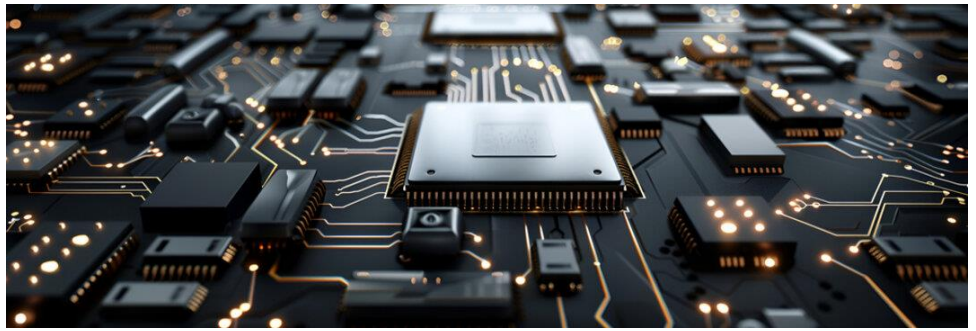
EECS 2032: Introduction to Embedded Systems



Course Instructor

Sunila Akbar

Lecture 4

Recap

- Command Substitution
- Arrays
- The `read` and `test` Commands
- The `if` Command
- Test Conditions (contd.)
  - Example – `if` Command

- The `case` Command
- Looping Commands
  - `for, while, until, shift, break, continue`
- Functions

Topics to be Covered

- BASH
  - here Document and here Strings
  - Test Subtleties
  - sed Command
  - String Manipulations
- Programming with C
  - Introduction to C
  - Software Development Cycle
  - Testing and Types of Errors
  - Getting started
  - Reading/Writing (I/O)
  - Variables and Data Types
  - Number Systems in C
  - Basic Arithmetic Operations
  - Pre and Post Operators
  - Boolean Expressions
  - while loop for Lab 4

here Document

- The here document is a special form of quoting
- It accepts inline text for a program expecting input until a user-defined terminator is reached
- The command receiving the input is appended with a << symbol, followed by a user-defined word or symbol, and a newline

```
$ n=`bc << EOF
```

EOF: A marker that denotes the end of the input

- The next lines of text will be the lines of input to be sent to the command

```
> scale=3
> 13/2
```

- The input is terminated when the user-defined word or symbol is then placed on a line by itself in the leftmost column (it cannot have spaces surrounding it)

```
> EOF`
$ echo $n
```

Outputs 6.500

Ref - Prev. Example

here Document

```
$ n=`bc << EOF
> scale=3
> 13/2
> EOF`
$ echo $n
```

Example 1

```
$  cat <<FINISH
>  This is line 1 of the input.
>  This is line 2 of the input.
>  This is line 3 of the input.
>  FINISH
This is line 1 of the input.
This is line 2 of the input.
This is line 3 of the input.
```

Example 2

- The word is used in place of `Ctrl-D` to stop the program from reading input
- The user-defined terminating word or symbol must match exactly from "here" to "here"
- It is much more practical to use them in scripts making the scripts easier to read and maintain by breaking them up into multiple lines

here Document – Example of a Script

```
#!/bin/bash
# Create a new file called "output.txt"
cat > output.txt <<EOF
This is line 1 of the output.
This is line 2 of the output.
This is line 3 of the output. EOF
```

here String

- A here string is a feature in Unix/Linux shell scripting that allows you to pass a single string as input to a command, without using a here document or an external file.
- It's more concise than a here document and is useful when you want to provide a small amount of data as input to a command.
- here strings are specified using <<<

```
$ command <<< "your input string"
```

```
$ bc <<< "scale=2; 10 / 3"
```

Outputs 3.33

```
$ read -a array <<< "1 2 3 4"
```

Test Subtleties

- Refers to the less obvious behaviors and conditions that can affect the outcome of tests or evaluations, particularly in conditional statements
- Often arise from the way shells interpret variables, return values, types, or special cases, leading to unexpected results

```
$ [ “$var" = "rightvalue" ] && echo OK
$ [  = rightvalue ] && echo OK
```

No Syntax Error if var is not previously defined, evaluates to false and no echo!

```
$ [ “X$var" = “Xrightvalue" ] && echo OK
```

✓

```
$ [ -d $dir ] || mkdir $dir
```

the command might fail or behave unexpectedly without quotes around the variable as the $dir may contain spaces or metacharacters

```
$ [ -d "$dir" ] || mkdir "$dir"
```

✓

Test Subtleties

```
1. read marks
2. if [ $marks -ge 80 ]; then
3.             grade=A
4. elif [ $marks -ge 70 ]; then
5.             grade=B
6. elif [$marks -ge 60]; then
7.             grade=C
8. else
9.             grade=D
10.fi
11.echo $grade
```

```
# what if user enter marks < 70 (e.g. 69)

# grade = C ?????? testing?

# Note that there is no compilation here, only interpretation

# Test for all test cases, not only for marks > 80 or marks > 70!
```

`sed` - Introduction

- `sed` is a streamlined editor – text processing tool

```
$ sed [-n] [-e script] [-f sfilename] [filename ...]
```

- `sed` edits the file (standard input default) according to a script or command and redirect the edited version to the file (standard output default)
- The file goes through the editor (filter) line by line, where every line may or may not change
- There is an interactive editor `ed` that accepts the same commands – first one in Linux

`sed` - Introduction

- Each command is in the form of address and action
- The address decides if the action will be applied to the line or not
- The address can be either a line number or a pattern enclosed between two slashes `/pattern/`
- If address is not mentioned, the command is applied to every line
- If one address, the command is applied to that line, if two addresses, the command is applied to the range defined by the two addresses
- If two commands are applied at the same line, the second command will be applied to the "possibly" modified line by the first command

`sed` Command

- Explore `man sed.` Here are few useful flags:

flags	Meaning
`-n`	▪ Suppresses the automatic printing of the pattern space ▪ By default, `sed` prints every line after applying the script, but with `-n`, it only prints lines explicitly requested with `p` (print command).
`-e`	▪ Allows specifying a script (set of `sed` commands) directly from the command line ▪ This is useful when you want to provide multiple editing instructions in a single sed command
`-f`	▪ Specifies a file containing a list of `sed` commands ▪ Instead of writing the sed commands on the command line, they can be placed in a file, and this flag tells `sed` to execute the commands from that file
`-r`	▪ Enables the use of extended regular expressions (ERE) in the `sed` script, which allows for more advanced pattern matching features compared to basic regular expressions

sed – Action (delete) - Examples

Examples	Meaning
`d`	Delete all the lines
`2d`	Delete line 2
`1,4d`	Delete lines 1 through 4
`/^$/d`	Delete all blank lines
`7,/^$/d`	Delete lines 7 through the first blank line *(^: beginning, $: end)*
`/^$/,$d`	Delete from the first blank line to the last line
`/a*b/,/[0-9]$/d`	Delete from the line that contains b, ab, aab, …. to the first line that ends with a digit

Here **d** (delete) is an action

sed - Examples

```
$ sed –f sed_file f1.txt
```

- To read editing commands from a file named **sed_file**.
- The file contains one or more sed commands that you want to apply to the input text **f1.txt**

```
$ sed  -e ‘2d’ f1.txt
```

Address and action in single quote

f1.txt, before execution

This is line 1
This is line 2
This is line 3
This is line 4

f1.txt, after execution

This is line 1
This is line 3
This is line 4

sed - Examples

```
$ sed -e ‘1d’ -e ‘3d’ -e ‘5d’ f2.txt    # multiple commands

$ sed –f com1 f2.txt                    # multiple commands in com1

$ sed –n –e ‘1p’ f2.txt                 # -n suppress the lines to be printed
                                        # (by default), p is an action

$ sed ‘3d’ file                         # delete the 3rd line

$ sed ‘$d’ file                         # delete the last line

$ sed ‘/north/d’ file                   # delete all lines
                                        # that contains north
```

`sed` - Action (substitution) - Examples

Examples	Meaning
`sed 's/word1/aaa2/' f3.txt`	Looks for the first occurrence of `word1` in each line of the file `f3.txt` and replaces it with aaa2
`sed 's/word1/aaa2/g' f3.txt`	`g` is the the global flag, which tells `sed` to replace all occurrences of `word1` on each line, not just the first occurrence
`sed 's/\(Mar\)got/\1Liann/' f4.txt`	Substitutes `Margot` with `MarLiann`, where `Mar` comes from the first group (backreference \1), and `Liann` is directly added

Here `s` (substitution) is an action. By default, `sed` performs the substitution only on the first occurrence in each line

`sed` – Examples (read, append, transliterate)

Examples	Meaning
`sed ‘/Mar/r f1.txt’ f4.txt`	Searches for lines in `f4.txt` that contain the string `Mar`. After each matching line, it inserts the entire content of `f1.txt`
`sed ‘/Tol/a <------>’ f4.txt`	Searches for lines in `f4.txt` that contain the string `Tol`, and for every matching line, it appends the text `<------>` on a new line immediately after the matching line
`sed ‘1,3y/abcdef/ABCDEF/’ datafile`	Translates lowercase characters a, b, c, d, e, and f to uppercase A, B, C, D, E, and F, respectively, in lines 1 through 3 of the file

- The `r` command in `sed` is used to **`read`** and insert the contents of another file (f1.txt) after lines that match the pattern
- The `a` command stands for "**`append`**“
- the `y` command stands for "transliterate“, it **`translates or replaces`** characters

Recap - Pattern Matching

- The character `*` matches any string of characters including an empty string
- `?`: matches a single character (preceding is optional, with `-E`) `([abc]?)`
- `.` (dot) means any single character except a new line `\n`
- `[0-9]`: matches any digit
- `[a-z]`: matches any small case letter
- `[abc]`: `x[ab]y` matches `xay` and `xby` and not `xaby`
- `\c`: matches `c` only
- `a|b`: matches `a` or `b` in case expression only

Outside of character classes, the hyphen `-` is treated as a literal character. You can match it directly in patterns

https://www.linode.com/docs/guides/differences-between-grep-sed-awk/

Pattern Matching - *

- For * to be useful, it must follow something that it can quantify. For example, preceding character:

`␣`: matches space

Outside of character classes, the space ␣ is treated as a literal character. You can match it directly in patterns

`[␣]*`: zero or more occurrence of space

`[abc]*`: zero or more occurrences of any of the characters a, b, or c

Start and End of Line

- `^name` means `name` starts at beginning of line
- `name$` means `name` at end of line

Zero (one) or more occurrence

- `\?`: Zero or one occurrence
- `\+`: One or more occurrences
- `*`: Zero or more occurrences
- `{m}` OR `\{3\}`: Exactly `m` numbers of occurrences

```
$ sed –n ‘/0\{3\}/p’ numbers
```

find and print lines that contain exactly three consecutive zeros (000) from a file named numbers

- `{n,}` OR `\{n,\}`: At least `n` occurrences

```
$ sed –n ‘/10\{3,\}/p’ numbers
```

print lines from the file numbers that contain the substring 10 repeated at least three times

- `\{m,n\}`: `m` to `n` occurrences

String Manipulations

- String length

```
$ echo ${#string}
```

- Length of matching substring (if no match returns `0`, 1-based position)

```
$ expr match "$string" "$pattern"
$ expr match "$string" 'regex'
```

The `expr` command in UNIX/Linux is used to evaluate expressions and perform basic string operations, arithmetic, and regular expression matching

- Index (the first occurrence of any character in `$substring` within `$string`. If no match returns `0`, 1-based position)

```
$ expr index "$string" "$substring"
```

- Substring extraction (extract a substring from a given string starting at a specified position, 0-based position)

```
$ echo "${string: position}"
```

Always quote variables (`"$string"`) in expr to avoid unintended behavior

String Manipulation - Examples

```
$ a=Catalogue

$ echo ${#a}  → 9

$ echo ${a:2}  → talogue

$ echo ${a:2:3}  → tal
```

Pattern Removal

- Allows to remove substrings from the start or end of a string based on a pattern
- Shortest substring match (delete)
  - `${string#pattern}` strips shortest match of `$pattern` from front of `$string`
  - `${string%pattern}` strips shortest match of `$pattern` from back of `$string`
- Longest substring match (delete)
  - `${string##pattern}` strips longest match of `$pattern` from front of `$string`
  - `${string%%pattern}` strips longest match of `$pattern` from back of `$string`

Pattern Removal - Examples

```
$ a=abcdefabc

$ echo ${a#abc}  → defabc

$ echo ${a%abc}  → abcdef

$ a=abcdefghdal

$ echo ${a##a*d}  → al
```

Find and replace

- 'Find and replace' operations can be performed on strings using parameter expansion
  - `${string/pattern/replacement}  # replaces first match`
  - `${string//pattern/replacement}  # replaces all matches`
- Examples

```
$ a=abcdefabkljab

$ echo ${a/ab/XX}  → XXcdefabkljab

$ echo ${a//ab/XX}  → XXcdefXXkljXX
```

Programming with C

Introduction to C

- C was originally designed for and implemented on the UNIX operating system by Dennis Ritchie
- C is a general-purpose programming language, which features:
  - Economy of expression
  - Modern flow control and data structures
  - A rich set of operators
- C is considered a low-level language compared to modern high-level languages like Python or Java
- C is of minimalistic nature—it provides just enough abstraction to make programming easier while still allowing control over hardware

Introduction to C

- **C** is permissive - it assumes that the programmer knows what they are doing, offering flexibility but without much safety
  - For example, it doesn't automatically check for buffer overflows or memory leaks
- **C** is efficient, portable, powerful, and flexible
- **C** is more prone to errors because of its permissiveness and low-level nature
  - Errors examples: pointer issues, memory leaks etc.
- Beginners may find **C**'s syntax and concepts harder to grasp, especially compared to higher-level languages (next slide)

Obfuscated C

- Refers to **C code** that has been deliberately written to be difficult to read and understand

```
int v,i,j,k,l,s,a[99];

main(){

for(scanf("%d",&s);*a-s;v=a[j*=v]-

a[i],k=i<s,j+=(v=j<s&&(!k&&!!printf(2+"\n\n%c"-

(!l<<!j)," #Q"[l^v?(l^j)&1:2])&&++l ||

a[i]<s&&v&&v-i+j&&v+i-j))&&!( l%=s),v||

(i==j?a[i+=k]=0:++a[i])>=s*k&&++a[--i]);

}
```

What could be the Reasons?

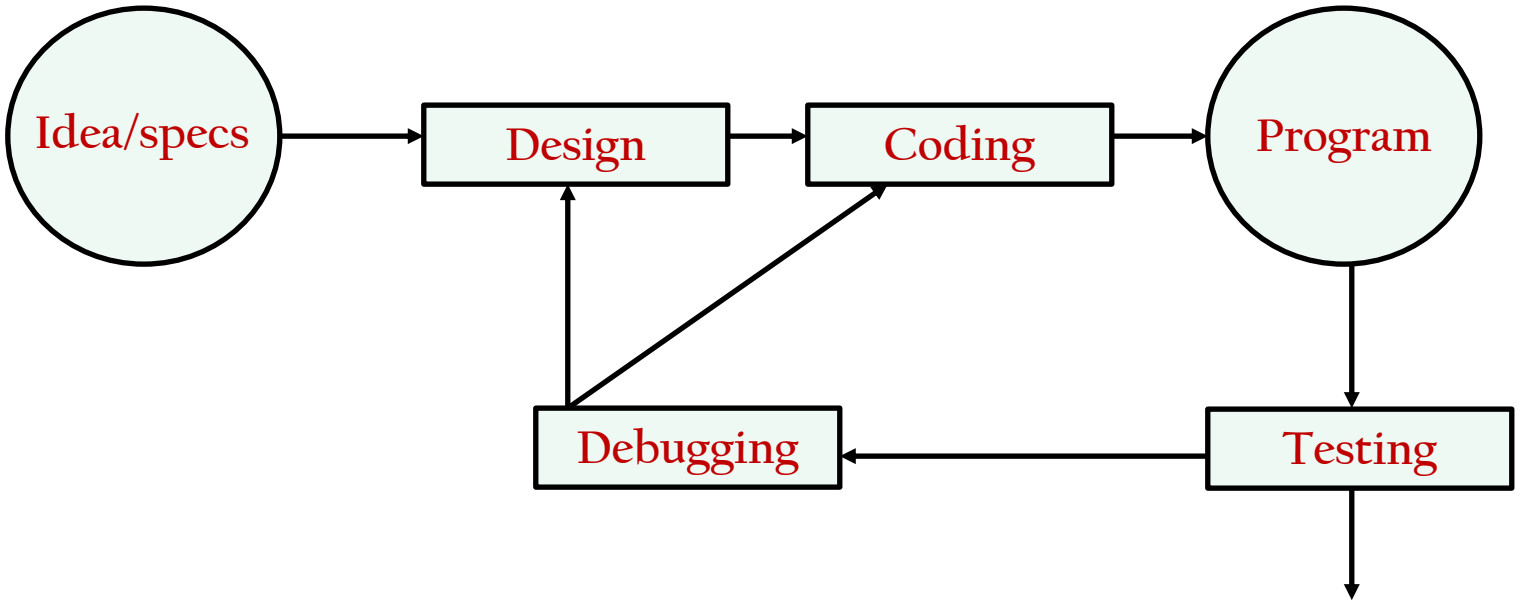
Tips

- Use tools to make programs more reliable
  - Use existing code library
    - For example, use standard library for functions like file handling, math operations, etc.
  - Adopt a sensible set of coding conventions
    - Indentation and braces
    - Naming conventions
    - Comments
    - Error handling
  - Avoid tricks and overly complex code
    - Code should be easy to maintain and extend by others in the future (Not like the one, we have seen in the previous slide)

Languages based on C

Programming Language	Key differences with C
C++	C with object-oriented features
Java	▪ C like syntax but much more restrictive – In Java, every variable and object must be declared with a specific type (cannot bypass these type restrictions), whereas C allows more flexibility with type conversions using pointers – C gives more control over memory using pointers, which is not the case in Java ▪ Java uses automatic garbage collection, whereas C allows manual memory management using `malloc()` and `free()`
C#	Described as "built over C", which alludes to its evolution from C and C++ with additional modern features and managed memory
Perl	Initially a scripting language, but "over time adopted many features of C", implying its growth in complexity and structure

Software Development Cycle



Idea:

Understand and define the problem to solve

Design:

Create the blueprint of the software

Coding:

Implement the design using programming languages

Testing:

Ensure the software works as expected

Debugging:

Identify and fix defects in the software

Software Specifications, Testing, and Debugging

- Specifications = LAW
  - The software must be developed strictly according to the specifications - the agreed upon requirements and constraints
- No Changes without Approval
  - Any improvement or change in functionality must go through a formal approval process
  - Ensures that every feature added or modified aligns with the original objectives
- If in Doubt, Ask
- First create test cases, then code – test driven development
- Test, if error, debug, repeat - cyclic testing and debugging
  - Testing is the process of looking for errors, debugging if found

Why Testing?

- **Testing can show the presence of faults, not their absence — Dijkstra**
  - The absence of errors during testing does not mean the system is completely free of faults
  - Testing alone cannot prove the software is perfect, only that it's functioning as expected under tested conditions
- Testing is very costly, in large commercial software, 1-3 bugs per 100 lines of code
  - Even with thorough testing, commercial software often contains some residual bugs
  - Some well-known examples on the next slide

Why Testing?

- 1990 AT&T long distance calls fail for 9 hours
  - Wrong location for `C break` statement
- 1996 Ariane rocket explodes on launch
  - Overflow converting 64-bit float to 16-bit integer
- 1999 Mars Climate Orbiter crashes on Mars
  - Missing conversion of English units to metric units
- Therac: A radiation therapy machine that delivered a massive amount of radiations killing at lease 5 people
  - Among many others, the reuse of software written for a machine with hardware interlock. Therac did not have hardware interlock

Types of Errors – 1. Syntax Errors

- Occur when the rules of the programming language are violated
  - Missing semicolons
  - Incorrect use of brackets or parentheses
  - Misspelled keywords
- Syntax errors are detected **before execution** during the compilation or interpretation phase
  - Modern Integrated Development Environments (IDEs) often highlight syntax errors in real-time
- <u>Fix: Correct the structure or code to follow the correct syntax</u>

1. Syntax Errors

```
#<include stdio.h>
int main ( );
(
    printf(‘Hello World’);
    /* Next line will output
            a name!  /*
    printf(“ Total is  %d \n”,total);
    printf(“Final result is \n,result);
}
```

```
#include <stdio.h>
int main()
{
printf(“Hello World”);
/*next line will output
a name */
printf(“Total is %d
\n”,total);
printf(“Final result
is \n”,result);
 }
```

2. Run-time Errors

- Occur **during program execution** due to unexpected conditions
  - Division by zero (`x = 5/0;`)
  - Accessing invalid memory locations
  - File not found errors when attempting to open a file (error message help to locate)
- Encountered when the program is executed with specific input or under specific conditions
- Fix: Identify the conditions that cause them and modify the program logic to handle those conditions

3. Logic Errors

Difficult to Find

- Occur when the program does not behave as expected, even though it runs without crashing
  - Using the wrong formula to compute a value
  - Incorrectly implemented algorithms (e.g., a search algorithm returning the wrong index)
  - Off-by-one errors in loops
- Discovered through **testing** and verification of the program's output
- Fix: Debugging logic errors requires careful review of the program's intended behavior

Getting Started

- The first program to write is the same for all languages

```
#include <stdio.h>
void main()
{
printf("hello, world\n");
}
```

```
#include <stdio.h>
void main()
{
printf("hello, world\n");
printf("Welcome to \"EECS2032\"\n");
}
```

To Compile: `gcc progname.c`

To Execute: a.out

Compiling C files with gcc – step by step

Some Explanations about the Program

- A `C` program, whatever its size, consists of functions and variables
  - A function contains statements that specify the computing operations to be done
  - Variables store values used during the computation
- Our example is a function named `main`
- Normally you are at liberty to give functions whatever names you like, but "`main`" is special - your program begins executing at the beginning of `main`
- This means that every program must have a `main` somewhere

Some Explanations about the Program

- `main` will usually call other functions to help perform its job:
  - Some of the functions you write
  - Others from libraries that are provided for you
- The first line of the program,

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

  tells the compiler to include information about the standard input/output library;
- One method of communicating data between functions is for the calling function to provide a list of values, called arguments, to the function it calls
- In this example, `main` is defined to be a function that expects no arguments, which is indicated by the empty list `( )`

Some Explanations about the Program

- The statements of a function are enclosed in braces `{ }`
- The function `main` contains only one statement:

```
printf("hello, world\n");
```

- A function is called by naming it, followed by a parenthesized list of arguments, so this calls the function `printf` with the argument `"hello, world\n“`
- `printf` is a library function that prints output, in this case the string of characters between the quotes
- The sequence `\n` in the string is `C` notation for the newline character

Special Characters and Escape Sequences

\n	New line
\t	Tab
\”	Double quote
\\	The \ character
\0	The null character
\’	Single quote

Special Characters - Examples

```
printf(“This is \t my answer %f in total \”%d\”\n”,x,y)
```

Output: This is my answer 1.200000 in total "4"

```
printf("C:\\Program Files\\MyApp\n");
```

Output: C:\Program Files\MyApp

```
char str[6] = {'H', 'e', 'l', 'l', 'o', '\0’};
printf("%s\n", str); // '\0' is the null terminator for the string
```

Output: Hello

Reading user Input + Commenting

`scanf` is used to read from the standard input

`&:` Memory address

```
scanf(“%d\n”,&i);
scanf(“%d%d\n”,&i,&j);
scanf(“%d,%d\n”,&i,&j);
scanf(“%d, %d\n”),&i,&j);
```

Two forward slashes (//) for single line comment

Multiline comment begin with /* and end with */

```
#include <stdio.h>
int main() {
    int number;
    printf("Enter an integer: ");
    // reads and stores input
    scanf("%d", &number);
    /* displays
       output */
    printf("You entered: %d \n", number);
    return 0;
}
```

Reading user Input

`scanf` is used to read from the standard input

```
scanf(“%d\n”,&i);
scanf(“%d%d\n”,&i,&j);
scanf(“%d,%d\n”,&i,&j);
scanf(“%d, %d\n”),&i,&j);
```

`&: Memory address`

Commenting

Two forward slashes (//) for single line comment

Multiline comment begin with /* and end with */

```
#include <stdio.h>
int main() {
    int number;
    printf("Enter an integer: ");
    // reads and stores input
    scanf("%d", &number);
    /* displays
       output */
    printf("You entered: %d \n", number);
    return 0;
}
```

Input/ Output - Standard

- Every program has a standard input and output (`stdin`, `stdout`, and `stderr`)
  - Standard Input (`stdin`):
    - Functions like `scanf()` or `getchar()` read input from the keyboard by default

```
int x;
printf("Enter a number: ");
scanf("%d", &x);  // Takes input from stdin (keyboard)
```

The `&` operator is used to pass the address of the variables x and y, so that `scanf` can store the input values at those memory locations

```
char c;
printf("Enter a character: ");
c = getchar(); // Read a single character from stdin
printf("You entered: %c\n", c); // Print the character
```

The `getchar` waits for the user to press `Enter`, so if you input multiple characters, `getchar()` will read them one by one on each subsequent call

Input/ Output - Standard

- Standard Output (`stdout`):
  - Functions like `printf` write output to the screen

```
int x = 5;

printf("Value of x: %d\n", x);  // Prints to stdout (monitor)
```

- Standard Error (stderr):
  - Error messages are sent to `stderr`, allowing normal output and error output to be handled separately

```
fprintf(stderr, "This is an error message.\n");
```

Input/ Output - Redirection

- **Input and output redirection** is not a feature of the **C** language itself but is instead managed by the **operating system** or **shell** in which a C program is executed
- You can use redirection in the **terminal/command-line** when running your **C** programs to redirect input or output to files
- For Example:

```
./my_program > output.txt
```

This command runs the program `my_program` and redirects the output that would normally appear on the screen to the file `output.txt`

```
./my_program < output.txt
```

This command runs `my_program` and uses the contents of `input.txt` as input instead of requiring the user to type the input manually via the keyboard

Variables

- In C, all variables must be declared before they are used:
  - Usually at the beginning of the function before any executable statements
- A declaration announces the properties of variables; it consists of a name and a list of variables, such as

```
int fahr, celsius;
float fr_num;
```

- The range of both `int` and `float` depends on the machine you are using

Data Types – 4 Basic

Representation	Data Type	Size/ Range	Declaration

`char`	Characters, can hold a single character (integer value based on their ASCII value) or a small integer	1 byte	`char letter = 'A';`
			`char num=10;`
			`Format specifier: %c, %d`

`int`	Integers	Usually 4 bytes	`int fahr, celsius;` `Format specifier: %d`
`float`	Single precision floating point numbers	Usually 4 bytes	`float fr_num;` `Format specifier: %f`
`double`	Double precision floating point numbers	Usually 8 bytes	`double pi = 3.141592653589793;` `Format specifier: %lf`

Modifiers/ `sizeof()`

- `unsigned`: Only stores positive numbers (0 and above) (default is `signed`, can omit `int(signed/unsigned)`)
- `long`: Increases the size of the integer (often 8 bytes on some systems)

  Also, `long long`
- `short`: Reduces the size of the integer (usually 2 bytes)

- `sizeof()`: used to determine the size, in bytes, of a data type or variable, `Format specifier: %zu` `sizeof(type)/sizeof(variable)`
  - printf("%zu", sizeof(int));

```
int a=2;
printf("%zu", sizeof(int));OR printf("%zu", sizeof(a))    // outputs 4
```

Characters

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Extended ASCII: Extended versions of ASCII use values from 128 to 255 to include additional characters (like symbols and characters from other languages)

Integer Suffixes

1. Long Integer (`%ld`)
   - You can explicitly declare a long integer by appending `L` or `l` to the number

```
long int long_num = 7L;
```

2. Unsigned Integer (`%u`)
   - To declare an unsigned integer, append `U` or `u` to the number

```
unsigned int unsigned_num = 8U;
```

Floating Point Representation

1. Normal Decimal Representation
   - Standard Decimal Values
   - Example: `24, 23.45, 123.45e-8` (scientific notation).
2. Suffixes for `float (%f, %.2f, etc.)` and `double (%lf, %.10lf)`
   - To specify a float explicitly, append `F` or `f` to the value

```
float float_num = 3.4F;
```

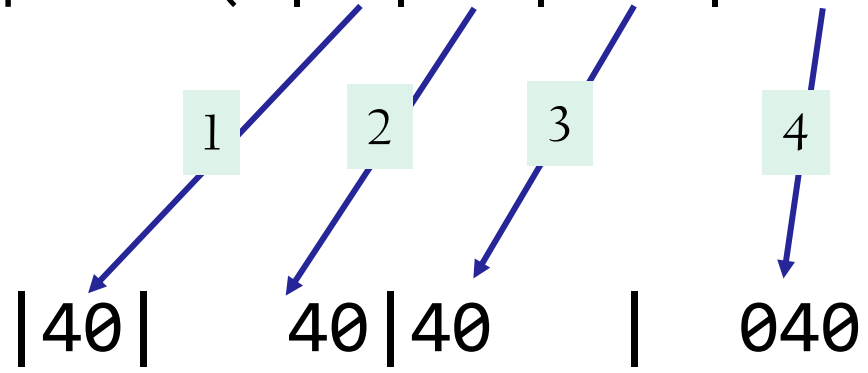
```
double scientific_num = 123.45e-8;
```

   - To declare a `long double (%.2Lf)`, append `L` or `l` to the value

```
long double long_double_num = 2.15L;
```

Formatting Output - `int`

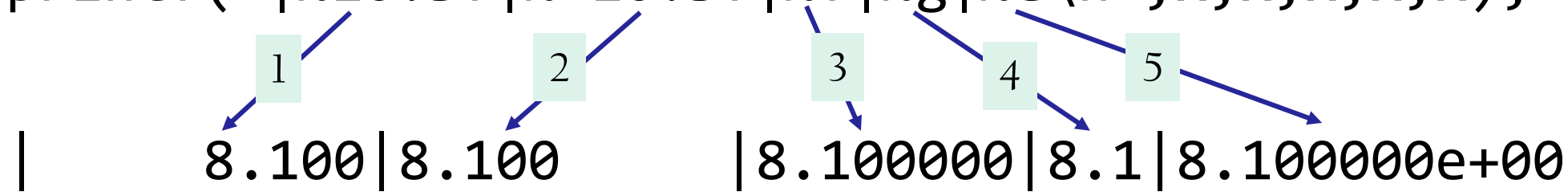
```
int i=40;
printf("|%d|%5d|%-5d|%5.3d\n",i,i,i,i);
```



1. Normal integer
2. Right-aligned, 5-character wide field
3. Left-aligned, 5-character wide field
4. Right-aligned, at least 5 characters wide, with 3 digits (padded with leading zeros if necessary)

Formatting Output - `float`

```
printf("|%10.3f|%-10.3f|%f|%g|%e\n",x,x,x,x,x);
```



1. Right-aligned, 10-character wide field, with 3 decimal places
2. Left-aligned, 10-character wide field, with 3 decimal places
3. Default floating-point, 6 decimal places
4. Shorter of `%f` or `%e`, with no trailing zeros
5. Scientific notation

Fixed-width Integer Types (`<stdint.h>`)

- Particularly useful in embedded systems programming or when writing portable code, where the exact size of an integer is crucial

Representation	Data Type	Range
`int8_t`	Signed 8-bit integer	-128 to 127
`uint8_t`	Unsigned 8-bit integer	0 to 255
`int16_t`	Signed 16-bit integer	-32,768 to 32,767
`uint16_t`	Unsigned 16-bit integer	0 to 65,535
`int32_t`	Signed 32-bit integer	-2,147,483,648 to 2,147,483,647
`uint32_t`	Unsigned 32-bit integer	0 to 4,294,967,295
`int64_t`	Signed 64-bit integer	very large, typically used for large numbers
`uint64_t`	Unsigned 64-bit integer	-do-

No String Data Type

- In C, a string is not a distinct data type as it is in many other programming languages
- Instead, a string is represented as an array of characters terminated by a null character (\0)
- This null character signals the end of the string
- Here's a breakdown of how strings are used in C:

```
char greetings[]="hello"
```

'H'	'e'	'l'	'l'	'o'	'\0'

- Format Specifier: %s

More on strings later

No Boolean Data Type

- In C, there is no built-in Boolean data type as found in many other programming languages
- Boolean values can be represented as:
  - 0 for False
  - 1 (or any non-zero value) for True
- You can represent Boolean values using other data types:

1. Typically, using `int`:

```
int isTrue = 1; // Represents true
int isFalse = 0; // Represents false
```

2. By including specific headers (since C99 `<stdbool.h>`) that define a Boolean type

```
bool isTrue = true; // Represents true
bool isFalse = false; // Represents false
```

Naming Variables

- Combination of letters, numbers, and underscore character _
- Can not start with a number and cannot be a keyword
  - Like `int`, `return`, `if`, etc.
- Valid Examples:
  - `Abc abc5 aA3_`
- Invalid Examples:
  - `5sda`

Symbolic Constants

- `define` preprocessor directive (used to create `macros` in C):
  - `#define N 5`
  - This specific `#define` directive assigns the value `5` to the identifier `N`
  - Everywhere **`N`** appears in the code, it will be replaced with **`5`** throughout the code

Number Systems in C

Number System	Base	Example
Decimal (`%d`)	`10`	`123487` (normal integer representation)
Octal (`%o`)	`8`	`0654` (number start with `0`, `0` to `7`) – equivalent to `6*8^2 + 5*8^1 + 4*8^0 = 428` in decimal
Hexadecimal (`%x`)	`16`	`0x4Ab2` (number start with `0x` or `0X`, `0` to `9` and letters `A` to `F`) – equivalent to `4*16^3 + A*16^2 + b*16^1 + 2*16^0 = 19122` in decimal

Number Systems in C

- Different number systems are useful for different reasons
  - **Decimal:** General-purpose calculation, Human-readable form
  - **Octal:** To manipulate file permissions directly in `UNIX/Linux OS`, using system calls like `chmod()`
  - **Hexadecimal:** Ideal for memory addresses, bitwise operations, and compact representation of large binary numbers
    - Often hardware documentation specifies register values and memory addresses in hexadecimal, so knowing how to use hexadecimal makes interfacing with hardware easier

Basic Arithmetic Operations

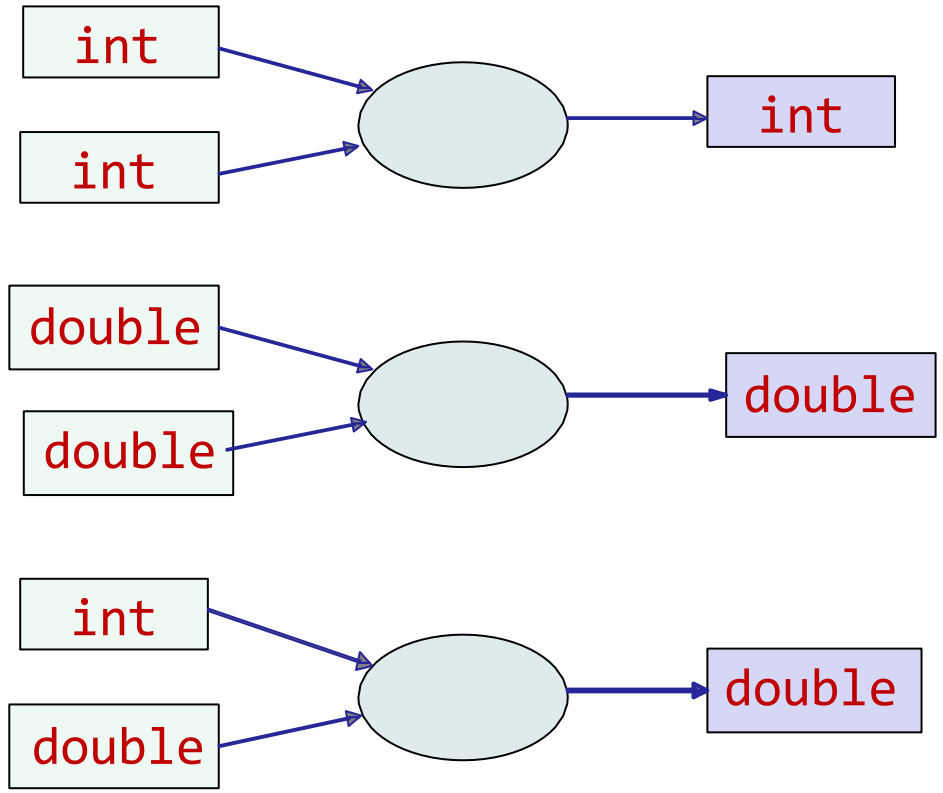
Operation	Operator	Example
Addition: Adds two operands	+	a+b
Subtraction: Subtracts the second operand from the first	-	a-b
Multiplication: Multiplies two operands	*	a*b
Division: Divides the numerator by the denominator. If both the operands are integers, it performs integer division (the fractional part is discarded)	/	a/b
Modulus: Returns the remainder of division between two integers	%	a%b

Basic Arithmetic Operations

Important Notes:

- **Integer Division:** If you divide two integers, the result will also be an integer. To get a decimal result, at least one operand must be a floating-point number (e.g., use `float` or `double`)
- **Modulus Operator:** This operator is only applicable for integers. Using it with floating-point numbers will result in a compilation error
- **Operator Precedence:** `C` follows operator precedence rules. For example, multiplication and division have higher precedence than addition and subtraction

Mixed Type Arithmetic




```
int x=5, y=2, w;  double z, q = 2;


z = x/y;    // z = 2.0
w = x/y; // w = 2
z = x/q; // 5/2.0 // z = 2.5
w = x/q; // w = 2
```

Mixed Type Arithmetic

```
17 / 5 = 3

17.0 / 5 = 3.4
```

```
9 / 2 / 3.0 / 4 = ?
9 / 2
4 / 3.0
1.333 / 4
```

```
=4
=1.333
=0.333
```

Mixed Type Arithmetic

- How do you cast variables?
- For example:

```
int varA = 9, varB = 2;  double varC;

varC = varA/varB; /* varC is 4.0. varA/varB gives 4
as both are of type int. The result is assigned to
varC, which is of type double*/

varC = varA/(double)varB; // varC is 4.5
```

Doesn't change the value of `varB`, just changes the type to `double`

Pre- & Post- Operators

- ++ (incrementing) or -- (decrementing)
- If placed in front of the variable, incrementing or decrementing occurs BEFORE the value is assigned

```
i = 2, k = 1;
k = ++i;
// i = i + 1; 3
// k = i; 3
```

```
k =--i;
// i = i - 1; 1
// k = i; 1
```

- If placed at the back of the variable, incrementing or decrementing occurs AFTER the value is assigned

```
i = 2, k = 1;
k = i++;
// k = i; 2
// i = i+1; 3
```

```
k =i--;
// k = i; 2
// i = i - 1; 1
```

Boolean Expressions

- **Relational Operators**
  - Used to compare two values or expressions
  - They return either 1 (true) or 0 (false) based on the result of the comparison

Operator	Function	Example
==	Checks if two values are equal	a == b
!=	Checks if two values are not equal	a != b
<	Checks if the left operand is less than the right operand	a < b
<=	Checks if the left operand is less than or equal to the right operand	a <= b
>	Checks if the left operand is greater than the right operand	a > b
>=	Checks if the left operand is greater than or equal to the right operand	a >= b

Boolean Expressions

- **Logical Operators**
  - Used to combine two or more conditions or to negate a condition
  - Often used in control structures like `if`, `while`, and `for`

Operator	Function	Example
`&&`	True only if both conditions are true (`1`)	`if (a > 0 && b > 0)` is true (`1`) if both `a` and `b` are greater than `0`
`||`	True if either of the conditions is true (`1`)	`if (a > 0 || b > 0)` is true (`1`) if either `a` or `b` is greater than `0` (or both)
`!`	Inverts the value of a condition (true (`1`) becomes false (`0`), and false (`0`)becomes true (`1`))	`if (!(a > 0))` is true (`1`) if `a` is `not` greater than `0` (i.e., `a <= 0`)

Introducing `while` loop (for the lab4 only)

```
while (expression)
   statement
```

- The ***expression*** is evaluated. If it is non-zero, ***statement*** is executed, and ***expression*** is reevaluated
- This cycle continues until ***expression*** becomes zero, at which point execution resumes after statement

More details: we will cover later

Thank You!

Happy Learning