-Can't operate 'mod' on floats.
-While comparing characters, the ASCII values are considered.
-Character, integer, and float can be compared.
-Use parenthesis for compound operators.
-Bracket priority followed, nested logic. (An int assigned two true booleans will have value 11.)
-Precedence => which one first.
Associativity => evaluation of the same precedence.
(CHECK THE ORDER IN SLIDES)
-Sequence point example => ;
-Ambiguity before sequence point makes the statement 'ambiguous'.
After a sequence point is executed then the preceding logic isn't affected.
which side of '+' would be executed while calling functions is an example of ambiguity.
(^ f1() + f2()) (^ ++x + x++)
Do not change the same variable in the same line before the sequence point.
(^ ++x + y++ is fine.)
error message -> warning: multiple unsequenced modifications.
ambiguity doesnt break the program. the code continues to execute.
The intention of the warning is to highlight ambiguity and the fact that the output might vary from compiler to compiler.
Use brackets to avoid ambiguity.
-<type_name> <variable> changes the datatype of the variable temporarily.
-If a float is assigned to an integer, the decimals are removed, not rounded off.
This is an ambiguity.
-Scope of an element is within the curly brackets.
-using "if" without a bracket will attach it to the immediate next logic statement (ONLY 1).
using "else" without brackets will attach it to the latest unattached "if".
-switch-case is only for integers and characters, not float.
-default statement in switch-case is executed when no case matches.
-every case is checked in switch-case, irrespective of the match being found, unlike if-else.
-if one case is satisfied, every case below it will be executed if "break" is not used.
-"break" will jump the execution to the point below the scope of the switch-case block.
-the position of "default" does not matter. But once the default is executed, everything below it will be executed provided "break" is missing.
-mathematical operations are costlier than comparing cases. So don't do that for optimization.
-(GCC COMPILER SPECIFIC) case 'a' ... 'z': will compress all the comparisons a to z. This is a range operator

-When the limit of an integer is exceeded, a seemingly random value is assumed by the variable.
Int range: -2147483647 to 2147483647
The seemingly random value is actually the variable assuming the value from the lowest possible limits again.
-Remember to update the iterator.
-"continue" skips the statements below it in the scope of the loop and either goes to the start of the loop again or jumps out of the loop based on the given condition of the loop.
-"break" will jump out of the loop and never start the loop again.
-without brackets, the "else" will associate with the last unattended "if" no matter how far it is.
-if a semicolon is present after the "if" statement, the if block will be considered as empty and a warning will be thrown.
-"for" loop takes three arguments, an initial condition, an condition, an updating condition. If either or all are absent, it won't be an error, but the loop will consider the last value of the variables and if the condition is missing, the loop will run till infinite.
-declare the iterator variable of "for" loop outside the loop to use it beyond the scope of the loop.
-arrays store data in memory in contiguous manner but any random element of an array can be accessed by the index which starts at 0 and goes till n-1 for n elements in the array.
{{ Continuous is time domain and contiguous is in space domain. }}
-the distance between the two array cells depends on the size of the datatype, the byte space. 1 byte = 8 bits.
char = 1 byte
Int = 4 byte
Float = 4 byte
-unallocated integers are taken as zeros.
-integer arrays are initialized as all zeros. (need to declare curly braces, else garbage values)
-char arrays are initialized with '' (need to declare curly braces, else garbage values)
-segmentation point- addressing a memory space beyond the declared ones. Happens when declared size of array is less than the assigned list. (very important, read more online)
-if array is initialized without the size, the compiler will assume the length of the array by itself based on the assigned array.
-to traverse the array if the size is unknown, use 'sizeof(arr)' to find the length of the array. The same can be used to get the size of any object in the memory.
-arrays are not assignable, must be declared directly, or must be assigned per index later.

cannot assign one array to another directly, need to be done index wise.
-'typedef' works like renaming the existing datatypes.
syntax: typedef <existing_type> <type_alias>;
typedef can also be used to define array types, alias should have the size of the array, however while callng it again, the user need not type the size.
-2D array can be defined by using typedef twice. use the alias of the previous typedef as the existing type for the second alias then the second alias will behave like a 2d array.
A[m][n] means an nth element of mth array, n is defined first then m.
can be scaled up to an n-dimensional array.
[!]-when taking an integer and character in one line, add a space before %c to avoid bugs.
-if using a variable-length array, assign a value to the length variable before calling it while declaring the array.
-Null value in the int array is 0 and in the char array it's an empty character.
-To print '%', use printf("%%")
-There can be two variables in the memory with the same name but different values because of the scope they are declared in.
-Integer division yields an integer.
-while calling functions with lists as parameters, the memory address of each element is passed, not the value of the elements. Therefore the changes made to he list will be auto updated in the funciton.
we dont need a return value if we update the passed list. (known as call-by reference)
-"sizeof()" will return the memory used in storing the element.
sizeof(array) will return (length of array)*(size of datatype)
-in a function, the return statement wont return anything until there exist a sensible value to return, which is in the form of the specified datatype of the function.
This, when used in recursion, a function call is present in the return statement, so here the return wont execute but the function will be called and it will keep calling the funciton recursively until it gets a definite value out of it, which is when it hits the base statement.
-Hexadecimal values has 16 digits for its representation, however, ABCDEF are used to represent digits beyond 9.
-CONVERTING DECIMAL TO BASE: Base expansion is the reverse of the list of remainders obtained by repeatedly dividing the given number by the desired base number.
-CONVERTING BASE TO DECIMAL: Example- (101101)2 = (1*2^0) + (0*2^1)+(1*2^2)+(1*2^3)+(0*2^4)+(1*2^5) = 1+0+4+8+0+32 = 45 = (45)10
(digit*base^positionFromRight)
(Refer slides for a shortcut method)
-to internally convert into decimal,octa,hexa use %d,%o,%x respectively.

-if an integer starts with 0, the compiler stores it as octa form. If an integer starts with 0x, the compiler stores it as hexadecimal form.
-case irrelevant a,b,c,d,e,f can be used to while declaring a hexadecimal in int variable.
-no way to convert the integer in binary in printf.
-bits in chip = electric charge = 0,1
Bits in hard drive = north/south magnetism = 0,1
-8 bits together male a byte, n bits yield 2^n patterns.
-ASCII: First bit (usually 0) is used as 'parity' bit which is used for error checking. This leaves us with 7 bits to use instead of 8, thats why total storeable values in ASCII code are 128 instead of being 256. This can be specifically mentioned not to do and then its called 'extended ascii' which stores 256 values.
-bits are read from right to left.
-32 bit operating system have storage size of 2 bytes for int while 64 bit os has 4 bytes.
-unsigned numerical data types start from 0. Negatives are not included. Use "%u" for unsigned.
-long double is 4 times float = 16
-single precision floating point number: example: 1.423E2
(sign)(exponent).(Mantissa)E(Exponent)
sign=1 bit, exponent=8 bits, mantissa=23 bits
-double precision floating point number:
sign =1 bit, exponent=11 bits, mantissa=52 bits
-CONVERT DECIMAL TO SINGLE PRECISION BINARY: (0.125)10 = (001)2
0.125*2=0.25 → 0
0.25*2 = 0.5 → 0
0.5*2 = 1.0 → 1
0.0*2 = 0
-exponent bias → exp = 8bits = 00000000 = -127
Therefore while storing the exponent, add by 127.
Rule by IEEE745 to get rid of the necessity of sign-in exponent.
-CONVERT SINGLE PRECISION TO DECIMAL:
Formula; value = (-1)^sign * 2^(E-127) * (1 + sum[i=1 to 23](b(23-i) * 2^-i))
……
-Declaring pointer variable: space is irrelevant.
(data_type) *(name);
(data_type)* (name);
(data_type) * (name);
-64 bits os → each memory address is of 64 bits. → 8 bytes (1 byte = 8 bits)
-pointer values are given by &
Example: int *ip = &i;

& → ‘address of’
-dereferencing: getting the value stored at the memory address pointed to by the pointer.
’*’ is used before pointer type variable for dereferncing.
-’%p’ is used to print pointers.
-adding pointer by any integer will add its memory address value by *integer*sizeOfDatatypeOfPointer*
-pointer arithematics only allows addition and subtraction.
addr(pointer + i) = addr(pointer) + (size of datatype i)
changes the adress value stored in pointer variable and not the adress of the pointer itself.
-== != <= >= are valid for pointer comparison provided the pointers are of same type.
-pointers are assignable
-*ptr = 0 will assign 0 to the variable whos adress is pointed by the pointer
-pointers to an array are not assignable to other array pointers.
-structure syntax:
creation:

```
struct example {
        int x;
        char y;
}; <-- notice the semicolon
```

declaration:

```
struct example x1, x2;
```

another way of declaration:

```
struct example {
        int x;
        char y;
} x1, x2;
```

state:

```
x1 = [x:_, y:_]
x2 = [x:_, y:_]
```

(kind of like class)
-assigning values: x1.x = integer value, x1.y = char value
-structures can be assigned
-sizeof(x1) will return sum of sizes of all variable types declared in structure of x1.
padding is done by the compiler, even for a character which is 1 byte long, 4 bytes are reserved and the next type is stored after 4 bytes.
Padding happens when char is present with int or any other datatype.
-when declaring function without definition, name of variables can be omitted and only entering datatypes will work.

however, while defining the function, the names must be included.
-#define <name> = value; is used to define constants.
-if arrays are present in an structure and you copy a structure to another one, the arrays also get copied.
A CASE WHERE ARRAYS GET COPIED WITHOUT 'ARRAYS ARE NOT ASSIGNABLE' ERROR
-aliasing structures:
typedef struct <structure name> <alias name>
using alias:
<alias name> <new structure name>
-it is possible to have an array of structures.
syntax of declaration: struct <structure name> <list name>[size];
-to pass an 2d array as a parameter in a function, while declaring the function, keep the lengths of dimensions before the array and use the length variables inside the array.
Example: int set(int a, int b, int arr[a][b]){}
-use structure to return multiple values from a function.
-yp->a is simplification of (*yp).a where yp is the pointer to structure
-<datatype> *arrayPointer = array; is the pointer to the first element of the array.
since the array is stored in contiguous manner in the memory, only gaining the address of the first element is enough to get acess to entire array.
-**pointer is used to dereference a pointer twice. Used for pointer to a pointer.
can be scaled up to more layers of pointers.
-cannot declare variable values in structure.
-dereferencing a 2d array pointer will return the address of the first array.
dereferencing twice will give the first element of first array.
-when a variable is declared in the scope of a function and returned, it results in a warning since the scope of the variable returned is only in that function and cannot exist outside it.
a garbage or unknown value will be assumed upon returning.
-a function with pointer datatype will return the address of the variable returned.
-dereferencing a 2d array will give the address of the first 'row', which is also the adress of the first element of that row.
both the adresses will be same just their type will be different.
-in a 2d array if a pointer is pointing to a row (array) then (pointer+1) will point to the next row (array) and not the next element of the same row.
-list is also of the type list*, an array of pointers.
-char c[4] = "ABC"; will store the string as {A,B,C,\0}, size in memory = (size of string +1).
\0 is a null character.
-strlen(string) will return the true size of the string, that is, without \0.

-strcpy(string2,string1) will copy the first string to second. It overwrites.
-strcat(string1, string2) will concatenate string2 to string1 and store it in string1.
-strcmp(string1, string2) will compare the two given strings based on the ASCII values of the characters.
-all string operations are present in string library
-"%s" is the place holder for string
-to take string input:
char str[30];
scanf("%s", name); <-- notice that no ampersent for name.
everything after a blank space will be ignored.
-to take the string input with blank spaces:
fgets(string, sizeof(string), stdin);
string[sizeof(string)-1]='\0'; <-- to replace the \n with \0 at the end of the string. when we press enter, fgets takes \n.
-fputs(string,stdout); prints the string.
-fflush(stdin); is used to clear the input buffer.
This will clear the input buffer for other programs too that may be running correspondingly.
-while(getchar()!='\n'); is a better approach to clear the buffer.
-tolower(char); changes to lowercase character.
-
-malloc(sizeof(datatype)); is a void type pointer which points at empty memory space of required size.
We need to typecast it itno the required datatype.
(Datatype *) malloc(sizeof (datatype));
-free(variable); will free the memory space used by the variable.
-strcmp(string2,string1); will return the difference of first non similar ASCII values.
-malloc has a global scope by default
-malloc is present in stdlib.h
-memory allocated through malloc needs to be freed explicitly.
-freeing a malloc doesnt deletes the memory, it just unallocates it, which means it can still be accessed later with some chances of getting a segmentation error. To avoid this, the malloc pointer variable needs to be set to NULL.
-segmentation fault: "pointing to something that is not yours".
-fputs(string,stdout); prints the string.
-fflush(stdin); is used to clear the input buffer.
This will clear the input buffer for other programs too that may be running correspondingly.
-while(getchar()!='\n'); is a better approach to clear the buffer.
-tolower(char); changes to lowercase character.

-malloc(sizeof(datatype)); is a void type pointer which points at empty memory space of required size.
We need to typecast it itno the required datatype.
(Datatype *) malloc(sizeof (datatype));
-free(variable); will free the memory space used by the variable.
-strcmp(string2,string1); will return the difference of first non similar ASCII values.
-DataType* var1, var2; gives an error.
DataType* var1, *var2; is the correct method

-Files:
FILE* fp;
fp= fopen("test.txt","w");
fputs("string",fp);
fclose(fp);
fgets(list, max_size, fp);

-strcmp(str1,str2); will return char1-char2 of first non equal character in both strings.
-puts(string); adds an extra '\n' at the end.
-tolower(c) and toupper(c) accepts and returns int type. They are present in ctype library.
-radian = degree*(3.14/180.0);
-sin, cos, tan take values in radians.
-format specifier for ciel() and floor() is "%lf"
-malloc and free are present in stdlib library
-use malloc and pointer type functions to declare variables in a function and be able to return them.
Example:

```
int* g(void)
{
  int *x;
  x = (int *) malloc(sizeof(int));
  *x = 20;
  printf("x = %p, *x = %d\n", x, *x);
  return x;
}
```

-fgets takes input till it encounters '\0' that is, end of the string or max length specified.
-example of linked lists:

```
void enterRecord()
{
  RecordPtr user;
```

```
  user = (RecordPtr) malloc(sizeof(Record));
  printf("Read name and id\n");
  scanf("%s",user->name);
  scanf("%d",&(user->id));
  user->next = 0;

  if (tableStart == 0)
  {
    tableStart = user;
    tableEnd = user;
  }
  else
  {
    tableEnd->next = user;
    tableEnd = user;
  }
}
```

-while reading from a file, the size of characters being read in fgets should be > 1.
-file opening modes:
r - read only
r+ - read and write, return NULL if the file doesnt exist.
w - write
w+ - deletes all the contents of the existing file and creates a new file if not present
-fseek(filePointer, int offset, int position)
to move the cursor by an offset
from the position defined by the following constants:
SEEK_SET = Beginning of the file
SEEK_CUR = Current Position of the file pointer
SEEK_END = End of FILE
-after using fseek, remember to return back to the start of the file before printing.
-fprintf is used to prinft the output in file instead of stdout
Syntax:
fprintf(file_pointer, "%format", variable);
-fscanf is used to scan the input from a file
Syntax:
fscanf(file_pointer, “%format”, variable_address);
This returns EOF when end of file is hit. Can be used in while loop as fscanf(...) != EOF
-a%-b is same as a%b