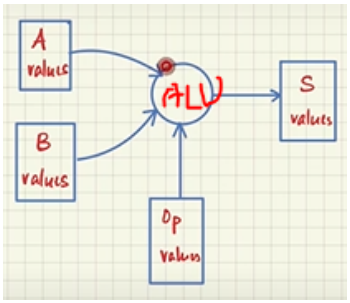
Week1 (How a computer works)

- A computer doesn't necessarily mean a desktop or laptop, but the whole system, including the CPU, memory, storage, and other components.
- Numbers can be represented as electrical signals. There is a circuit inside the CPU that can add such signals.
- Adding two numbers each stored as part of two number lists can be achieved by storing each list into memory, retrieve the numbers from either lists, perform addition and repeat this with all pairs of numbers. This can be generalized to perform any operation, other than addition - including math/logical operations, or operations like *sqrt, cos, sin, exp* etc.
- Memory is like a bag of words, where objects can be stored (written) or retrived from (read). Location of each word in the bag is called its *address*.
- Each word is represented by a set of bits. The number of bits used is called the word *width*.
- There is a finite space available to store the words, and this is called the *capacity*.
- Not all CPUs offer functions other than addition, natively. In such cases, those functions are performed through Arithmetic & Logic Unit (ALU) by emulating the corresponding operations.



- Each of the *Operands*, *Operation* and *Result* could be stored inside the same memory block, and referred to individually through an *address map*.
- *Operands and Result* constitute the *data*, and *Operation* constitute the *instructions*, both of which are stored in memory . Both of these can be represented using bits.
- In a *stored program computer, data* and *instructions* are stored in a mixed fashion.
- However, it's preferred to store the *instructions* in one memory segment (referred to as program/code) and store the *data* in a different memory segment. A different memory segment is used to *store* the peripherals of the system.
- This scheme works for the most part, until the computer needs to handle higher amounts of memory. When this happens, number of gates and number of connections required to decode the memory slots grow a lot higher. This demands more *chip area* and causes causes the memory to work much slower than the ALU.
- This problem is solved by using a split-memory architecture, wherein the memory is

divided into the *main memory* and *registers* that remain closer to the ALU. *Registers* are used for temporary storage, but works extremely fast since they're closer to the ALU. Data gets *loaded* from the main memory into the registers, thus allowing ALU to perform operations using these registers, after which their contents get *stored* back into the main memory.

- In a computer with 4GB RAM, there'll be 4 billion bytes of storage, but there'll be far lesser number of registers (~30) in the CPU.
- CPU keeps track of the position of next instruction through a *program counter*. It increments by 1, after each item in a sequence of operations is completed.
- For repeating the same sequence of operations n times, we also need to keep track of a *loop counter*. After all operations in the sequence is executed, increment the *loop counter by 1* and compare with n. Also, reset the *program counter* to the first operation in the sequence, using a *JMP* instruction.
- When CPU tries to read data out of a memory address, the following steps happen essentially.
  - CPU posts an address (set of bits) to the address bus.
  - An address decoder turns on appropriate memory cells in the memory circuit.
  - The output of these flip-flops are transferred back into the CPU via the data bus.
  - For certain specific addresses, the memory interprets them as peripherals and activates the corresponding devices, thus allowing the CPU to communicate directly with these external devices.
- CPU reads the values from the peripherals by reading the value at the memory address mapped to them.
- *Control Unit* is responsible for directing the operation of the processor. It fetches instructions from memory, decodes them to understand what actions need to be performed, and then coordinates the execution of these instructions by sending signals to the ALU (Arithmetic Logic Unit), registers, and other components.
- Algorithms + Data Structures = Programs (Nicholos Wirth, 1976)
- PDP = Programmable Data Processor.
- Unix OS invented by Ken Thompson and Dennis Ritchie in 1970's
- Primary motivation for inventing a programming language is to allow portability of programs, across various processors. C was invented by Dennis Ritchie.
- C language uses a bootstrapped compilation process.
  - Stage 1: A very basic C compiler is created in assembly language or another high-level language.
  - Stage 2: This basic compiler compiles a simple version of a C compiler written in C.
  - Stage 3: The simple C compiler compiles a more advanced version of itself, iteratively improving the compiler's capabilities.
- Main advantage of this process is the easy portability of C programs to other

systems/hardware. It suffices to create a stage-1 compiler for the new hardware (using its own assembly language). This step allows to bootstrap itself to create more advanced compilers for C programs, for the hardware.

- C is imperative, has a static type system, is weakly typed (allows conversion from one type to another), and allows structured programming. C programs looks like a free form text, and hence easy to store and share.

Week2 (Data representation)

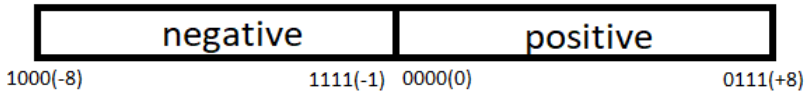
- Numbers when represented in decimal system, uses place values that equals powers of 10. Thus, place value of unit's place is $10^0 = 0$ (hence giving it a place value of 1), place value of ten's place is 1 (hence giving it a place value of $10^1 = 10$) etc.
- Conventions are at the centre of all data representations in a computer system. It can be interpreted suitably, based on where in memory the strings of 0's and 1's are stored, and what is stored before/after it.
- $5_{(10)} = 101_{(2)}$
- How to convert a decimal number to a binary? Divide the number by 2 repeatedly, each time replacing the number with the quotient from the previous step. Read out the remainders at each step, in reverse. Let's try find the binary representation of $27_{(10)}$.
  $27 \div 2 = 13 \;\; Remainder = 1$
  $13 \div 2 = 6 \; Remainder = 1$
  $6 \div 2 = 3 \; Remainder = 0$
  $3 \div 2 = 1 \; Remainder = 1$
  $1 \div 2 = 0 \; Remainder = 1$

  Read the remainders at each step from bottom to top. Thus, the binary representation of 27 is 11011
- How to convert a fraction from decimal system to binary? Multiply the number by 2 repeatedly, while remove the mantissa from the result each time. Read out the mantissa of the results. For example, to convert $1.23_{(10)}$ to binary, do the following.
  - Convert the integer part (1) first to binary. $1_{(10)} = 1_{(2)}$
  - For the fractional part 0.23, do the following calculations.
    - $0.23 * 2 = 0.46 \; (integer \; part: \; 0)$
    - $0.46 * 2 = 0.92 \; (integer \; part: \; 0)$
    - $0.92 * 2 = 1.84 \; (integer \; part: \; 1)$
    - $0.84 * 2 = 1.68 \; (integer \; part: \; 1)$
    - $0.68 * 2 = 1.36 \; (integer \; part: \; 1)$
    - $0.36 * 2 = 0.72 \; (integer \; part: \; 0)$

  One might continue, depending on the precision one needs. If one stops at this step, the result can be read it out as $1.001110_{(2)}$

- To convert a binary representation to decimal equivalent, multiply the integer part with $2^0, 2^1, 2^2, 2^3$.... and the fractional part by $2^{-1}, 2^{-2}, 2^{-3}$... and add all results.
- Any number of 0's can be prepended to the binary representation, without changing the value. Thus, $11011_{(2)} = 00000011011_{(2)} = 27_{(10)}$. This means, the minimum number of digits to represent 27 in binary is 5.
- 8 bits is typically called a byte, and 4 bits is called a nibble.
- In order to recognize where a binary number starts and where it ends, it's important to follow a convention of adhering to a certain number of bits for each number. For example, if a number is represented using 1 byte (8 bits), the system could represent numbers from $00000000_{(2)}$ to $11111111_{(2)}$. Thus, the lowest decimal number that could be represented is 0 and the highest is 255 when 8 bits are used. Note that the maximum number that can be represented using N bits is $2^N - 1$.
- With a 32 bit representation for (binary) numbers, each binary number would be represented using 32 bits, and thus represent $2^{32} - 1$ numbers inside the computer's memory.
- The left-most digit in the representation is called Most Significant Bit (MSB) and the right-most bit is called Least Significant Bit (LSB)
- A programming language like C establishes a primitive data type, its representation and makes it portable.
- In order to represent a negative number, there are two options.
  - Use a convention where MSB will be 1 for negative, 0 for positive. For example, in a 4 bit signed representation, then $0011_{(2)} = +3$ and $1011_{(2)} = -3$. In this case, N bits can be used to represent $2^{N-1} - 1$, instead of $2^N - 1$, since 1 bit is used up for the sign. The downside is that it'll have 2 representations for 0: one for +0 and another for -0!
  - Use two's complementation notation. If number is positive, store it as a regular binary number. If number is negative, store it as $2^N - number$. For example, $+17$ is represented as 00010001, and -17 is represented as $2^8 - 17 = 239_{(10)} = 11101111_{(2)}$. Note that the largest number that could be represented using 8 bits is $01111111_{(2)} = +127_{(10)}$.
- Two complement's notation makes it easier to subtract two binary numbers - convert the second number to two's complement and add them.
- Since, binary numbers aren't easy to read and interpret, we use hexadecimal notations. Thus, $01011011_{(2)} = 91_{(10)} = 5B_{(16)}$
- 16-bit binary values can range from $0x0000$ to $0xFFFF$. In two's complement notation,
  - Positive numbers: ranges from $0x0000$ (0_{10}) to $0x7FFF$ (32767_{10}),
  - Negative numbers: ranges from $0x8000$ (-32768_{10}) to $0xFFFF$ (-1_{10}).
- 32-bit binary values can range from $0x00000000$ to $0xFFFFFFFF$. In two's

complement notation,
  - Positive numbers: ranges from $0x00000000$ (0_{10}) to $0x7FFFFFFF$ (2147483647_{10}),
  - Negative numbers: ranges from $0x80000000$ (-2147483648_{10}) to $0xFFFFFFFF$ (-1_{10}).
- How to convert a negative decimal number to a binary in 2's complement? Start with the binary representation of the absolute value of the decimal number. Invert all the bits, and add 1 to it. Let's try to find the binary representation of $-2_{(10)}$ using two's complement.
  $Convert\ 2\ to\ binary: 2_{(10)} = 0000000000000010_{(2)}$
  $Invert\ all\ bits: 1111111111111101_{(2)}$
  $Add\ 1\ to\ inverted\ number: 1111111111111110_{(2)}$
  NOTE: This can also be written as $0xFFFE_{(16)}$
- How to convert a binary in 2's complement to a decimal number? Invert all the bits, and add 1 to it. Convert the result to a decimal number. If the original number starts with 1, add a negative sign to the result.
- Here's a pictorial representation of the range of numbers represented using 2's complement notation on a 4 bit number.

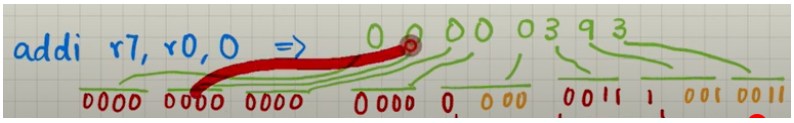


- To represent floating point numbers, there are two convensions.
  - Use fixed point notation and stored as say 16-bit integers, and multiplied with fixed scaling factors (say, 2^{-8}) while reading/interpreting them.
  - Use floating point notation, and stored along with the exponent value. For example, the computer uses 32 bits to store the number, wherein 1 bit is used for sign, 8 bits for exponent and the remaining 23 bits for the actual value(mantissa).
- The smallest 32 bit number that could be stored using **single precision** is 10^{-38} and the largest number is 10^{+38}.
- The smallest 64 bit number that could be stored using **double precision** is 10^{-308} and the largest number is 10^{+308} and is used in scientific computing.
- ML applications etc typically use half precision, and uses 16 bits to store the numbers.
- Encoding of characters (alphabets, digits and punctuations) was done using ASCII (using its 7 bit system), until characters other than English language came along. Unicode represent the characters from various languages using code points, and could use a varying number of bits to represents them. UTF-8 is the most popular standard in Unicode, which could represent English language alphabets very similar to ASCII.
- UTF-8 and UTF-16 are variable length encoding techniques, whereas UTF-32 is a

fixed length encoding (uses 32 bits).

- We'll now see how to encode instructions like ADD, SUB, LOAD, STORE, JUMP etc.
- Instruction Set Architecture (ISA) is a part of a computer architecture that defines the set of instructions the computer can execute. It acts as the interface between software and hardware. Some well-known ISAs include x86, ARM, MIPS, and RISC-V.
- To give an example of how an instruction can be encoded, consider the instruction $ADDI\ r7,\ r0,\ 0$. It is formulated as

It essentially adds 0 to $r0$ (rs1) and stores result in $r7$ (rd). Assuming a 32 bit RISC-V architecture, $r0$ is represented as 5-bit number 00000 and $r7$ is represented as 5-bit number 00111 and the immediate value to be added is represented by a 12 bit number 000000000000. Putting all of this together, the above instruction can be represented as



Note: the pattern given in orange above codifies $ADDI$ instruction.

- Code and data are stored in the same memory block, and separated from each other only by convention. For example, out of the total 1000K (1MB) memory locations, the first 400K could store code, and the next 600K could store data.
- Compilation is the process where a human readable program gets transformed to machine interpretable program.
- At a high level, compilation does syntax check, map statements into equivalent code, and finally generate actual machine code. Compiler could do optimizations, including improvements in speed or program size. Note that compilers can be CPU dependent.
- Multi-processing allow multiple programs to run concurrently - run each for a few milliseconds, before swtiching to the next program, thus giving the illusion to the user that all programs are running simultaneously.
- OS decides which program executes when, and for how long. It orchestrates the control flow between the programs, by loading and executing them.
- BIOS is a minimalistic program to run first at the start up of the computer, by setting the program counter to a known value. BIOS is stored in ROM. It initializes hardware components like the CPU, memory, and peripherals. The BIOS locates and loads the bootloader from the chosen storage device (such as an HDD, SSD, or USB drive) into memory. The bootloader is a small program responsible for loading the operating system. Once the OS kernel runs, it loads various applications (like graphics drivers etc) to memory as required. The process by which BIOS loads the OS kernel is often termed bootstrapping.

Week3 (Programming)

- All C programs have a *main* function (the entry point) and typically start with pre-processor statements (*#include* statements)
- The C runtime, also known as the C runtime library (CRT), is a collection of software routines that support the execution of C programs. It provides essential functionalities required during program execution, such as
  - Memory mgmt - malloc, calloc, realloc, free
  - Input/output operations - printf, scanf, fopen, fclose, fread, fwrite
  - String Manipulation - strlen, strcpy, strcat, strcmp, strstr
  - Math operations - sqrt, pow, sin, cos
  - Process Control - exit, system
  - Utility functions - qsort, bsearch

  Note that the runtime can be statically or dynamically linked (from shared libraries like msvcrt.dll in Windows).
- $\#define\ N\ 10$ defines a constant N having a value 10.
- $printf$ is part of the *stdio.h.*
- Functions must specify return value in the signature, unlike Python.
- Variables need to be declared as belonging to a specific type, unlike Python.
- Compiler checks syntax, but not semantics (logic)
- Variables can refer to memory locations or locations inside CPU registers.
- Variables names can consist of alphabets (small or capital letters), digits and underscore. No other characters are allowed. Variable cannot start with digit or two underscores. Using camelcase for naming variables is a convention.
- Variable can belong to one of the types - *int, short, char, float, double*. *int* typically uses 4 bytes, *short* uses 2 bytes and *char* uses 1 byte. *float* uses 4 bytes too, *double* uses 8 bytes and *long double* uses 16 bytes. Note that the most hardware doesn't support *long double* natively, and hence computations are much slower than that with *float*.
- Types can be converted between each other using *typecasting*
- Variable location cannot be decided by the programmer, but decided by the compiler in association with OS.
- THere is no *boolean* type in C, but zero is treated as false and non-zero is treated as true.
- *Operators* are mostly resolved at compile time. *Functions* are generally resolved at run time, although there are exceptions based on the context.
- *Operator overloading* is not allowed in C language.
- $5/2\ ==\ 2$, whereas $5/2.0\ ==\ 2.5$
- Ternary operator has 3 components.
  <condition> ? <value_when_condition_is_true> : <value_when_condition_is_false>
- $i++$ is post-increment operation. $++i$ is pre-increment operation. For example,

$j = i++$ assigns the current value to j and then increment i, whereas $j = ++i$ increment i first and assigns it to j.

- $i = ++i$ increments the value i and then assigns back to i, whereas $i = i++$ assigns the current value of i back to i, and increments it. But, the last increment gets ignored. Thus, the following program prints $Value\ of\ i:\ 3$ at lines 10 and 12.

```
1   #include <stdio.h>
2
3   void main() {
4       int i = 0,j = 0;
5       j = i++;
6       printf("Value of i: %d, Value of j: %d\n", i, j);
7       j = ++i;
8       printf("Value of i: %d, Value of j: %d\n", i, j);
9       i = ++i;
10      printf("Value of i: %d\n", i); //i = 3
11      i = i++;
12      printf("Value of i: %d\n", i); //i = 3
13  }
```

- gcc is a commonly used C compiler. In order to compile $test.c$ using *gcc* and create the executable file $test.exe$, use the command $gcc\ test.c\ -o\ test.exe$.
- $a = b = 10;\ c = 20;$ assigns 10 to a and b, and assigns 20 to c. However, $a = (b = 10;\ c = 20)$ assigns 20 to a (the last assignment in the RHS), b is assigned with 10 and c with 20.

```
3   void main() {
4       int a, b, c;
5       a = b = 10; c = 20;
6       //Output: a = 10, b = 10, c = 20
7       printf("a: %d, b: %d, c: %d\n", a, b, c);
8
9       a = b = 10, c = 20;
10      //Output: a = 10, b = 10, c = 20
11      printf("a: %d, b: %d, c: %d\n", a, b, c);
12
13      a = (b = 10, c = 20);
14      //Output: a = 20, b = 10, c = 20
15      printf("a: %d, b: %d, c: %d\n", a, b, c);
16  }
```

- Conditions can be combined using && and ||. Use ! to negate a condition.
- *sizeof* operator returns the size in bytes of an object, data type. It returns an unsigned integer value of type $size\_t$ representing the size of the object or data type. Note that *sizeof* is an operator, and not a function. It executes during compile time, unlike a function which executes during runtime.
- Conversion from small to higher data type (eg.int to float) is implicit as demonstrated by this code, but that from higher to small data type (eg.float to int) must be explicit.
- $char$ data promoted to int if the latter can represent all the values of the original type; otherwise, they are promoted to $unsigned\ int$.
- If either operand is of type $double$, the other operand is promoted to $double$. If either

operand is of type $float$, the other operand is promoted to $float$.

- When unsigned integer is reduced from *0*, it wraps around to the maximum value representable by an unsigned int due to underflow. Thus, in the below code, *line #8* results in the maximum value of unsigned integer - *4294967295*

```
1  #include <stdio.h>
2
3  void main() {
4      unsigned int a = 1;
5      a--;
6      printf("%u\n", a);
7      a--;
8      printf("%u\n", a);
9  }
```

-

Week4 (Structured programming)

- Flowcharts use symbols to indicate different operations.
  - Ellipse for Start/Stop
  - Parallelogram for I/O operations
  - Rectangle for computations
  - Diamond for conditionals
- Pseudo code - Neither assembly or a program.
- Control flow dynamics
  - Sequence - Blocks, Subroutines, Functions, Scope
  - Selection - if-else, switch-case
  - Iteration - for, while, do-while
- During compilation process, dead code can be detected by the compiler and removed.
- If a variable is declared but not initialized, it could contain *any value* of the declared type.
- The variable can be initialized in the same line as it's declared. Thus $int\ a = 0, b = 0, c = 0$ declares 3 integer variables a, b, c and initializes all of them to 0.
- *do-while* is guaranteed to run at least once, unlike the *while* loop (which will run only if the condition is true)
- In a *switch-case* statement, use a *break* stmt mandatorily after each *case*. If there's no *break* stmt, the cases are no more independent, but works together instead. For example, the following code prints 0, because the first case ($choice\ ==\ 4$) doesn't have a $break$ and runs all the subsequent cases, even though the value of $choice$ do not match them.

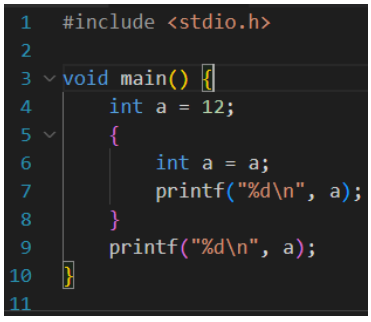
```c
#include <stdio.h>
void main() {
    int a = 12, b = 6;
    int choice = 4;
    switch (choice)
    {
    case 4:
        choice = a / b * 4;
    case 7:
        choice = b / a * 6;
        break;
    case 8:
        choice = (b + 12) / a;
    }
    printf("%d", choice);
}
```

- Each *case* in a *switch-case* statement can have multiple lines in the block.

Week5 (Functions)

- Reusability of code is the main purpose = Don't Repeat Yourelf.
- Generalizability (using parameters) is another advantage of a function.
- Blocks can be created using curly brackets, and is most fundamental in a function definition.
- Normally, the function will not perform any I/O operations inside it. Instead, it'll use parameters or return a value(s).
- $stdio.h$ contains the declaration of the C-runtime functions like $printf$ and $scanf$ (which are defined in the cstd library). In order to declare a function, we need the return type, function name and the argument names.
- In order to call a function, it must be defined before the call, or must be declared before the call.
- If you need to distribute a function you wrote to another developer, follow these steps.
  - Write the function in, say *my_functions.c*
  - Create a header file (say *my_functions.h*) that contains the function prototype/signature.
  - Distribute the files, *my_functions.c* and *my_functions.h.* Optionally, distribute the object file *my_functions.o*
  - $\#include\ my\_functions.h$ in the first line of *main.c*.
  - Compile and link using the command $gcc\ main.c\ my\_functions.c\ -o\ my\_program$, if *my_functions.c* is available. Else, it can be linked using the command $gcc\ main.c\ my\_functions.o\ -o\ my\_program$

- Function variables are scoped or restricted to only the functions they're declared in, and not visible outside. In gcc compiler, it's allowed to have two variables with the same name in different scopes - say, inside as well as outside a function. Other compilers don't allow this, so it's better to avoid such declarations.
- Similar to Python, parameters are passed into function arguments as copies.
- It's not possible to return multiple values from a function, like in Python.
- Unlike Python, it's not possible to raise exceptions in C.
- After compilation/linking process, an executable file is produced which will contain a lookup table that tells which address locations are each function located.
- Typically, CPU registers are used by a C function to store arguments. When the function is called, the parameters use these registers to pass values to its arguments. Return values from the function also uses CPU registers.
- C Runtime ensures that the functions are allocated memory (stack-frames) dynamically to hold variables and other data. Each call to the function will create a new stack-frame. When a function is called, the caller needs to save the existing stack address, so that the C Runtime knows where to return to, when the callee is exited.
- Running recursive algorithms beyond a limit can create segmentation fault, because stack-frames created during each iteration can add up beyond the memory allocated to the program currently being executed.
- C language uses block-level scoping of variables. The same variable name can be redefined inside a nested block, without affecting the variable in the outer scope.
- Consider the following code



```
1  #include <stdio.h>
2
3  void main() {
4      int a = 12;
5      {
6          int a = a;
7          printf("%d\n", a);
8      }
9      printf("%d\n", a);
10 }
11
```

Here, variable a is declared in two scopes within the *main* function. In line #6, it seems like the variable a in the inner scope is redefined and assigned with its value from the outer scope. However, it assigns a value stored in the variable a declared in the same line #6. Since declaring variable a initializes it with a random value, a gets assigned with the same random value.

Week6 (Memory and Pointers)

- RAM is volatile and loses information when the computer is restarted. It gets reset to a garbage value, and not essentially 0.
- Every byte of memory has an associated addresses.
- $1KB \equiv 2^{10}\ bytes \approx 10^3\ bytes.$ $1MB \equiv 2^{20}\ bytes \approx 10^6\ bytes.$ $1GB \equiv 2^{30}\ bytes \approx 10^9\ bytes.$
- Pointers store address, while variables store data at that address.
- Some typical operations with pointer are shown below.

```c
void main() {
    int a;

    int* p;
    printf("Pointer initialized at %p\n", p);

    p = &a; //Assigning the address of a to p
    printf("Pointer to a is %p\n", p);

    //Assigning 10 to a, by dereferencing the pointer
    *p = 10;
    printf("Value of a is %d\n", a); //10

    //Assigning a to b, by dereferencing the pointer
    int b = *p;
    printf("Value of b is %d\n", b);

    //Assigning 20 to a, by dereferencing the
    // address of a (which is stored in p)
    *&a = 20;
    printf("Value of a is %d\n", a); //20

    //Address of a. Same as p!
    printf("Address of a is %p\n", &*p);
}
```

  During above operations, a and $*p$ are interchangeable. This implies that p is a's address. b has the same value as a, but have different addresses.
- Declaring a pointer doesn't allocate memory (and hence "scanf" to the pointer won't work), until it's assigned with the address of some data that belongs to an appropriate data type.
- Pointer can be used to set or get (by dereferencing it) its content.
- In a 64 bit machine, address is 8 bytes. So, you can use either of the techniques below.

```c
printf("Pointer to a is %p\n", p); //eg. 0061FF18
printf("Pointer to a is %lu\n", (unsigned long)p); //eg. 6422296
```

  Note that *long* datatype is also 8 bytes long, and address must have no sign.
- $Byte$ is the most conventional unit used for addressing, but a $word$ is used for most

practical purposes. It's same as what CPU registers can handle - 32 bit or 64 bit. It's hardware and OS dependent.

- Here is the memory usage of the common data types in C language:
  - *int*: not less than 16 bits, but most commonly fixed at 32 bits (even in 64 bit OS).
  - *char*: not less than 8 bits.
  - *short:* not less than 16 bits
  - *long:* not less than 32 bits
  - *long long:* not less than 64 bits.

C compiler requires that *sizeof(char)* $\leqslant$ *short* $\leqslant$ *int* $\leqslant$ *long* $\leqslant$ *long long*

- Use *char* for handling text data, and use *unsigned char* for handling raw-binary/byte-level data. While the former can handle range -128-127, the latter can handle range 0-255.
- ILP32 architecture uses 4 bytes for int, long and pointer types. LP64 architecture uses 4 bytes for int and 8 bytes for long and pointer types. LLP64 architecture uses 4 bytes for int and long, and 8 bytes for pointer types. Use $\#include\ <stdint.h>$ in your program, to precisely define the storage widths for each type.
- Data spread across multiple bytes can be stored in two different fashions: big-endian and little-endian.
  - Big-endian
    * *Memory Addresses:* Increase from left to right (low memory to high memory).
    * *Storage Order:* The most significant byte (first byte) is stored at the lowest memory address.
    * For example, store the value 0x04D2 in memory as 04 followed by D2
  - Little-endian
    * *Memory Addresses:* Also, increase from left to right (low to high memory).
    * *Storage Order:* The least significant byte (last byte) is stored at the lowest memory address.
    * For example, store the value 0x04D2 in memory as D2 followed by 04
- x86 is little endian. SUN SPARC is Big Endian. ARM, RISC-V has good support for both.
- Alignment of memory addresses is a challenging problem, when data storage is done at addresses that aren't multiples of 4 bits. It'll take some extra time for the compiler to align mis-aligned data storages.
- Memory (RAM) is dividied into two sections called Stack and Heap. Parts of stack are allocated to user programs. When the user program runs, it creates stack frames, which grow downward and could potentially encroach Heap section, when allowed to grow uncontrollably.
- Heap is used to store global variables, static variables and dynamic memory, and occurs at the bottom of the RAM. So, addresses of global variables will be *smaller* than variables inside functions.

Week7 (Arrays)

- $int\ i;\ i\ ==\ *\&i.$ *() is called dereferening operator.
- *Arrays* in C contain homogenous data, unlike in Python.
- In C, *Array* is a convenient shortcut for accessing a contigous set of memory locations. Array definition specifies the start of the list, number of its entries and how to reach the next memory location. An example of an array definition (and initializing it) is

  $float\ D[20]\ =\ \{1,\ 2,\ [5]\ =\ 3.14,\ 7,\ 8.2\}$

  Note that in the above code a *float* array that can contain a max of 20 float values is declared. $D[0]$ is initialized to 1.0, $D[1]$ to 2.0 followed by $D[5]$ initialized to 3.14. Thus, $D[2],\ D[3],\ D[4]$ gets 0 automatically.
- Similarly $int\ A[1000]\ =\ \{10\}$ declares an integer array A with 1000 elements, and initializes all except the first array element to 0. The first array element alone is initialized to 10.
- While declaring an array, note that the number of elements can be a *const* value, but not a *variable*. Though some compilers might support variable number of elements, this syntax is disallowed if the array is also initialized during declaration time.
- In C, array elements start at index 0. In Matlab and Julia, array index starts at 1.
- Theoretically, there's no limit to the number of elements in an array, but it could be limited by the stack size.
- $int\ A[10]$ creates a pointer A. It points to the first memory location in the array $A[]$.
- Pointers can be printed using the format specifier %p, %lx, %lX. Note that the addresses printed using %p don't have to be the actual addresses, as the OS may implement address space layout randomization (ASLR) and other security measures to mask the real addresses for security reasons.
- When *sizeof()* is used on an array, it gives the size of the entire array - the size of of each element multiplied by the number of elements in the array.
- $A[i]\ \equiv\ *(A\ +\ i)$, where A is an array and i represents the array index (offset)
- When *p* is a pointer to integer, *p*++ increments the pointer by 4 bytes. When *p* is a pointer to character, *p*++ increments the pointer by 1 byte.
- C allows to access a memory location outside the bounds of an array. Thus, after declaring $int\ A[10]$, $A[11]$ could be accessible even if the memory location is outside the bounds of array *A*. However, in certain cases, it could result in a crash due to segmentation fault or segmentation violation.
- C doesn't have string, and is represented as an array of characters. Terminated by NULL.
- String declaration and manipulation in C.

```c
//Following 3 lines are equivalent
//all are valid ways to declare a string.
char *s1 = "Hello";
char s2[10] = "Hello";
char s3[] = "Hello";

printf("%s\n", s1); //Hello
printf("%s\n", &s1[2]); //llo
printf("%s\n", s1 + 2); //llo
```

Note that string is essentially a pointer/address, until it encounters a NULL character.

- When the string is declared as an array of characters (s2), a specific index of the array can be changed by assigning it with a new character. However, it will not work if the string is declared as a pointer (s1)
- $sizeof(s1)$ returns 8 in a 64-bit system, $sizeof(s2)$ returns 10 and $sizeof(s3)$ return 6 (including the NULL terminator)
- If the string tries to store more characters than the array size, it could end up holding junk beyond the array size.
- In order to terminate a string at a specific index, modify the character at the index to number 0 (indicating NULL character).
- Note that it's only possible to assign a string to a string pointer or a string array, during its declaration. Thus $char* s;\ s = "hello"$ isn't valid, but $char *s = "hello"$ is. In the first case, $strcpy(s, "hello")$ will fix the issue.
- *strcmp* compares two string arguments and returns 0 if they're same, and returns a positive number, if there's a difference.
- *strcat* will concatenate a given string to an existing string.
- Use *strlen* to find the length of a given string, and includes the NULL terminator. *strnlen* is safer than *strllen, and* accepts an additional parameter for maximum length. *wcslen* is used to handle wide characters (*wchar_t*), each of which take 4 bytes. *wchar_t* are used typically to handle unicode characters.
- Use *strstr* to check if the substr is present in str. $strstr(str, substr)$ returns 1 if *substr* is present as part of *str*.

Week8 (Structs)

- *struct* is used to represent structured records and enables grouping of attributes. When *struct* is declared, the memory for its fields is allocated contiguously. Each field is stored as part of the same block of memory.
- *size of struct* isn't necessarily the sum of individual attribute sizes. This is because of specific alignment requirement of types (eg. integers are stored at addresses that are multiple of 4, doubles at addresses that are multiple of 8 etc).

- When a *struct* is assigned to another (using '=' operator), all attributes of the source *struct* are copied into the destination *struct* as far the individual attribute types match, but doesn't perform deep copy. Pointers inside the source *struct* are copied as such to the destination *struct* and hence both structs could end up pointing to the same memory location.
- Comparing two structs using *'=='* operator is *undefined*. Need to perform attribute-wise comparison.
- *struct* cannot reassigned as a whole, unless when typecast. but its attributes can be individually reassigned.

```c
struct Fruit bananas = {"Banana", 20, 10};

//Assigning later isn't allowed.
// bananas = {"Banana", 20, 10};

//unless, typecast.
bananas = (struct Fruit){"Banana", 30, 20};

//Reassigning individual attributes is also allowed!
bananas.price = 35;
```

- The address of a struct might coincide with the address of its first attribute, but this is not guaranteed.
- *struct* can contain members that are of the same or another struct. In such cases, dot operator can be used to access members of the nested struct. For example, if you have a struct *a* that contains a nested struct *b*, you access a member *c* of *b* with $a.b.c$
- struct that contains a pointer to an instance of the same *struct* is called a recursive struct.
- *typedef* helps define custom (and re-usable) data types as part of a header file.
- *union* is a way to define what possible types can a data represent.

```c
union PI {
    int a;
    float b;
} pi;
pi.b = 3.14159;
printf("Float representation of PI is %f; "
    "Binary representation of PI is %x", pi.b, pi.a);
```

- *union* can lead to inefficient memory usage.
- Size of a *union* is determined by the size of its largest member. This is because a *union* can only hold one of its members at a time, and all members share the same memory location.
- Since all members in a *union* share the same memory location, the last assignment to any member of the union determines the value stored in that memory. The issue with this is, if you attempt to print the first member, you may not get a *meaningful* value.
- *enum* is used to create a new datatype with a fixed set of named values, and each

named value takes a number. This allows to use the *text,* instead of using numbers, thus improving readability of code. *enum* is typically used with switch-case.

```
typedef enum{MON, TUE, WED, THU, FRI, SAT, SUN} days;
days x, y;
x = MON;
y = THU;
printf("x = %d, y = %d\n", x, y);
```

- Though *enum* use text to represent integers, it's stored as integers in memory.
- *enum* type can be assigned with data outside the allowed *enum* range. No checks are done.

Week9 (Dynamic memory)

- Stackframes are space allocated to function calls by C Runtime. This stays as long as the function is in scope, and released when it exits.
- Heap is used to keep global variables in the program, and function static variables.
- *malloc* can be used to request for allocation of memory and it returns a *void pointer* to the chunk of memory. When the malloc fails, it returns a NULL pointer. Before using the memory thus allocated, it must be cast into a pointer of an appropriate data type.
- *malloc* allocates memory in the *heap*, whereas *stack* is used exclusively for the local variables, all of which are pre-allocated memory. Note that strings (say, $char* a = "hello"$) are stored in the area where constant data of the program are stored and is distinctly different from the stack and the heap areas of memory.
- *calloc* is similar to *malloc*, except that it initialize the memory to 0 in addition to allocating necessary memory.
- Creating a huge array in stack, say to hold 10 million integers might result in segmentation fault - core dump, since the stack has limited size. The stack size limit varies by system, but it's usually much smaller than the available heap memory.
- Similarly, allocating a huge amount of memory in heap, say to hold 100 million integers could fail. If memory allocation fails, malloc returns NULL. Even if the memory allocation works, it could fail (get *killed*) while trying to use the memory. This is because the OS overcommits memory, assuming that the program isn't going to use all of the allocated memory.
- Once allocated, the memory is reserved till explicitly freed, or the program is exited.
- Memory allocated can be freed using *free()* by passing it a pointer.
- Freeing parts of the memory that has been allocated in multiple chunks can cause fragmentation and cannot be avoided easily.
- Declaring a pointer to any datatype doesn't necessarily initialize it on its own, but could point to some garbage.
- After freeing an already allocated chunk of memory, you can still access the memory.

However, there's no guarantee of the allocated memory to be available.
- Never free the memory multiple times. Though C Runtime gives a warning, it doesn't prevent you from doing that.
- The easiest and the most efficient way to store multi-dimension arrays is to flatten the array and storing the individual cells of the matrix in a *row-major* format.
- In the context of a 2-dimensional array, $A[i][j] \equiv *(*A + i * C + j)$, where *C* is the number of columns in the matrix.
- In the context of a 3-dimensional array, $A[i][j][k] \equiv *(*A + i * R * C + j * C + k)$, where *R* is the number of rows and *C* is the number of columns in the matrix.

Week10 (File handling)

- DDR = Double Data Rate. RAM = Random Access Memory
- Files are yet another level of abstraction of data storage. These are managed by OS, but through explicit function calls (called system calls). Direct hardware access is not allowed, but must go through OS kernel.
- Files have the following meta data apart from its contents - name, path, permissions, state.
- Files are accessed through its metadata FILE, stored as a STRUCT. All operations with file are done using a pointer to FILE, $FILE*$
- $fopen(<file\_path>, <mode>)$ returns a $FILE*$. It will return a NULL, if the file open didn't succeed.
- When a file pointer with a NULL is closed (using *fclose*), OS will flag a *segmentation fault (core dumped)* error.
- STDIN, STDOUT, STDERR are special files.
- APIs or system calls available for text manipulation in files - *fgetc, fputc, fgets, fputs, fscanf, fprintf.*
- *fgets()* from *STDIN* is equivalent to *gets().* Similarly, *fputs()* to *STDOUT* is equivalent to *puts().* However, note that *fgetc()* from *STDIN* is NOT equivalent to *getc()*
- Also, *fprintf* to *STDOUT* is equivalent to *printf().* Simlarly, *fscanf* from *STDIN* is equivalent to *scanf().*
- *fgets()* will read specified number of bytes (2nd parameter) from the file pointer (in 3rd parameter) into a buffer (in the 1st parameter). Note that it includes the NULL terminator also, so technically 1 less character is read into the buffer.
- *fscanf()* don't necessarily read an entire line from a file, but reads formatted input from a file, until it encounters a whitespace character. Thus, essentially, use *fscanf()* to read a word at a time, and use *fgets()* to read the entire line.
- *fscanf()* keeps track of an index until which it has read. Thus, if you use *fgets()* after the initial *fscanf(),* it'll continue only from where the *fscanf()* left off and not from the beginning of the file.
- When a file is opened in *write* mode, it gets overwritten. When a file is opened in

append mode, it gets appended to.

- *fputc* and *fputs* prints character/string to the specified file. *fprintf* prints data in the specified format to the file. All functions write data immediately following the current byte.
- Using *fprintf* with STDOUT or STDERR prints given string to the console. The difference is that in the first case, the result can redirected to a text file using > symbol during execution, but will continue to print in the console in the second case. If STDERR also needs to be redirected, use 2 > instead of a simple >
- Binary mode of manipulating files work with bytes directly, instead of text
- *Use fread()* and *fwrite()* for binary mode manipulation of files. Due to the multiple endianness supported in various systems, and to ensure compatibility between them, it's recommended to use "protocol buffers" that will provide serialization.
- Use *wb* or *rb* to mode to write/read in binary.
- Use *hd* tool in Linux to see the bytes in the file. Can be used with a binary or a text file.
- & is a bitwise AND, | is a bitwise OR, ^ is a bitwise XOR, ~ is a bitwise NOT (negation)
- Bitwise masks can be used to check the status of a specific bit.
- Use sprintf to convert an any datatype to a null terminated string.

```c
int main() {
    char buffer[100];
    int number = 42;

    // Use sprintf to format a string
    sprintf(buffer, "The number is: %d", number);

    // Print the formatted string
    printf("%s\n", buffer);

    return 0;
}
```

- Following are some sample constructs

1. $FILE* fp = fopen("input.txt", "r");$
   $if (fp != NULL) \{...\}$

2. $while (fgets(line, sizeof(line), fp) != NULL) \{...\}$ $// read line off a file$

3. $fputs("Hello, world!", fp);$ $// Writes a string$

4. $char *pos = line; while ((pos = strstr(pos, str)) != NULL) \{pos += strlen(str);\}$

5. $while ((c = fgetc(fp)) != EOF) \{...\}$

6. $fputc(c, fp);$

7. $while\ (fscanf(fp,\ "\%d",\ \&var)\ ==\ 1)\ \{...\}\ //Compare\ with\ number\ of\ items\ read$

8. $fprintf(fp,\ "\%d\backslash n",\ var);\ //with\ 3\ parameters$

9. $fprintf(fp,\ "/\backslash n");\ //with\ 2\ parameters$

10. $fclose(fp);$

11. $//read\ words\ off\ a\ string$
$char\ *pos\ =\ line;\ while\ (sscanf(line,\ "\%s",\ \&word)\ ==\ 1);\{pos\ +\ =\ strlen(str);\}$

12. $sprintf(query,\ "SELECT\ *\ FROM\ users\ WHERE\ id\ =\ \%d",\ id);$

- Following are modes in which a text file can be opened.
  - r: read text file
  - w: write text file
  - a: append text file
  - r+: read and write text file
  - w+: read and write text file
  - a+: read and append text file
- Following are modes in which a binary file can be opened.
  - rb: read binary file
  - wb: write binary file
  - ab: append binary file
  - rb+: read and write binary file
  - wb+: read and write binary file
  - ab+: read and append binary file

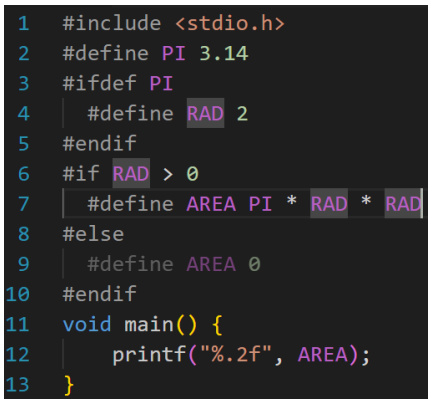
Week11 (Macros)

- Most times, macros use upper-case alphabets. macros support arguments. For example, the following is a valid macro.

$$\#define\ ABS(x)\ x\ <\ 0\ ?\ -x\ :\ x$$

- $\#define$ is used to replace specified text with *another_text.* Note that the replacement is performed during compile time. This is not done by the compiler, but by another program called pre-processor.
- $\#define\ flag$ is typically used to simplify the compilation process, by removing lines that are defined under $\#ifdef\ flag$.
- $\#if\ defined$ can also be used instead of $\#ifdef$

- Sample code to demonstrate *#define*, *#ifdef* and *#if*

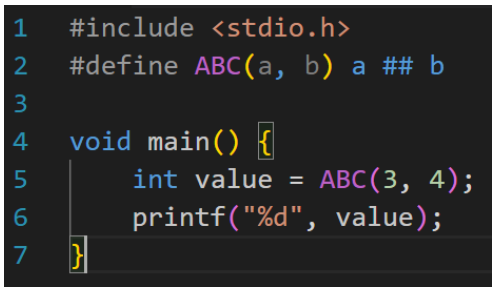


```
#include <stdio.h>
#define PI 3.14
#ifdef PI
  #define RAD 2
#endif
#if RAD > 0
  #define AREA PI * RAD * RAD
#else
  #define AREA 0
#endif
void main() {
    printf("%.2f", AREA);
}
```

- $\#if\ !defined$ negates $\#if\ defined$
- $\#undef$ undefines the $\#define$.
- Use brackets, if you have $\#define$ with arguments. This is required to take care of the precedence. Thus, write $\#define\ SQR(x)\ ((x) * (x))$
- If I use a # notation inside a $\#define$ body, it expands to the parameter name. Thus,

$$\begin{array}{l} \#define\ PRINTALL(x) \quad \backslash \\ \{ \quad \backslash \\ printf("Name\ of\ the\ variable:\ \%s",\ \#x); \quad \backslash \\ \} \end{array}$$

  In the above example, $\#x$ in the 3rd line expands to A, when $PRINTALL(A)$ is called. Also, note the use \ symbol at the end of each line, except the last. This is used to indicate *continuation* of code.
- Using ## with define body implies concatenation. Thus, following code will produce 34 as its output.



```
#include <stdio.h>
#define ABC(a, b) a ## b

void main() {
    int value = ABC(3, 4);
    printf("%d", value);
}
```

- $\_\_func\_\_$, $\_\_FILE\_\_$, $\_\_LINE\_\_$, $\_\_DATE\_\_$, $\_\_TIME\_\_$, $\_\_STDC\_VERSION\_\_$ are a few other macros available in a C program.
- $\#include$ can work recusively. Thus, if the first file includes another file, which includes yet another in turn, the preprocess *cpp* accumulates all contents until there're no more inclusions.
- To include functions defined in *file2* into the main program in *file1*, either

  - use a function signature on top of file1.
  - #include "file2.h" on top of file1, where *file2.h* contains the function signature.

NOTE: *file2.h* will contain the following lines to avoid recursive inclusions.

```
#ifndef FILE2_H
#define FILE2_H
```

- gcc -c file.c produces *file.o*. If there are multiple .c files - *file.c, file2.c, main.c* - that must be combined together to create the .exe file, use
  gcc file.o file2.o main.o -o main.exe
- Use *make* to do incremental compilation of files - it helps track changes (detect if the files have been changed recently) and respect dependencies.
- Following are the variable storage classes used in C.
  - auto: by default, all variables in functions have function scope. Contains garbage value, if uninitialized
  - extern: global variables, accessible across multiple files. Initialized to 0, automatically.
  - dynamic: created using *malloc, calloc or realoc*
  - static: retains value across function calls. Initialized to 0, automatically.
  - register: Used for variables which are accessed very frequently. If no register is available, the variable is stored in the memory, and it behaves the same as “auto” storage class.