Tutorial Session 2
CS3210

Kyuhong Park

Overview

- Goal: Understand **C and GDB**

- Part 1: **C programming**
- Part 2: **GDB**
- Part 3: **In-class exercises**

Part 1 : C programming

- Part 1: **C programming**
    - Bitwise
    - Pointers
    - Review of the prep quiz

Features of C

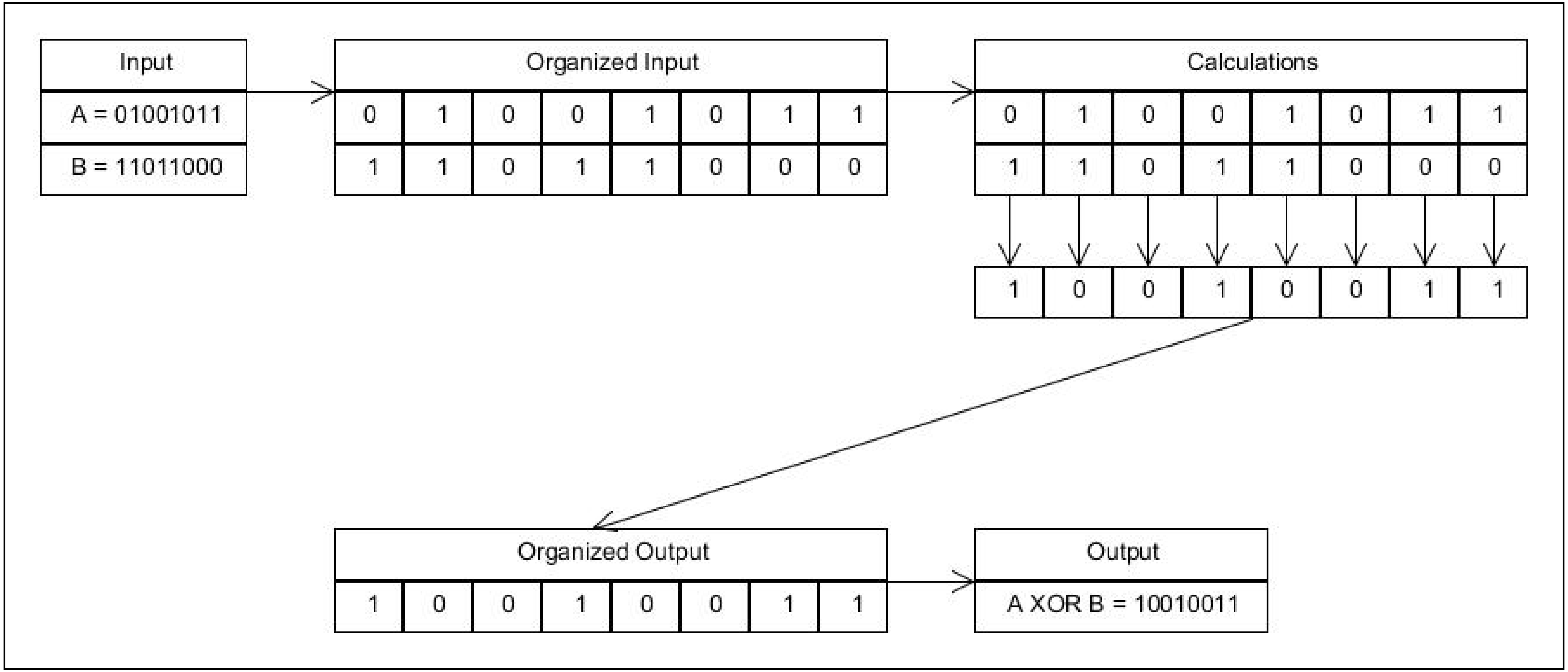
- Few keywords
- Structure, unions
- Macro preprocessor
- Pointers – memory, arrays
- External standard library – I/O, etc..

- But lacks
  - 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



Input
A = 01001011
B = 11011000

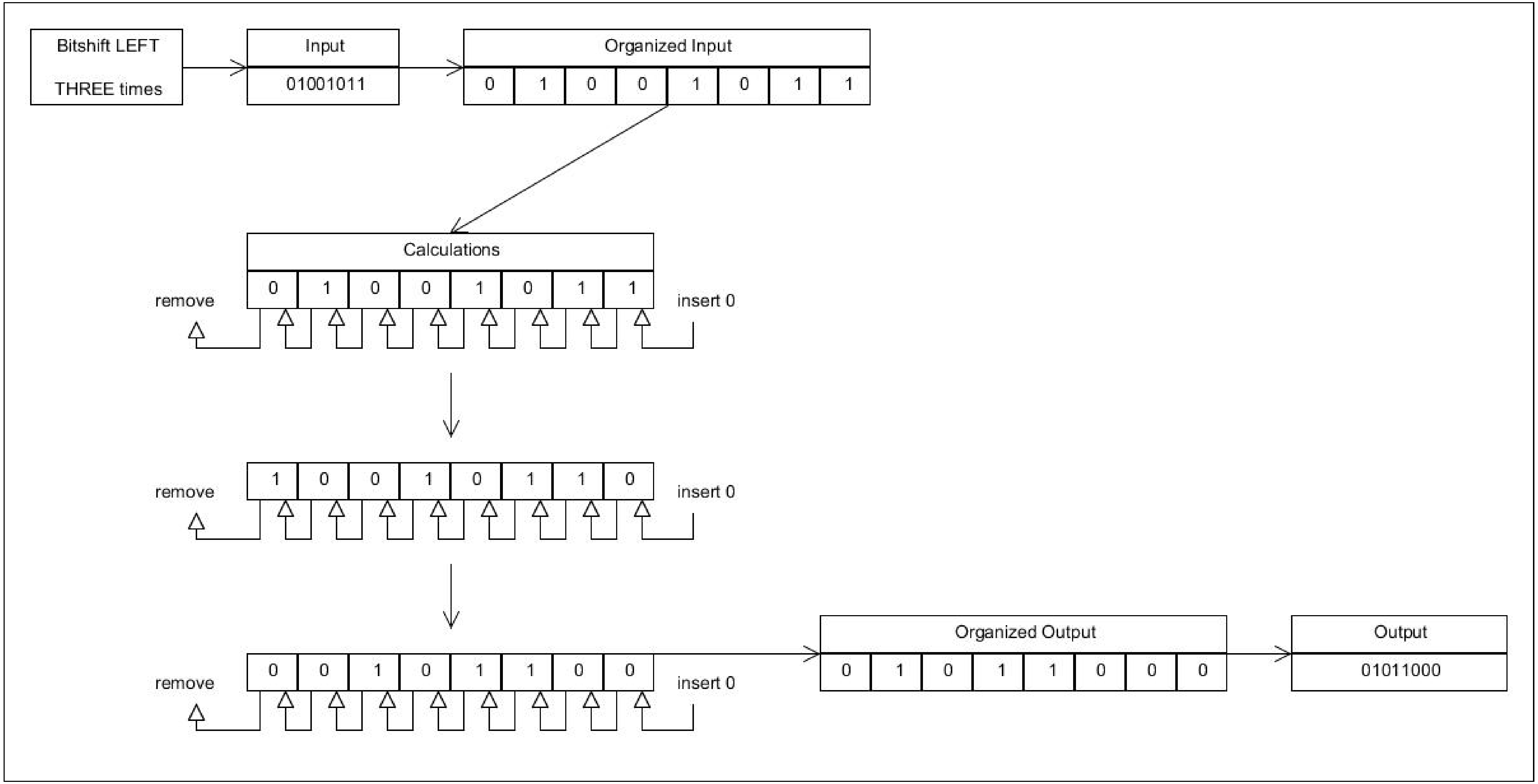
Organized Input							
0	1	0	0	1	0	1	1
1	1	0	1	1	0	0	0

Calculations							
0	1	0	0	1	0	1	1
1	1	0	1	1	0	0	0
1	0	0	1	0	0	1	1

Organized Output							
1	0	0	1	0	0	1	1

Output
A XOR B = 10010011

Bitwise shift (left and right)



Bitshift LEFT

THREE times

Input

01001011

Organized Input

0 1 0 0 1 0 1 1

Calculations

remove 0 1 0 0 1 0 1 1 insert 0

remove 1 0 0 1 0 1 1 0 insert 0

remove 0 0 1 0 1 1 0 0 insert 0

Organized Output

0 1 0 1 1 0 0 0

Output

01011000

Bitwise - 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 → 10+1-4 = 7
- 1100 1000 1010 >> 7 →   0000 0001 1001
- ~(~0 << 4) → ~(1111 1111 0000) → 0000 0000 1111
- 0000 0001 1001 & 0000 0000 1111 → 0000 0000 1001 → 9

Pointers

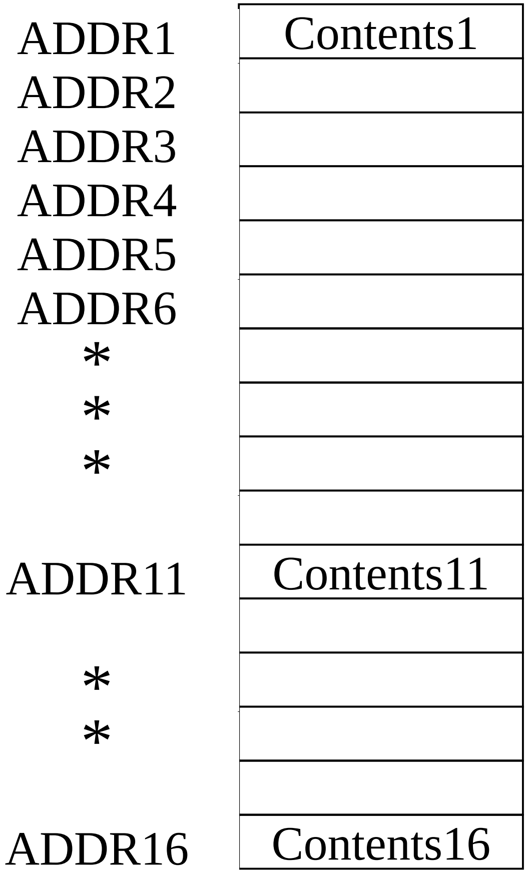
- Pointers are variables that contain **memory addresses** as their values
- A variable name **directly** references a value
- A pointer **indirectly** references a value
  - Referencing a value through a pointer is called **indirection**
- A pointer variable must be declared before it can be used

Concept of address and pointers

- Memory can be conceptualized as a linear set of data locations

- Variables reference the contents of a locations

- Pointers have a value of the address of a given location



ADDR1	Contents1
ADDR2	
ADDR3	
ADDR4	
ADDR5	
ADDR6	
*	
*	
*	
ADDR11	Contents11
*	
*	
ADDR16	Contents16

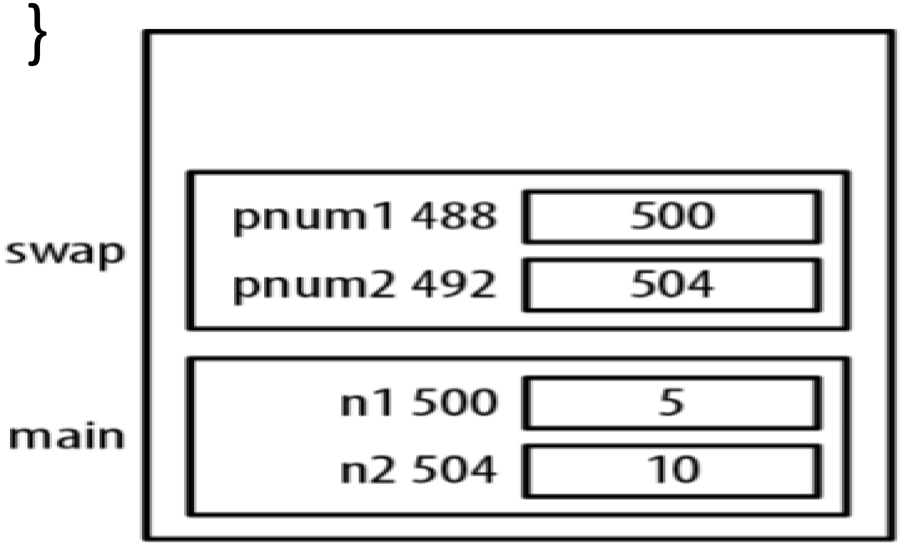
How to read a declaration

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;**

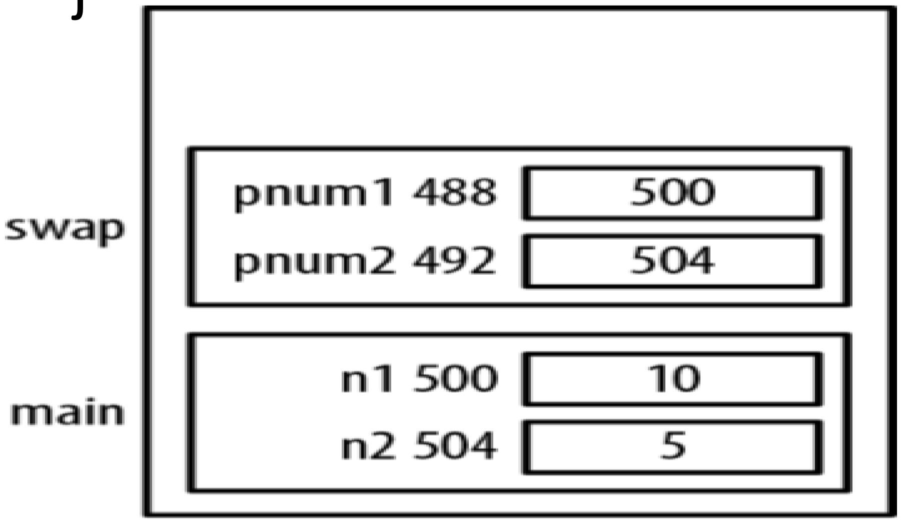
Example (1)

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

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

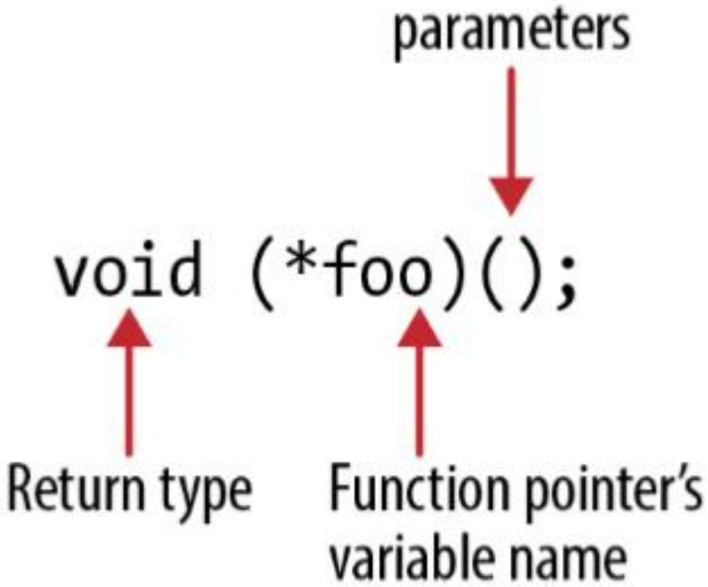


Program Stack: Before



Program Stack: After

Function pointer declaration



```
void (*foo)();
```

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);
}
```

```
printf(“%d\n”, compute(add,5,6);
printf(“%d\n”, compute(sub,5,6);
```

Review of the prep quiz

Q1 and Q2

```
uint32_t v = 0xdeadbeef;
printf (“0x%02x”,((unsigned char*)&v)[0]);
```

- Unsigned char* is 8 bits.
- Little endian
- &v represents address of v

V[0]	V[1]	v[2]	V[3]
ef	be	ad	de

Q3

printf (“%d, abs(-2147483648));

- INT_MIN in <limits.h> is a macro ➔ #define INT_MIN (-2147483647 -1)
- -2147483648 is a constant expression(- unary + integer constant)
- 2147483648 cannot be represented in a signed 32-bit integer.
- Compute - 2147483648 as (-2147483647 -1)
- movl $-2147483648 , %eax

  negl %eax ➔ 0-(-2147483648) ➔ 0x80000000 in hex

  ➔ 1000 0000 0000 0000 0000 0000 0000 0000.

  ➔ -2147483648

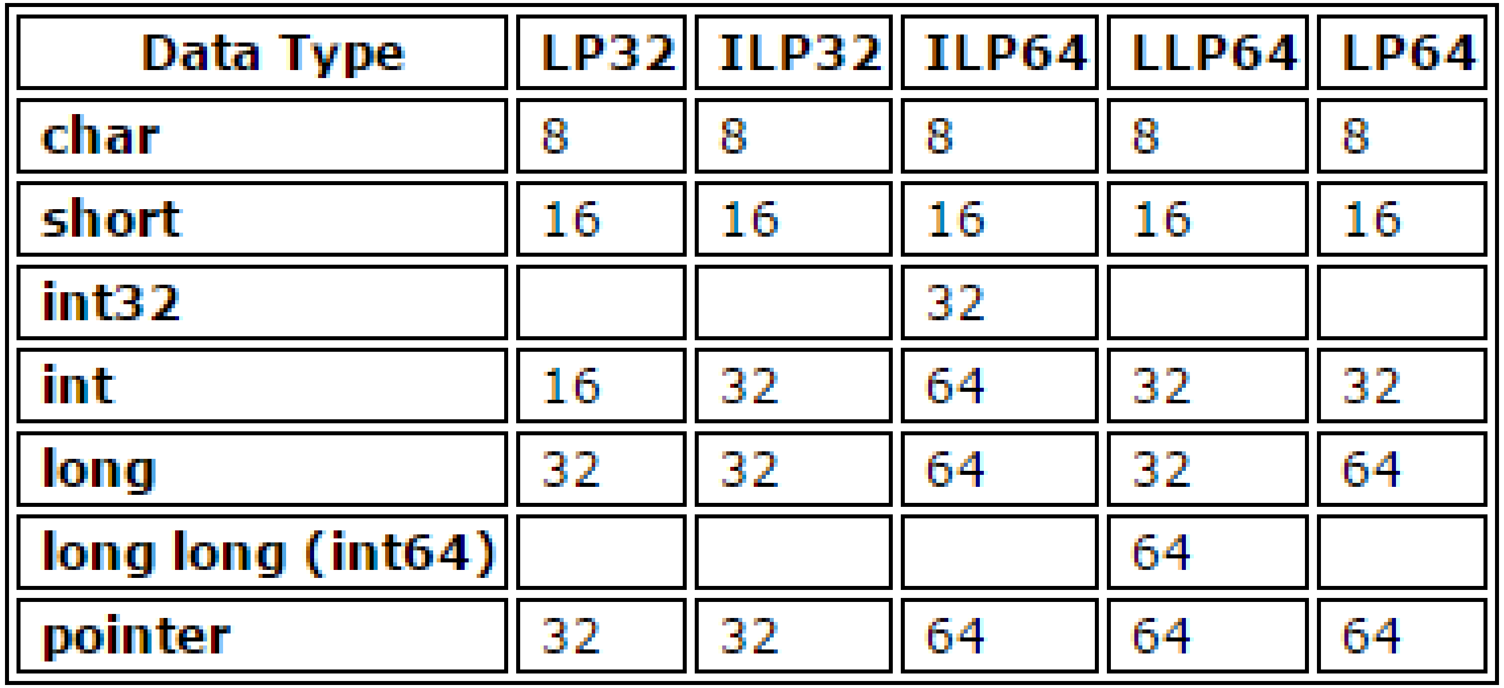
Q4

What does the expression, 1>0 evaluate to (on 64bit)

- True?
- False?
- On 32 bit?

Q5

```
printf("char=%d, int=%d, long=%d", sizeof(char), sizeof(int), sizeof(long));
```



Data Type	LP32	ILP32	ILP64	LLP64	LP64
char	8	8	8	8	8
short	16	16	16	16	16
int32			32		
int	16	32	64	32	32
long	32	32	64	32	64
long long (int64)				64	
pointer	32	32	64	64	64

Q6

```
unsigned int i = 0;
printf("%u", i--);
```

Q7

```
int main ()
  {
        int i, j, *p, *q;
        p = &i;
        q = &j;
        *p = 5;
        *q = *p + i;
        printf("i = %d, j = %d\n", i, j);
        return 0;
  }
```

	addr	**value**
i	100	5
j	104	10
p	108	100
q	112	104

Q8

What’s the value of NULL?

- 0x0000000?
- 0xffffffff?

Q9

```
main() {
    int x[5];
    printf("1 = %p\n", x);
    printf("2 = %p\n", x+1);
    printf("3 = %p\n", &x);
    printf("4 = %p\n", &x+1);
    return 0;
  }
```

(assuming the first printf results in the follow string)

"1 = 0x7fffdfbf7f00"

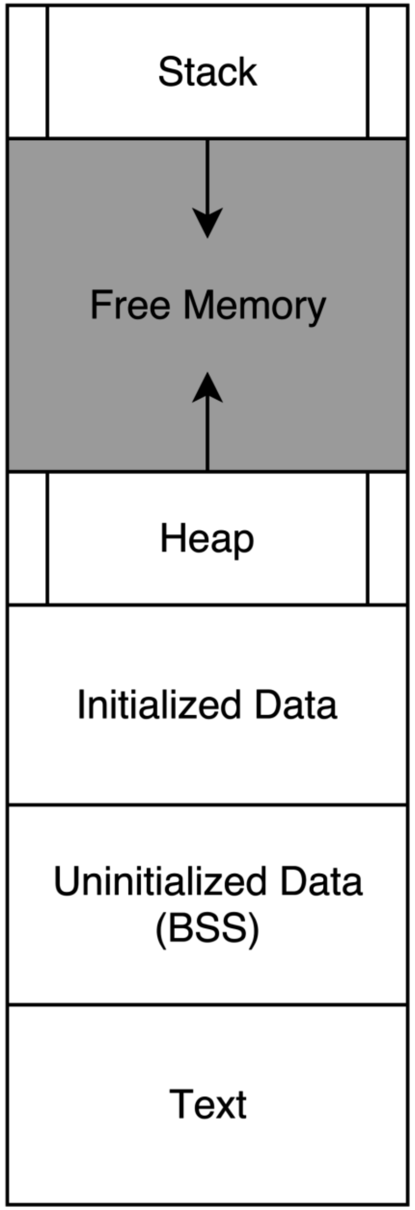
	addr	**value**
x[0]	f00	
x[1]	f04	
x[2]	f08	
x[3]	f0c	
x[4]	f10	
	f14	
	f18	
	f1c	

Q10

# Where does the string, "hello world", locate?

```
main() {
  const char *str = "hello world";
  printf("%s\n", str);
}
```

- .text?
- .data?
- .bss?
- Stack or heap?

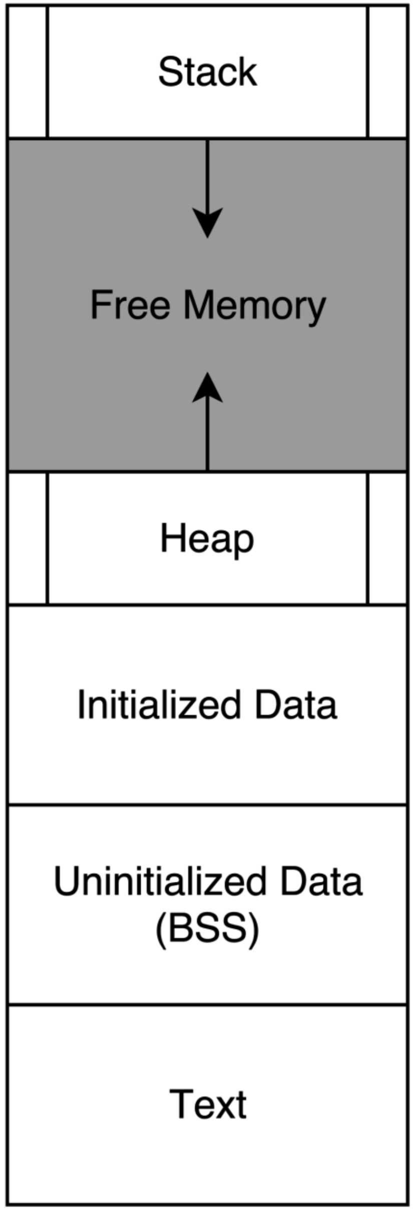


Q11

# Where does the string, “str", locate?

```
main() {
  const char *str = "hello world";
  printf("%s\n", str);
}
```

- .text?
- .data?
- .bss?
- Stack or heap?

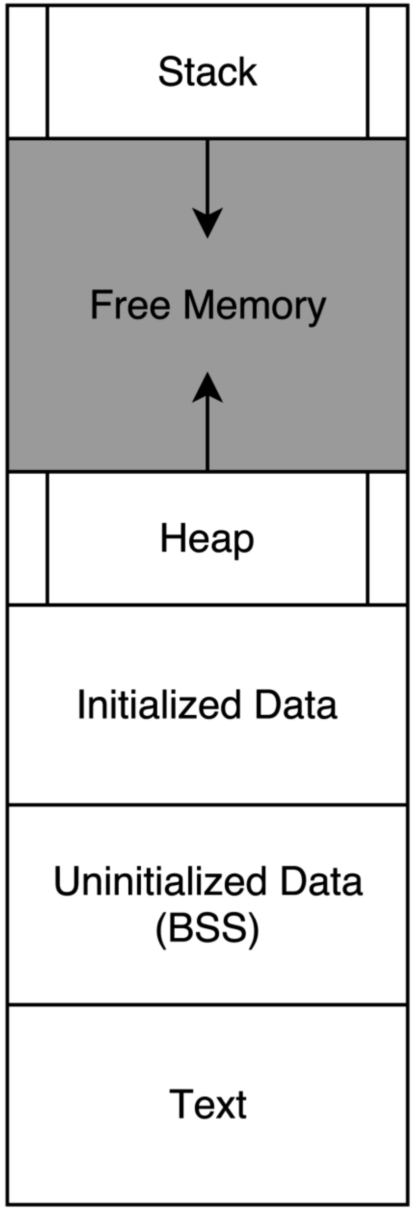


Q12

# Where does the string, “main", locate?

```
main() {
  const char *str = "hello world";
  printf("%s\n", str);
}
```

- .text?
- .data?
- .bss?
- Stack or heap?



Q13

# where does the arga locate relative to func's ebp (32-bit)?

func(arga, argb, argc, argo);

- ebp – 4?
- ebp + 0?
- ebp + 4?
- ebp + 8?

Stack Growth

Higher Addresses

…	
…	…
Local variable 3	EBP- c
Local variable 2	EBP – 8
Local variable 1	EBP – 4
Saved ebp	EBP
Return address	EBP + 4
arga	EBP + 8
argb	EBP + c
argc	EBP + 10
arg0	EBP + 14

Q14

```
main() {
    char array[] = {1, 2, 3, 4, 5};
    int i = 4;
    printf("%d", array[i++]);
  }
```

Q15

```
#define PTXSHIFT 12
#define PTX(va)  (((uint32_t)(va) >> PTXSHIFT) & 0x3FF)
  printf("0x%x", PTX(0x12345678))
```

- 0x12345678 → 0001 0010 0011 0100 0101 0110 0111 1000
- After >> 12 → 0001 0010 0011 0100 0101
- 0001 0010 0011 0100 0101
  & 0000 0000 0011 1111 1111 → 0011 0100 0101 → 0x345

Q16

```
#define PGSIZE          4096
#define CONVERT(sz)     (((sz)+PGSIZE-1) & ~(PGSIZE-1))
        printf("0x%x", CONVERT(0x123456));
```

- 0x123456 → 1,193,046 + 4095 == 0001 0010 0100 0100 0101 0101
- ~(4095) → 0000 0000 0000
- 0001 0010 0100 0100 0101 0101

  & 1111 1111 1111 0000 0000 0000 → 0x124000

Q17

```
#define ASSERT(a, b) do {switch (0) case 0:  case (a):;} while (0)
```

```
ASSERT(1==2,”error: should be equal”);
```

Q18

# what does the expression, -1U > 0, evaluate to (x86)?

- True?
- False?
- Undefined?
- Depending on architecture?

- 1U is unsigned int. → -1U is 1111 1111 1111 1111 > 0

Q19

# what does the expression, -1L > 1U on x86-64 and x86?

0 on both platforms?

1 on both platforms?

0 on x86-64, 1 on x86?

1 on x86-64, 0 on x86?

Undefined.?

- On x86-64, sizeof(int) == 4, sizeof(long) == 8
  - -1L > 1U second operand is promoted to long int. So False
- On x86 , sizeof(int) == 4, sizeof(long) == 4
  - -1L>1U first operand is promote to unsigned int. So true.

Part 2 : GDB

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

Introduction of 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.

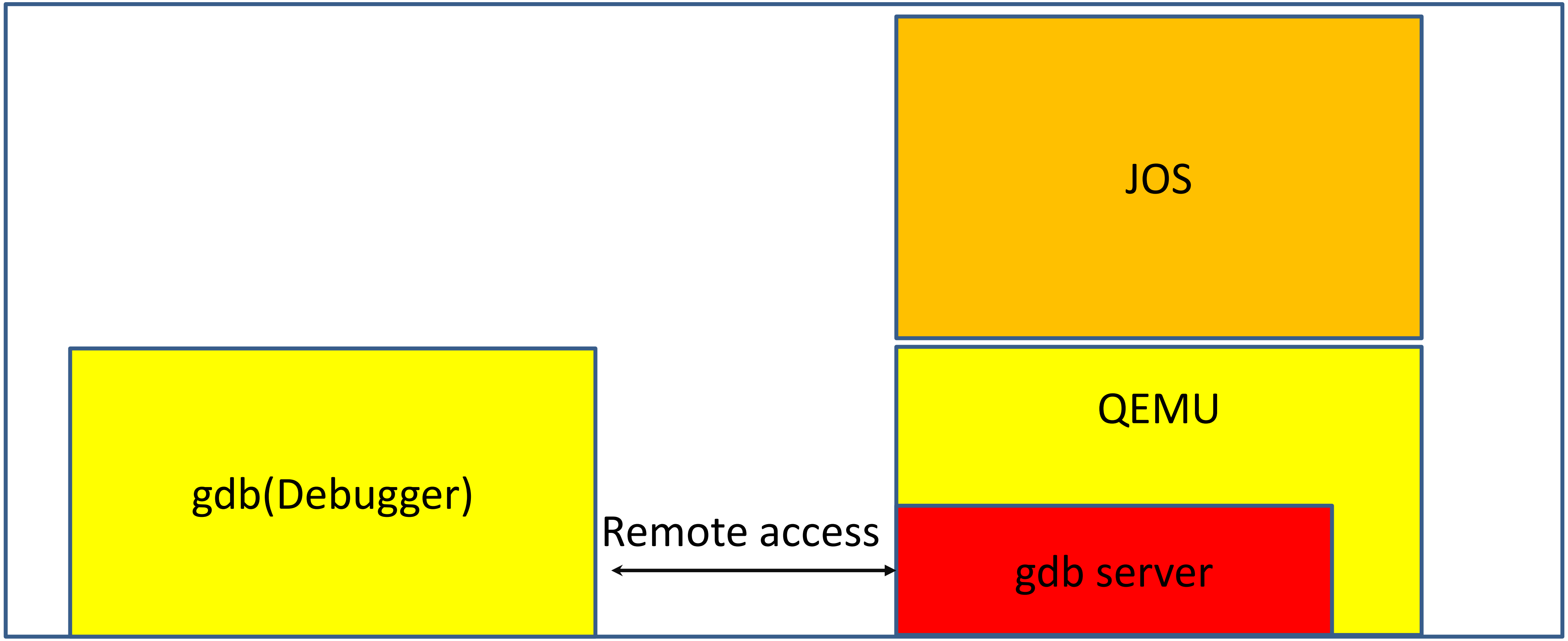
GDB structure

- 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's 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 #
  - Not enter functions
- 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/nfu 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
- p/a , x/w
  - Hex address, 4 bytes of memory pointed by address

Part3: In-class exercises

1) git clone git://tc.gtisc.gatech.edu/cs3210-pub
   or
   git pull in your cs3210-pub directory
2) cd cs3210-pub/tut/tut2
3) Open README and follow all the steps
4) Have a fun ☺