Basics of C Programming

CS 0449: Introduction to System Software

CS0449 Teaching Assistants

https://pitt.edu/~shk148/

University of Pittsburgh | School of Computing and Information

Meta-Notes

- These slides were adapted heavily from recitation slides created by *Martha Dixon* who was a teaching assistant (TA) for this course in Fall of 2020. They contain materials which were obtained from various sources, including, but not limited to, the following:

[1] J. Misurda, CS 0449: Introduction to Systems Software, 3rd ed. Pittsburgh, PA: University of Pittsburgh, 2017.

[2] S. J. Matthews, T. Newhall, and K. C. Webb, Dive into Systems: A Gentle Introduction to Computer Systems. San Francisco, CA: No Starch Press, 2022.

[3] R. Bryant, D. R. O'Hallaron, and M. S., Computer Systems: A Programmer's Perspective. Princeton, NJ: Pearson, 2016.

[4] L. Oliveira, V. Petrucci, and J. Misurda, in Introduction to Systems Software, 2022

Agenda

- Course News!
- Review of Bitwise Operations
- C Programming: Basic I/O
  - Using `scanf()` and `printf()`
- Lab 1 - Data Lab

Course News

- TA office hours
  - See https://cs0449.gitlab.io/fa2023/general
- Lab 1 announced
  - Due: 17:59 Thursday, September 14th, 2023.
- TopHat

Bitwise Operations

With materials from Jarrett Billingsley

Bitwise AND ("Logical product")

- AND takes two bits and gives you one new one.
- it can be written a number of ways:
  - A&B A ∧ B A · B AB
- if we use the and instruction (or & in C/Java):

	1	1	1	1	0	0	0	0
&	0	0	1	1	1	0	1	0
=	0	0	1	1	0	0	0	0

we did several **independent** OR operations.

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise OR ("Logical sum")

- we might say "and/or" in English
- it can be written a number of ways:
  - A|B A∨B A+B
- if we use the or instruction (or | in C/Java):

	1	1	1	1	0	0	0	0
|	0	0	1	1	1	0	1	0
=	1	1	1	1	1	0	1	0

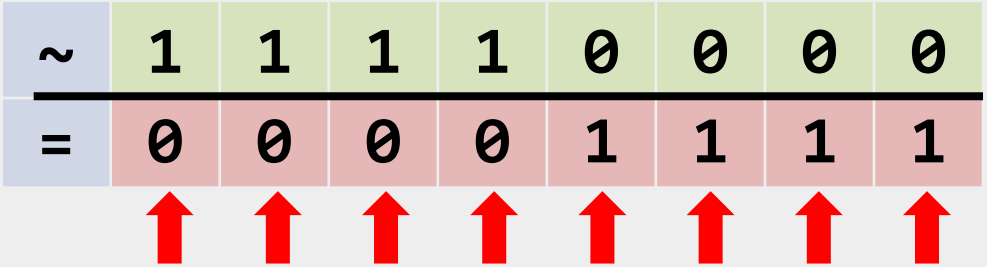
we did several **independent** OR operations.

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise NOT

- the ~ instruction

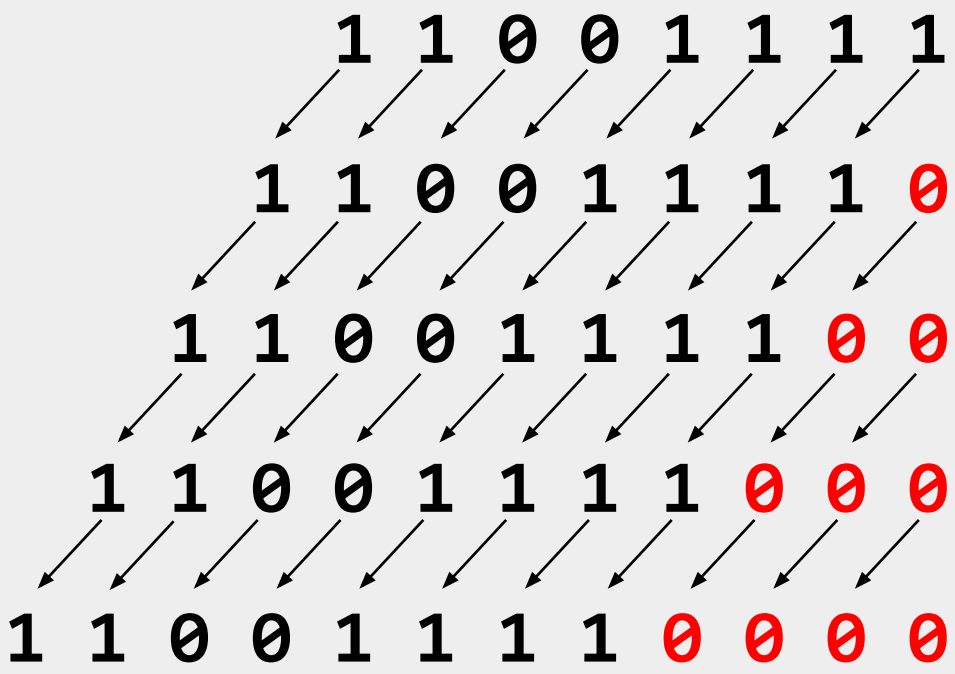
A	Y
1	0
0	1



we did several **independent** NOT operations.

Bit shifting

- besides AND, OR, and NOT, we can move bits around, too.



if we shift these bits **left by 1…**

we stick a **0** at the bottom

again!

AGAIN!

AGAIN!!!!

Left-shifting in C

- C/Java/Python/etc. use the << operator for left shift
  - B = A << 4; // B = A shifted left 4 bits
- but wait, If the bottom 4 bits of the result are now 0s…
  - …what happened to the top 4 bits?

bits that get "shifted off" the top are **discarded.** this *may or may not* lead to problems!

So… what does it DO?

- let's start with a value like 5 and shift left and see what happens

Binary	Decimal
00000101	5
00001010	10
00010100	20
00101000	40
01010000	80

a << n == a × 2n

- **shifting left by *n* is the same as multiplying by 2^n**
  - you probably learned this as "moving the decimal point"
  - and moving the decimal point *right* is like shifting the digits *left*
- with bit shifting, we're moving the **binary point** (yes, really)
- **shifting is fast and easy on most CPUs**
  - way faster than multiplication in any case
  - HLL compilers will try *really* hard to replace "multiplication by a constant" with shifts and adds

<_< >_>

- we can **shift right, too**

```
0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 1 1 0 1 1 1 0 0 1 1 1 1
0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 1 1 0 1 1 1 0 0 1 1 1
0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 1 1 0 1 1 1 0 0 1 1
0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 1 1 0 1 1 1 0 0 1
0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 1 1 0 1 1 1 0 0
```

a >>> n == a ÷ 2n

- **shifting right by *n* is the same as dividing by 2^n**

Binary	Decimal
01010000	80
00101000	40
00010100	20
00001010	10
00000101	5
00000010	2

that's what integer division gives us too, right?

5 / 2 == 2

but soon we'll see that right-shifting and division can sometimes **disagree.**

Signed numbers messing things up again

	Unsigned	Signed
1 0 1 0 1 1 0 0	= 172	= -84
0 1 0 1 0 1 1 0	= 86	= 86
0 0 1 0 1 0 1 1	= 43	= 43

well that's a little unfortunate.

Arithmetic Right Shift is used for signed numbers: it "smears" the sign bit into the top bits.

1 0 1 0 1 1 0 0	= -84
1 1 0 1 0 1 1 0	= -42
1 1 1 0 1 0 1 1	= -21

C uses >> (depends on data type)

Uh oh, they're fighting

n	Binary	Decimal	$a \div 2^n$
0	10110000	-80	-80
1	11011000	-40	-40
2	11101100	-20	-20
3	11110110	-10	-10
4	11111011	-5	-5
5	11111101	-3	-2
6	11111110	-2	-1
7	11111111	-1	0

well that's a little weird.

actually, this is *correct.*
but **so is the way that integer division works.**
they're *both* right.

(we'll come back to this.)

Doing modulo with bitwise AND

- in decimal, dividing by powers of 10 is trivial.

53884 ÷ 1000 = 53 R 884

- in binary, we can divide by powers of 2 easily with shifting...
- and we can **get the modulo by powers of 2 with bitwise AND!**

10010110 ÷ 1000 = 10010 R 110

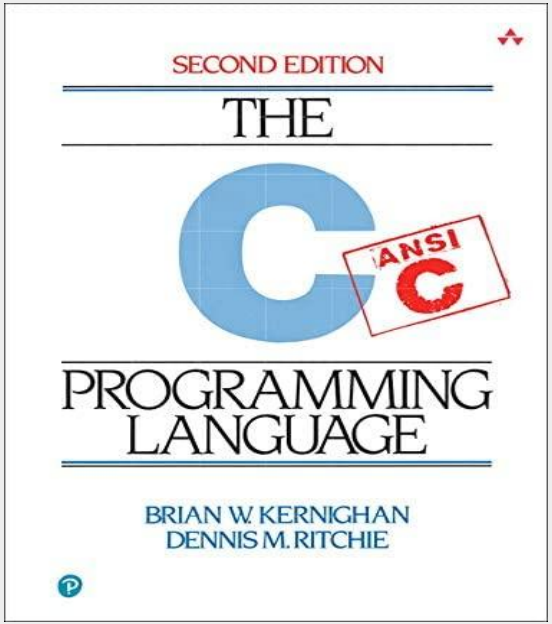
```
  10010110      10010110
>>>      3   & 00000111
  ---------   ---------
  00010010      00000110
```

so, $a \% 2^n == a \& (2^n-1)$

Bitwise != Logical

- `!` is a boolean operator, so it changes the logic value of the expression.
  - E.g., `!1 == 0` (b/c `!true == false`)
  - In C, booleans are just `int`s
    - `false == 0`
    - `true != 0`
      - » Caveat: C only guarantees that true is a non-zero integer.
      - » Practically, many systems/libraries define `true` to be `1`
  - `!42== 0`
- `~` is a bitwise operator, it affects the values of individual bits:
- E.g. (with 8 bits)
  - `~0 → -1 (00000000 → 11111111)`
  - `~5 → -6 (00000101 → 11111010)`

C Programming



Basic Input/Output using the C Standard Library

Standard C Library (libc)

- In Lab0, you used `printf()` in the Hello World program
  - `> printf("Hello world! x is currently %d \n", x);`
  - `> Hello world! x is currently 2`
  - `printf()` stringified the arguments and printed to the standard output
    - formatted the string and filled-in the placeholders (e.g., `%d`)
- Notice we didn't need to implement that `printf()` function ourself
  - `printf()` is a function built-in to C's standard I/O library
    - Hence, we needed to tell our compiler to make use of the standard library functions with `#include <stdio.h>`
    - You will talk about how the libraries are linked to your code in lecture soon

man gives us information about functions, commands, libraries

- On most Unix/Unix-like systems, you can use `man` to learn more about functions/commands/etc.
- The manual has the most accurate information about all the library functions, programs, commands, etc.

```
> Man printf
      SCANF(3)          Library Functions Manual          SCANF(3)
NAME
    printf, fprintf, ... , vsnprintf - formatted output conversion

                                    ...
DESCRIPTION
    The functions in the printf() family produce output according to a
    format as described below
```

If you are having trouble running man on Thoth, google `man printf`

Detailed look at using `printf()`

```
int printf(const char * format string, ...);
```

Returns an integer: number of characters printed (excluding null terminator)

Remember, in C, a string is just an array of characters

We place placeholders which begin with a percent sign (%). The variables which comes after the formatter will replace the placeholders when printing

```c
#include <stdio.h>
int main()
{
    printf("Name: %s, Info:\n", "John");
    printf("\tAge: %d \t Ht: %g\n", 20, 5.9);
    printf("\tYear: %d \t Dorm: %s\n", 3, "Towers");
    return 0;
}
```

```
Name: John, Info:
    Age: 20 Ht: 5.9
    Year: 3 Dorm: Towers
```

Reading Input using scanf()

- Like `printf()`, `scanf()` is another C standard library function
  - Used to read character, string, numeric data from keyboard
  - Again, if you want to use it in your program you have to include the header (`#include <stdio.h>`)

```
int scanf(const char * format, ...)
```

Returns an integer: number of input items successfully matched and assigned

Defines what we are reading (character? Integer? Float?)

Passes by reference (a pointer) to the variable which will hold our input

Example code using `scanf()` (live demo)

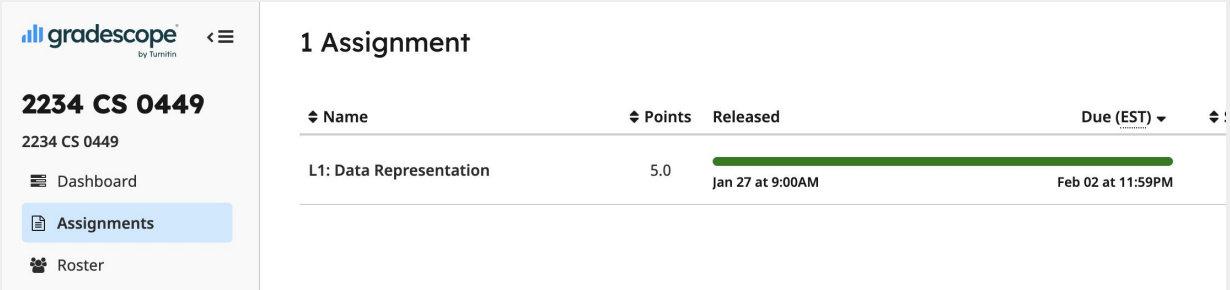
```c
#include <stdio.h>
int main()
{
    char ch;
    int x;
    printf("Enter any character \n");
    scanf("%c", &ch);
    printf("Entered character is %c \n", ch);
    printf("Enter any integer\n");
    scanf("%d", &x);
    printf("Entered integer is %d\n", x);
    return 0;
}
```

Lab 1: Data and Pointers

Practicing with data and input

Part A: Practicing Data and Bitwise Manipulation

- Collaboration: You are encouraged to work with one other person.
  - Select your partner's name on Gradescope
  - Part A – Problems
  - See L1: Data Representation on Gradescope
  - Multiple choice, fill-in-the-blank type of questions



B1: Understanding `sizeof()`

TODO: Write a program that prints the size of three structs

- REQUIREMENTS: create two files (`size.c`, `size.h`)
  - `size.c` has your implementation
  - `size.h` contains the structs
- Expected o

```
The size
The size
The size of struct C is #
```

- See Lab Handout for how to create struct A, B, and C

Hint: The special '`sizeof()`' macro gives us the byte size.
Also, remember to include the header file(s) in your `.c` file

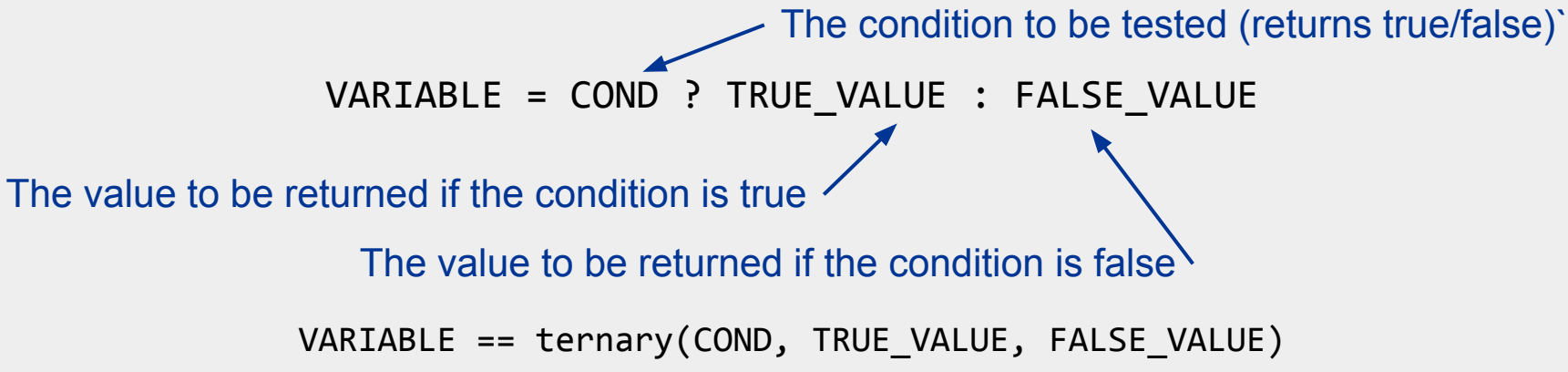
```
#include “header.h”  (for your header files)
#include <header.h>  (for standard libraries)
```

B2: Understanding ternary operators

TODO: Implement the function ternary in `ternary.c`

`int ternary(int cond, int true_value, int false_value) { /*...*/}`

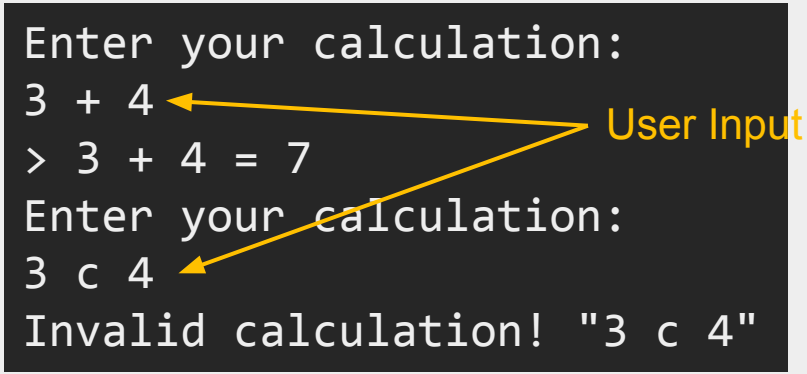
- Cannot use the ternary operator
- Output of ternary should be the same as: `cond?true_value:false_value`




```
VARIABLE == ternary(COND, TRUE_VALUE, FALSE_VALUE)
```

In C, `FALSE == 0`; `TRUE != 0` (usually TRUE == 1 but not always)

B3: Creating a simple calculator



HINT: Take a look at `calculator.c` from Lab0

Requirements

- Create 1 files: calculator.c
  - Inputs must be read from keyboard (use scanf())
- Support the following operations:
  - +, -, *, /, %(mod)
  - & (bitwise and), ~ (bitwise not)
- Your output must match the sample output