can be used

# Concept of address and pointers

- Memory can be conceptualized as a linear set of data locations
- Variables reference the contents of these locations
- Pointers have a value of the address of a given location

# How to read a declaration

## Definition Code

---

1. `p` is a variable `const int *p;`
2. `p` is a pointer variable `const int* p;`
3. `p` is a pointer variable to an integer `const int* p;`
4. `p` is a pointer variable to a constant integer `const int* p;`

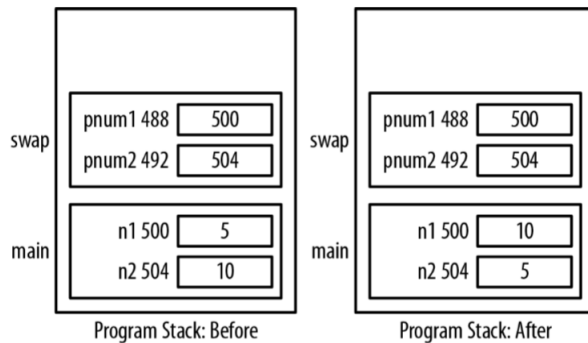
# Example (1)

```
int main(){  
    int n1 = 5;  
    int n2 = 10;  
    swap(&n1, &n2);  
    return 0;  
}
```

What should swap() look like?

# Example (1) - Answer and Result

```
int swap(int* pnum1, int* pnum2){  
    int tmp;  
    tmp = *pnum1;  
    *pnum1 = *pnum2;  
    *pnum2 = tmp;  
}
```



# Function pointer declaration

parameters

void (\*foo)();

Return type

Function pointer's variable name

```
// Example
```

```
int (*f1)(double); // passed a double, returns an int  
void (*f2)(char*); // passed a pointer to char and returns void
```

# Example (2)

```
int add(int num1, int num2){
    return num1 + num2;
}
int subtract(int num1, int num2){
    return num1 - num2;
}
int (*fptrOperation)(int,int);
int compute(fptrOperation op, int num1, int num2){
    return op(num1, num2);
}
//usage
printf("%d\n", compute(add,5,6));
printf("%d\n", compute(sub,5,6));
```

# Part 2: GDB

- Introduction of GDB
- How GDB works
- How GDB interact with QEMU



# Introduction to GDB

- GDB is the GNU program debugger
- GDB allows you
  - set a breakpoint in your program at any given point
  - examine the program state when stopped
  - change things in your program

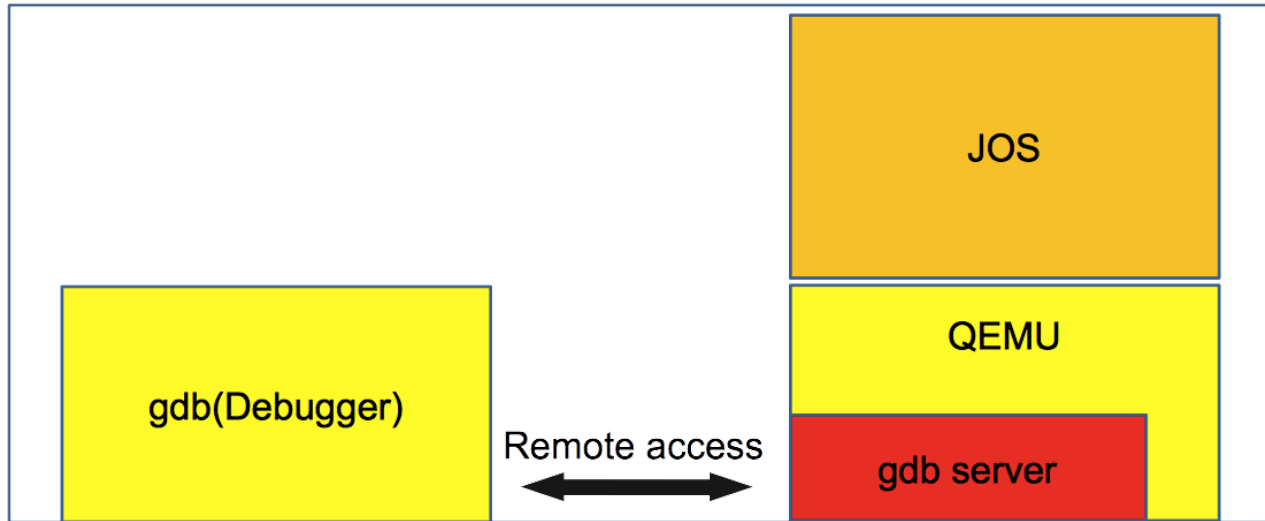
# GDB structure

- User interface
  - Several actual interfaces, plus supporting code
- Symbol side
  - Object file readers, debugging info interpreters, symbol table management, etc.
- Target side
  - Execution control, stack frame analysis, and physical target manipulation

# GDB debugger

- Kernel support
  - Debugger support has to be part of the OS kernel
  - Kernel able to read and write memory that belongs to each and every process
- Debugger-debuggee synchronization
  - Signal
- Hardware Breakpoint -Built-in debugging feature
- Software Breakpoint
  - Trap, illegal divide, some other instructions that cause an exception - INT3`

# GDB interaction with QEMU



# Example: make qemu-gdb

- Open cs3210-lab/lab/Makefile
  - .gdbinit
  - target remote localhost:26000

## Basic commands of GDB

- run / r / r arg1 arg2 arg3
  - Start program execution from the beginning of the program
- continue / c
  - Continue execution to next break point
- Kill
  - Stop program execution
- quit / q
  - Exit gdb

# GDB: break execution

```
break function-name/line-#/ClassName::functionName
break filename:function/filename:line-#
break *address
break line-# if condition
clear function/line-#
delete br-#
enable br-#
disable br-#
```

# GDB: line and instruction execution

- `step / s / si / s # / si #`
  - Step into
- `next / n / ni / n # / ni #`
  - Do not enter functions (step over)
- `Until / until line-#`
  - Continue processing until you reach a specified line number
- `Where`
  - Show current line number and which function you are in
- `Disassemble 0x[start] 0x[end]`
  - Displays machine code for positions in object code specified

# GDB: examine variables

- `x 0xaddress`
- `x / n 0xaddress`
- `print variable-name`
- `p/x` , `p/d` , `p/u` , `p/o`
  - Hex, signed integer, unsigned integer, octal
- `p/t variable` , `x/b address`
  - Binary, byte
- `p/a` , `x/w`
  - Hex address, 4 byte word of memory pointed by address



# Part3: In-class exercises

```
git clone git@github.gatech.edu:cs3210-fall2017/cs3210-pub
```

or

```
git pull in your cs3210-pub directory
```

```
cd cs3210-pub/tut/tut2
```

- Open README and follow all the steps. (There is an alternate README since the bootblock compiles differently for some people.)

