

# CSE320 System Fundamentals II

---

TONY MIONE

# Lecture overview

---

The C Language

Building C Programs on Linux

Debugging C Programs with gdb

Comments

Constants and Variables

Expressions

C Operators and Operator Precedence

Assignment Statement

C Selection Statements

C Loops

# C Programming Language

---

Compiled language (vs <sup>Python, JS.</sup> Interpreted)

Fairly static but efficient

Non-Object-Oriented

# C Programming

---

## A 'Systems' Programming language

- Suited for low-level operations (close to CPU)
- Useful for implementing Operating System code, drivers, etc.

## High-level language

- Variety of data types
- Functions
- Arrays, Structures (Records)
- Decision and Looping constructs
- Rich Run-time library

# Trivial C Program Example

---

**firstprog.c:**

```
#include <stdio.h>  
  
int main(int argc, char **argv)  
{  
    int a, b;  
  
    printf("Hello. Please enter an integer: ");  
    scanf("%d", &a);  
    b = a * a;  
    printf("%d is the square of %d\n", b, a);  
}
```

# Trivial C Program Example

---

**firstprog.c:**

```
#include <stdio.h>
```

```
int main(int argc, char **argv)
```

```
{
```

```
    int a, b;
```

```
    printf("Hello. Please enter an integer: ");
```

```
    scanf("%d", &a);
```

```
    b = a * a;
```

```
    printf("%d is the square of %d\n", b, a);
```

```
}
```

C programs always start in the function *main()*

# Trivial C Program Example

---

**firstprog.c:**

```
#include <stdio.h>
```

```
int main(int argc, char **argv)
{
    int a, b;

    printf("Hello. Please enter an integer: ");
    scanf("%d", &a);
    b = a * a;
    printf("%d is the square of %d\n", b, a);
}
```

*main()* takes two arguments  
(actually, can be 3 arguments):

1. An integer count of command line arguments
2. An array of pointers to the argument strings.

# Trivial C Program Example

---

**firstprog.c:**

```
#include <stdio.h>
```

```
int main(int argc, char **argv)
```

```
{
```

```
    int a, b;
```

```
    printf("Hello. Please enter an integer: ");
```

```
    scanf("%d", &a);
```

```
    b = a * a;
```

```
    printf("%d is the square of %d\n", b, a);
```

```
}
```

*main()* returns an integer (zero/non-zero for success, failure-code)



# Trivial C Program Example

---

**firstprog.c:**

```
#include <stdio.h>
```

```
int main(int argc, char **argv)
{
    int a, b;

    printf("Hello. Please enter an integer: ");
    scanf("%d", &a);
    b = a * a;
    printf("%d is the square of %d\n", b, a);
}
```

*stdio.h* contains definitions to allow access to library functions like *printf()* [print formatted] and *scanf()* [scan formatted data].

# Template main()

---

**Use the below template as a starting point for most simple C applications.**

```
#include <stdio.h>

int main(int argc, char **argv)
{

}

}
```

# Building C programs on linux

---

gcc firstprog.c

# compiles and links firstprog.c, writes 'a.out'

gcc -o firstprog firstprog.c # Same but writes executable to firstprog

↳ give a name to replace "a.out"

# To execute a program, type its path (directory/name) at the prompt

./firstprog

gcc name.c

./a.out

# Compiling C for Debugging

---

Required option:

- -ggdb – Add debug information to help gdb

Recommended:

- -O# - Set 'optimization level of code, #=0-3 (use 0 for debug)  
*More optimized ⇒ difficult to debug*

Example (multiple module build)

```
gcc -c -O0 -ggdb -o bigprocmain.o bigprocmain.c
gcc -c -O0 -ggdb -o bigprocutils.o bigprocutils.c
gcc -ggdb -o bigprog bigprocmain.o bigprocutils.o
```

# Using gdb

---

Command format:

```
gdb <executableFileName>
```

Programs may need command line arguments...use 'set args' from within gdb:

```
(gdb) set args <arg1> <arg2> ...
```

# Basic gdb commands

---

## Running program/breakpoints

- **b** <addr/symbol/line#> - Set a breakpoint at <addr/symbol/line#> (an address, line #, or function name)
- **clear** <addr/symbol/line#> - Clear breakpoint (must match argument used to set breakpoint)
- **r** – Run program from beginning
- **c** – Continue running program from last breakpoint
- **n** – Next (step over calls)
- **stepi** – Step (instruction) – Also, steps into a function

# Basic gdb commands

---

## Printing data and registers

- **info registers** – This dumps the cpu registers
- **p <addr>** – Print value in memory location <addr>
- **x/<format> <addr>** - eXamine value in <addr> as <format> (char, short, instruction, etc.)

## Stack status/contents

- **bt** – Backtrace – This command shows the list of stackframes for invoked functions.
- **up** – Go to previous (caller's) stack frame. Commands from this point will look at symbols from that stack frame's point of view.
- **down** – Go to the stack frame of the function called by the current stack frame (this has no effect if the stack frame is of the function containing the breakpoint.)

# Closer look at 'x' (examine)

---

- `x/<format> <addr>` - eXamine value in <addr> as a <format> (char, short, instruction, etc.)

## Format letters:

- o – octal
- x – hex
- d – decimal
- u – unsigned decimal
- f – float
- i – instruction
- c – char
- s – string

## Size format letters:

- b – byte
- h – halfword
- w – word
- g – giant (8 bytes)



# C Comments

---

Comments are notes to developers

Compiler never sees comments (preprocessor removes them)

Styles:

- Old (K&R) - `/* Comment goes here */`
- Newer (C99) - `// Comment goes here`

Older style can span lines:

```
/*  
 * This is a multi-line.  
 * comment  
 */
```

# C Types

---

## Basic types

- 'int' – Integers (related: char, short, long)
  - Values are signed by default
- 'float' – floating point (related: double)
- 'char' – A single character (8 bits)
- 'enum' – User defined collections

## Compound types

- array – Series of values of same type
- struct – A record or collection of fields of varying types

# C Constants

---

## Variables with value that cannot be changed

- Numeric:
  - const int fcOffset = 32;
  - const double myPi = 3.1415926535;
- Character:
  - const char defaultTempScale = 'F';
- String:
  - const char greeting[] = "Hello!\n";

## Literals [placed in C code without a name]:

- a = 5;
- scale = 'c';
- strcpy(buffer, "This is a literal string");

# C Variables

---

**Must** be 'declared' before use

**Must** have a type

**May** have an initial value in declaration

Examples:

- `int d;` // d is an integer (probably 32 bits)
- `unsigned short s;` // s is an integer, always positive
- `float pi = 3.14;` // pi is a float initialized to 3.14
- `double doubval;` // doubval is 64-bit float

Variable declarations must:

- Occur at top of block (function, loop body, etc.)
- Precede all executable code in the block


# Example: C Declarations

---

```
int main(int argc, char **argv)
{
    int a, b, c;
    float myfloat = 5.7e-02;
    char option = 'n';
    int odds[5] = {1, 3, 5, 7, 9};

    a = odds[0] + 2;
}
```

'Initializers' – These set the variable(s) to a value before the program or function starts executing.



# Expressions

---

Expressions can include:

- variables, constants
- function calls
- arithmetic operations
- Boolean/relational operations

Expressions have a resulting value

Expression values can be assigned to a variable or used immediately  
'inline'

# Arithmetic Expressions

---

Large number of arithmetic operators

Operators can be:

- unary (take 1 'operand')
- binary (take 2 'operands')

Operators have:

- Precedence – Order of evaluation \* / | + - 곱셈 나눗셈
- Associativity – How operator instances 'group' (left or right)
- These properties can be overridden with parenthesis

# C Operators (partial list)

---

++, -- (pre/post) increment/decrement

+, - (unary),

+, -, \*, /, % Add, subtract, multiply, divide, mod

! – logical NOT

~ - bitwise NOT     $\sim 1101 = 0010$

==, !=, <=, <, >=, > - Relational tests ops

&, |, ^ Bitwise logical (and, or, not)

&&, || Logical Operators (and, or)

=, +=, -=, \*=, /=, &=, |=, ^= Assignment



# C Operator precedence

	precedence	associativity
1	() [] -- (post) ++ (post)	left-to-right
2	--(pre) ++(pre) unary - unary + ! ~	right-to-left
3	* / %	left-to-right
4	+, -	left-to-right
5	<< >>	left-to-right
6	< <= > >=	left-to-right
7	!= ==	left-to-right
8	&	left-to-right
9	^	left-to-right
10		left-to-right
11	&&	left-to-right
12		left-to-right
13	assignments (=, +=, -=, *=, /=, etc)	right-to-left

# Example expressions

---

Assume  $a=10$ ,  $b=5$ , and  $c=2$

What are the values assigned to each variable below?

- $d = b + c * 10;$
- $e = 2 * a / 10;$
- $f = c + b * a;$

# Exercise

---

## **exercise1.c**

```
#include <stdio.h>
```

```
int main(int argc, char **argv)
```

```
{
```

```
    int a=10;
```

```
    int b=5;
```

```
    int c=2;
```

```
    // Copy expressions from previous slide here...
```

```
}
```

Now, compile exercise1.c with debug support. Run gdb and step through the program. Examine variable values after each statement.

Add one more statement: `e = e + f;`

Recompile and rerun. Examine the value of 'e' after the second assignment.

# C statements

---

Assignment

Selection

Iteration (loops)

# C Assignment Statements

---

## Assignment

- Simple (=)
- Compound (+=, -=, \*=, etc)

# Compound assignment

---

Combines arithmetic or bitwise operators with assignment

## Examples

- `a += 5;`     $\rightarrow$     `a = a + 5;`
- `b |= 0x7F;`     $\rightarrow$     `b = b | 0x7F;`
- `e = 17;`
- Note : Right side is fully evaluated first (as though surrounded by parens!)
- `e *= 5 + 3;`     $\rightarrow$     `e = 17 * (5 + 3); => 136`

# C Selection statements

---

## Selection

- if/then/else
- ?:
- switch()

# if Statement

---

Syntax:

```
if (<relational test>
    { statements }
[else
    { statements }]
```

Example:

```
if ((a > b) && (c < 10)) {
    printf("This is not good!\n");
} else {
    printf("Ok, keep going\n");
}
```



# ?:

---

Syntax:

(<relational test>) ? expression : expression;

Example:

```
printf((a>b) && (c<10)) ?  
    "This is not good!\n" :  
    "Ok, keep going\n" );
```

# Exercise

---

Write an if statement to test if a value (a) is positive, negative, or zero. If it is positive, replace the value with twice the value. If negative, replace it with three times the value. If it is zero, assign the value 1 to it.

```
int a;
```

```
// Code that reads a value for a from the terminal
```

```
printf("Type an integer: ");
```

```
scanf("%d", &a);
```

```
// If statement to test value:
```

```
// Code to print new value:
```

```
printf("New value: %d\n", a);
```

# switch

---

Syntax [Note, the ‘{ }’ around the case’s code are optional]:

```
switch (<var>) {  
    case <val1>: {  
        statements;  
        [break;]  
    }  
    case <val2>: {  
        statements;  
        [break;]  
    } ...  
    default:  
}
```

# switch Example

---

Example:

```
switch (a) {  
    case 1:  
        printf ("1\n");  
        break;  
    case 3:  
        printf ("3\n");  
        break;  
    default:  
        printf ("Unknown\n");  
}
```

# C Loops

---

## Iteration (loops)

- while()/do...while()
- for()

# while

---

## Syntax:

```
while (<relational test>) {  
    statements  
}
```

## Example:

```
a = 10;  
while (a < 10) {  
    printf("a = %d\n", a);  
    ++a;  
}
```

# do...while()

---

## Syntax:

```
do {  
    statements  
} while(<relational test>);
```

***!!! Note: This tests at the end of the loop so 'statements' are guaranteed to execute at least 1 time!***

## Example:

```
a = 10;  
do {  
    printf("a = %d\n", a);  
    ++a;  
} while (a < 10);
```

# for

---

## Syntax:

```
for (<init-code>; <relational>; <end-of-loop-code>) {  
    statements  
}
```

## Example:

```
for (a = 0; a < 10; ++a) {  
    printf("a = %d\n", a);  
}
```



# for and while

---

A for statement can always be converted to an equivalent while loop structure:

```
for (a = 0; a < 10; ++a) {  
    printf("a = %d\n", a);  
}
```



```
int a = 0;  
while (a < 10) {  
    printf("a = %d\n", a);  
    ++a;  
}
```

# Exercise

---

Write a loop to sum the integers from 1 through 20 placing the value in an integer variable named *sum*. Feel free to use any of the C loop statements.

# Formatted Printing

---

`printf()` – Allows printing text and variable values with diverse formatting directives

Sends formatted text to `stdout` (terminal screen, by default)

Must include `stdio.h`

Takes a variable number of arguments:

- 'Format' string – includes text and '%' format specifiers
- Need 1 additional argument for each format specifier in the format string

# Format specifiers

---

Start with ‘%’ and end with a letter to indicate type

Between percent and type, length and format options can refine the output

Template: %[-,0]#.#<formatletter>

- Leading symbols like ‘-’ or ‘0’ indicate options like left justify or 0-fill
- First # is usually a field length
- Second # is usually a precision length (i.e. for fractional parts)

To print a literal ‘%’, use %%

# Format Specifiers

---

Specifier	Description
<code>%[0]#d</code>	Print integer in a field '#' bytes long [0-fill if lead 0 present]
<code>%.#pf</code>	Print float in a field '#' bytes long with 'p' digits of fraction
<code>%[0]#x</code>	Print hexadecimal value in a field '#' bytes long [0-fill if lead 0 present, use Uppercase A-F if specifier X is in Uppercase]
<code>%c</code>	Print a character
<code>%[-]#s</code>	Print a string in a field '#' bytes long. [Left justify if '-' present]

# printf() example

---

## **The code:**

```
int count = 4;
int sum = 19;
float average = (float)sum / count;
int memaddr = 0x40bc;

printf("The sum of the values is %d\n", sum);
printf("Average: %6.2f\n", average);
printf("Address: 0x%08X\n", memaddr);
```

## **Should print:**

```
The sum of values is 19
Average:    4.75
Address: 0x000040BC
```

---

# Questions?