Section 4

All notes for the tutorial will be here

Makefiles

Makefiles are tools used for automating the building task. They have variables called **Macros**

A macro can be initialized as

```
CC = gcc
```

conventionaly CC is the program to be executed. We can define any value as a macro

```
CC = gcc
CFLAGS = " -v "
```

Then we have something called a process. A process can be defined with the files that are being affected.

```
main: main.c function.c
    $(CC) $? $(CFLAGS) -o $@.o
```

Tha make program will just fill in the macros and execute any command. The Macros will be distributed as:

- `$(CC)` and `$(CFLAGS)` will split out the values of the variables
- `$?` will give the changed filenames.
- `$@` will give the current process name in this case *main*

Variable Scopes

Global variables

Global variables are defined in the top of a file and outside of any function

```
#include <stdio.h>


int multip = 10; // This is a global variable

void printmult(){
    printf("The Multiplier is %d\n",multip);
}

```

```
int main(int argc, char const *argv[])
{
    printf("%d\n",multip);
    printmult();
    return 0;
}
```

Here all the functions inside the file can access the variable

External variables

External variables are variables which are declared in a file and can be accessed across any file in the project.

This is the *fun.c* file

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

extern int myextern;  // The external variable is defined here

void getmyextern(){
    printf("%d\n",myextern);
}
```

The *main.c* file is:

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

int myextern = 10; // This is the initialisation

int main(int argc, char const *argv[])
{
    printf("%d\n",myextern); // Output: 10
    getmyextern(); // Output: 10
    return 0;
}
```

Header file uses

- A program can be divided into individual files and that can be maintained easyly.
- Macros , functions , enums and structers are all can be defined in a header file eg:

```
extern int hello;
int myfunc(int a);
```

```
void getmyextern();
```

Memory types

Stack memory

- Memory that is used inside a function.
- Locally available so easy to track.
- Uses LIFO (last in first out) method to save variables.
- Allocated and freed automatically.

Heap memory

- Have a hierarchial structure.
- A larg pool which is used dynamicaly
- Managed by programer (not automatic)
- Accessed using pointers
- Created using `malloc()`
- Deleted using `free()`
- Restriction is only the physical size of memory.
- Global
- Slower

How to selct

Stack	Heap
For small memory	When need larger memory
When only need to persist when the function is alive	When to keep the data for a long time
Easier and faster	When to allocate data dynamically (arrays,struct)

Section 5

Storage classes

classified with respect to it's scope,visiblity and lifetime.

Auto variable

- The lifetime is defined automatically.
- A variable in a function or loop is defaultly automatic.
- Because the default `auto` keyword is not used that much.

eg:

```c
#include <stdio.h>

int main(){
    for (int i = 0;i<=10;i++){
        auto int age = 18;
        printf("%d",age);
    }
    // age is destroyed when exited the loop
}
```

Extern variable

- The lifetime is as the program exits.
- Can initialize an external variable with a legal value where it declared.
- Can be accessed within files.
- every fle must reinitialize them to use. But the value will be same everywhere.

fun.c file:

```c
#include "functions.h"
#include <stdio.h>

extern int myextern;  // The extern varable is defined

void getmyextern(){
    printf("%d\n",myextern); // can be accessed in the same file
}
```

main.c file:

```c
#include <stdio.h>
#include "functions.h"


int myextern = 10;  // The variable is initialized.


int main(int argc, char const *argv[])
{
    printf("%d\n",myextern); // can be used in the same file
    getmyextern();
    return 0;
}
```

When using arrays we dont have to give the dimention of the size while defining an extern variable.

fun.c file:

```
#include <stdio.h>

extern char password[];
void printpass(void){
    printf("%s\n",password);
}
```

main.c file:

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

char password[30] = "12345";
void printpass(void);

int main(int argc, char const *argv[])
{
    printf("%s\n",password);
    printpass();
    return 0;
}
```

for multi-dimentional arrays we have to specify the second size of the array.

```
extern int data[][10];
```

This denotes a multi dimentional array with 10 fixed columns and as many rows as we initialize.

We can make functions also

Section 6

Advanced Datatypes

Variable length arrays

A variable length array have the length of an expression executed at the runtime. It doesn't mean you can modify that after we create. Just means that we can give a variable at the position of the size f the array.

```
#include <stdio.h>

int main(){
    int size;
    scanf("%d",&size);
    int arr[size];
    printf("array of %d elements created.",size);
}
```

Flexible array members

Flexible array membes is a technique where we can specify the length of the array in runtime and also we can modify it. Here is an example code.

```
#include <stdio.h>
#include <stdlib.h>
struct flexarr
{
    int arraysize;
    int array[];
};

int main(int argc, char const *argv[])
{
    int desiredSize = 5;
    struct flexarr * ptr;
    ptr = malloc(sizeof(struct flexarr) + desiredSize + sizeof(int));
    return 0;
}
```

Complex Numbers

A number in the form of *a+ib*

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

int main(int argc, char const *argv[])
{
    double complex var = 12+10*I;
    printf("Real part : %f\nImaginary part : %f",creal(var),cimag(var));

    double complex sum = var + var;

    printf("\nSum is : %f+%fi",creal(sum),cimag(sum));

    return 0;
}
```

Designated Initialization

Designated initializers are used to initialize only some certain members of the array. For example:

```
int arr[6] = {[0]=100,[4]=20};
// Output: 100 0 0 0 20 0
```

This initializes only 0th position and 4th position. All other elements are set to be zero.

Also we can specify a range of numbers to initialize.

```
int arr2[22] = {[0 ... 9] = 1,[10 ... 20] = 11,[21]=111};
// Output: 1 1 1 1 1 1 1 1 1 1 11 11 11 11 11 11 11 11 11 11 11 111
```

Similiar will work in struct also.

```
struct person{
    char * name;
    int age;
};

int main(){
    struct person p1 = {.name = "Vijay" , .age = 20};
}
```

What you know

What is the output of the following program?#define NUMBER 55 #define NUMBER2 20 int main() { int i = NUMBER > NUMBER2; printf("%d", i); }

What will be the output of the following program?#include<stdio.h> #define x 4 int main() { int y; y= *x*x*x; printf("%d",y); return 0; }

Write a preprocessor directive to accomplish each of the following:Define symbolic constant YES to have the value 1

The keyword typedef is used to define a new data type (T/F)

What will be the output of the following program?typedef int integer; int main() { int i = 22, *ptr; float f = 33; integer j = i; ptr = &j; printf("%d\n", *ptr); return 0; }

We want to declare x, y and z as pointers of type int. The alias name given is: int_ptrThe correct way to do this using the keyword typedef is:

One of the major differences between typedef and #define is that a typedef interpretation is performed by the _________________ where as a #define interpretation is performed by the _____________.

Declare an array of 100 ints and initialize it so that the last element is -1

An array whose length is defined in terms of a value determined at execution time is a __________.

A variable-length array can change in size during its lifetime. (T/F)

If you include the <complex.h> header, you can use the complex type (T/F)

What you should review

What will be the output of the following program?#include<stdio.h> #define int char int main() { int x=8; printf ("sizeof (x) =%d", sizeof (x)); }

__________ allow us to initialize elements of an array explicitly by using a subscript.

Declare an array of 100 ints and initialize it so that elements 5, 10, 11, 12, and 3 are 101 Continue

Section 7

Type qualifiers

Type qualifiers can be used to give more information about the variable to the compiler. Examples are const,volatile,restrict

Const

const keyword represents a constant variable once it is initialized with a value it cannot be changed. Trying to do so will produce an error.

```
#include <stdio.h>

int main(int argc, char const *argv[])
{
    const double pi = 3.14;
    const int nums[2] = {1,2};
    typedef const int myvarname;
    const myvarname var = 10;
    const int * ptr;
    return 0;
}
```

Volatile

volatile keyword represents that this variable will be changed during runtime so the compiler will supress various optimizations against reduntant assignments. Also prevents caching of variables.

Chosable when:

- Programs that have a lot of threading
- Programs where resources are scarce

```c
#include <stdio.h>

int main(int argc, char const *argv[])
{
    volatile double loc1;
    volatile double *ptr1;
    volatile int clock;
    return 0;
}
```

Restrict

Used as an optimization hint for type qualifiers. In case of pointers, restrict tells to compiler that this pointer is not going to be changed throughout it's lifetime and it is the sole object to save the data. This cancels all the other checks. Also there will be only that pointer which access a specific data. Anyway the compiler can chose to ignore it.

```c
#include <stdio.h>

int main(int argc, char const *argv[])
{
    int * restrict ptr2;
    int * restrict ptr3;
    // Cannot have same address
    return 0;
}
```

What you know

- What will be the output of the following program? #include <stdio.h> void main() { int k = 4; int *const p = &k; int r = 3; p = &r; printf("%d", p); }
- What will be the output of the following program? #include <stdio.h> void main() { int const k = 5; k++; printf("k is %d", k); }
- One of the main use cases for using the volatile type qualifier is to ensure that global variables accessed by multiple tasks within a multi-threaded application are not optimized by the compiler
- Can a pointer be volatile? Explain your answer.
- The restrict type qualifier can be applied to all data types (T/F)
- The restrict type qualifier is not supported by C++ (T/F)

What you should review

- Which of the following statements is false?
- Can a parameter be both const and volatile? Explain your answer.

- The restrict type qualifier is an optimization hint for the compiler that is must follow (T/F)

Section 8

Bit Manipulation

bits for the basic datatypes in c are

| bool | 1 | | --- |