Agenda of This Session

- Preprocessor Directives, I/O functions
- File I/O, Multiple file compilation, Header files
- Dynamic Memory Management, Error Handling
- GNU GCC compiler, creating libraries, Make files
- Simple project

Preprocessor Directives

- The preprocessor provides the ability for the inclusion of header files, macro expansions, conditional compilation, and line control.

- Including files

#include <stdio.h>

#include "stdio.h".

<...>-file is searched for in the standard compiler include paths

"..."-search path is expanded to include the current source directory

#include often compels the use of #include guards or #pragma once to prevent double inclusion

Conditional compilation

The if-else directives #if, #ifdef, #ifndef, #else, #elif and #endif can be used for conditional compilation

“#ifdef” directive checks whether particular macro is defined or not.

Conditional compilation cont'd

"#ifdef" directive checks whether particular macro is defined or not. If it is defined, "If" clause statements are included in source file.

```
#include <stdio.h>
#define RAJU 100
int main()
{
  #ifdef RAJU
  printf("RAJU is defined. So, this line will be added in " \
         "this C file\n");
  #else
  printf("RAJU is not defined\n");
  #endif
  return 0;
}
```

O/P: RAJU is defined. So, this line will be added in this C file

#ifndef exactly acts as reverse as #ifdef directive. If particular macro is not defined, "If" clause statements are included in source file

```
#include <stdio.h>
#define RAJU 100
int main()
{
  #ifndef SELVA
  {
    printf("SELVA is not defined. So, now we are going to " \
           "define here\n");
    #define SELVA 300
  }
  #else
  printf("SELVA is already defined in the program");
 #endif
  return 0;
 }
```

O/P: SELVA is not defined. So, now we are going to define here

Preprocessor Directives cont’d

“If” clause statement is included in source file if given condition is true.

```
#include <stdio.h>
#define a 100
int main()
{
  #if (a==100)
  printf("This line will be added in this C file
since " \
        "a \= 100\n");
  #else
  printf("This line will be added in this C file
since " \
        "a is not equal to 100\n");
  #endif
  return 0;
}
```

O/P: This line will be added in this C file since a = 100

- Macro definition and expansion

```
#define <identifier> <replacement token list>
// object-like macro
```

```
#define <identifier>(<parameter list>)
<replacement token list>  // function-like
macro, note parameters
```

```
#undef <identifier>
// delete the macro
```

```
#define PI 3.14159
```

```
#define RADTODEG(x) ((x) * 57.29578)
```

RADTODEG(r + 1) expands correctly as ((r + 1) * 57.29578)

Preprocessor Directives cont’d

- Special macros and directives

__FILE__ and __LINE__, expand into the current file and line number,can be manipulated with the #line directive

- Example:

```
#line 314 "pi.c"
printf("line=%d file=%s\n", __LINE__,
__FILE__);
```

generates the printf function:

```
printf("line=%d file=%s\n", 314, "pi.c");
```

- Token stringification

# – "Stringification Operator" converts a token into a string

- Example:

```
#define str(s) #s

str(p = "foo\n";) // outputs "p = \"foo\\n\";"
str(\n)           // outputs "\n"
```

Preprocessor Directives cont'd

- User-defined compilation errors

The #error directive outputs a message through the error stream.

#error "error message"

Files

- Five major operation:
  - Create a new file
  - Open an existing file
  - Read from a file
  - Write to a file
  - Close file

- Steps for processing a file
  - Declare a file pointer variable

    FILE *fp;
  - Open a file using fopen() function

    fp = fopen(filename, mode);
  - Process the file using suitable function

    fprintf(), fscanf(), fputc, and fgetc
  - Close the file using fclose() function

    fclose(fp);

File modes

Mode	Description
r	Opens an existing text file for reading
w	Opens a text file for writing. If it does not exist, create a new file
a	Opens a text file for writing in appending mode. If it does not exist, create a new file
r+	Opens a text file for both reading and writing.
w+	Opens a text file for both reading and writing(overwrites the file)
a+	Opens a text file for both reading and writing(in append mode)

```
#include <stdio.h>

main() {
   FILE *fp;

   fp = fopen("/tmp/test.txt", "w+");
   fprintf(fp, "This is testing for fprintf...\n");
   fputs("This is testing for fputs...\n", fp);
   fclose(fp);
}
```

File I/O: Standard input: fscanf()

- Syntax:

int fscanf(FILE *f, char *format, arg1, arg2, ...);

- Eg:

int x;

double d;

char c, array[MAX];

FILE *infile;

❖fscanf(infile, "%d,%c", &x, &c);

// read an int & char from file where int and char are separated by a comma

❖fscanf(infile,"%s", array);

// read a string from file into array stops at white space

❖fscanf(infile, "%lf %24s", &d, array);

// read a double and a string upto 24 chars from infile

❖fscanf(infile, "%20[012345]",array);

// read a string of at most 20 chars consisting of only chars in set

❖fscanf(infile, "%[^.,:!;]", array);

// read in a string, stop when hit punctuation mark

❖fscanf(infile, "%ld %d%c", &x, &b, &c);

// read in two integer values store first in long, second in int read in end of line char into c

Note: fscanf returns the number of items read (above example would return 3) or it returns EOF if it reaches EOF while reading

Standard input : fgets(), fgetc(), fgetchar()

- Syntax:

char* fgets(char* str,int buffer,FILE* stream);

Eg:

fgets(array,20,fp);

- Syntax:

int getc(FILE *stream);

Eg:

ch=fgetc(fp);

Syntax:

int getchar(void);

Eg:

c = fgetchar();

Standard output

- Syntax: fprintf()

int fprintf(FILE *f, char *format, addrofarg1, addrofarg2, ...);

Eg:

fprintf(outfile, "%d:%c\n", x, c);

// write int & char char values to file separated by colon, followd by new line char

- Syntax: fputs()

int fputs(char *s, FILE *f);

Eg:

fputs("This is a test file",fp);

- Syntax:

int fputc(int c, FILE *stream);

Eg:

fputc(ch, fp);

- Syntax:

int putchar(int c);

Eg:

fputchar(c);

Error testing

- Check to see if file is open

```
In_file = fopen (in_name, “r");
if( in_file == NULL )
printf("Cannot open %s for reading.\n", in_name);

out_file = fopen (out_name, "w");
if( out_file == NULL )
printf("Can't open %s for writing. \n", out_name);
```

- Testing for the end- of-file terminator (feof)

It returns 1 if the file pointer is at the end of the file.

```
if( feof ( input_file ))
printf("Ran out of data.\n");
```

Binary file operation

Mode	Description
rb	open binary file for reading
wb	truncate to zero length or create a binary file for writing
ab	append; open or create binary file for writing at end-of-file
rb+	open binary file for update (reading and writing)
wb+	Overwrite or create a binary file for update
ab+	append; open or create binary file for update

- open the source file for reading/writing in binary mode.

```
fp=fopen(file,"rb");
fp=fopen(s1,"wb");
```

- Read/write a character from the source file.

```
ch=fgetc(fp);
fputc(ch,fp);
```

- if EOF, close file.

```
if(feof(fp))
fclose(fp);
```

Binary file I/O

- Fread():

fread() reads a block of data of specified size into a memory location

Syntax:

fread(&object, sizeof(object), n, fp);

// Reads n blocks of size sizeof(object) into object. Unless object is an array n=1.

- Fwrite():

The function fwrite() writes (i.e. appends) a block of data of specified size to a file.

Syntax:

fwrite(&object, sizeof(object), n, fp);

// Writes n objects of size sizeof(object) starting at &object.

```
#include <stdio.h>
#include <string.h>
int main()
{
  FILE *fp;
  char c[] = "this is a text";
  char buffer[100];

  /* Open file for both reading and writing */
  fp = fopen("file.bin", "wb+");

  /* Write data to the file */
  fwrite(c, strlen(c) + 1, 1, fp);

  /* Seek to the beginning of the file */
  fseek(fp, SEEK_SET, 0);

  /* Read and display data */
  fread(buffer, strlen(c)+1, 1, fp);
  printf("%s\n", buffer);
  fclose(fp);
  return(0);
}
```

Header Files

- contains C function declarations and macro definitions to be shared between several source files.
- extension .h
- Once-Only Headers- prevents repetation of header files

```
#ifndef HEADER_FILE
#define HEADER_FILE

the entire header file content

#endif
```

Eg: myfun.h

```
/*define macro MY_FUN, this will reduce
multiple calling of the header file*/


#ifndef MY_FUN
    #define MY_FUN


    /*function declarations*/
    int sumOfNumbers(int x,int y);
    float   avgOfNumbers(int x,int y);


#endif
```

Multiple Files

- Types of files in C
  - file.c - Source file that will contain function definitions.
  - file.h - Header file that will contain function declaration which have definition in file.c source file.
  - main.c – Main Source file that will contain complete code and access the functions which are declared in file.h header file.

- Eg: myfun.c - Source file

```
/*  function name: sumOfNumbers() this function
will return sum of two numbers*/
int sumOfNumbers(int x,int y){
   int sum;
   sum=x+y;
   return sum;
}

/*  function name: sumOfNumbers() this function
will return average of two numbers*/
float avgOfNumbers(int x,int y){
   int sum;
   float avg;
   sum=x+y;
   avg=sum/2;
   return avg;
}
```

Multiple Files cont’d

main.c – Main Source file

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

int main()
{
    int a,b;
    int sum;
    float avg;

    printf("Enter first number: ");
    scanf("%d",&a);
    printf("Enter second number: ");
    scanf("%d",&b);

/*calling sumOfNumbers()*/
    sum=sumOfNumbers(a,b);

    /*calling avgOfNumbers()*/
    avg=avgOfNumbers(a,b);

    printf("Sum: %d \nAverage: %f\n",sum,avg);

    return 0;
}
```

Multiple Files cont’d

- In an IDE:

Just like normal program compilation

- In gcc:

```
gcc main.c myfun.c –o main
gcc *.c –o main
```

or

```
gcc –o main *.c .
```

Dynamic Memory Management

- process of allocating or de-allocating the memory at run time
- header file: <stdlib.h>
- Differences between malloc() and calloc()

Malloc	Calloc
malloc() takes a single argument(the amount of memory to allocate in bytes)	calloc() needs two arguments (the number of variables to allocate in memory, and the size in bytes of a single variable)
malloc() does not initialize the memory allocated	calloc() initializes all bytes of allocated blocks to 0.

Function	Description
Malloc	allocates the specified number of bytes
Calloc	allocates the specified number of bytes and initializes them to zero
Realloc	increases or decreases the size of the specified block of memory. Reallocates it if needed
Free	releases the specified block of memory back to the system

Dynamic Memory Management cont’d

- Syntax:

```
ptr = (cast-type*) malloc(byte-size)
```

- Creating an array of ten integers

```
int * array = malloc(10 * sizeof(int));
```

- Type safety

```
int * ptr;
ptr = malloc(10 * sizeof(int));          /* without a cast */
ptr = (int *)malloc(10 * sizeof(int));   /* with a cast */
```

calloc will allocate and clear the memory

- Syntax:

```
ptr = (cast-type*)calloc(n, element-size);
```

- Eg:

```
int * array = calloc(10, sizeof (int));
```

allocates a region of memory large enough to hold 10 integers, and sets to zero

- Deallocate the memory block

```
free(array);
```

Dynamic Memory Management cont’d

- Increase or decrease the size of an allocated memory block using realloc()

- Syntax:

```
ptr = realloc(ptr, newsize);
```

- Eg:

```
ptr= malloc( 30 * sizeof(char) );
ptr = realloc( ptr, 100 * sizeof(char) );
```

- Common errors
- allocation failures

Memory allocation is not guaranteed to succeed

making sure if allocation is successful

```
int * array = malloc(10 * sizeof(int));
if (array==NULL)
{
  fprintf(stderr, "malloc failed\n");
  return(-1);
}
```

Common Errors cont’d

- Memory leaks
  - Failure to deallocate memory using free leads to buildup of non-reusable memory

- Logical errors
  - memory usage after a call to free (dangling pointer)
  - memory usage before a call to malloc (wild pointer),
  - calling free twice ("double free"),

Example Program

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
  char name[100];
  char *description;
  strcpy(name, "Zara Ali");

  /* allocate memory dynamically */
  description = malloc( 30 * sizeof(char) );

  if( description == NULL ) {
    fprintf(stderr, "Error - unable to allocate required
memory\n");
  }
  else {
    strcpy( description, "Zara ali a DPS student.");
  }
 /* suppose you want to store bigger description */
  description = realloc( description, 100 * sizeof(char) );

  if( description == NULL )
{
    fprintf(stderr, "Error - unable to allocate required
memory\n");
  }
  else {
    strcat( description, "She is in class 10th");
  }
  printf("Name = %s\n", name );
  printf("Description: %s\n", description );

  /* release memory using free() function */
  free(description);
}
```

Error handling

- Global Variable errno
- C language uses the following functions to represent error

- perror() return string pass to it along with the textual representation of current errno value.
- strerror() is defined in string.h library. This method returns a pointer to the string representation of the current errno value.

Example Program

```
#include <stdio.h>       /* perror */
#include <errno.h>       /* errno */
#include <stdlib.h>      /* malloc, free, exit */

int main(void)
{
   /* Pointer to char, requesting dynamic allocation
of 2,000,000,000
    * storage elements (declared as an integer
constant of type
    * unsigned long int). (If your system has less
than 2 GB of memory
    * available, then this call to malloc will fail.)
    */
   char *ptr = malloc(2000000000UL);

 if (ptr == NULL) {
       perror("malloc failed");
       /* here you might want to exit the program or
compensate
         for that you don't have 2GB available
        */
   }
 else
 {
       /* The rest of the code hereafter can assume
that 2,000,000,000
        * chars were successfully allocated...
        */
       free(ptr);
   }
   exit(EXIT_SUCCESS); /* exiting program */
}
```

Error handling cont’d

Preventing divide by zero errors

```
if (divisor == 0)
 {
      /* Example handling of this error. Writing a message to stderr, and
       * exiting with failure.
       */
      fprintf(stderr, "Division by zero! Aborting...\n");
      exit(EXIT_FAILURE); /* indicate failure.*/
   }
```

GNU GCC Compiler

- standard way to compile a c program

```
gcc -o hello.exe hello.c
  // Compile and link source file hello.c into executable hello.exe
hello
  // Execute hello.exe under CMD shell
```

- GCC Compiler Options

-o: specifies the output executable filename.

-Wall: prints "all" warning messages.

-g: generates additional symbolic debuggging information for use with gdb debugger.

-c: compile into object file "Hello.o"

```
gcc -Wall -g -o Hello.exe Hello.c
```

- Compile and Link Separately

```
// Compile-only with -c option
 gcc -c -Wall -g Hello.c
// Link object file(s) into an executable
 gcc -g -o Hello.exe Hello.o
```

- Compile and Link Multiple Source Files

```
 gcc -o myprog.exe file1.c file2.c
or
 gcc -c file1.c
gcc -c file2.c
 gcc -o myprog.exe file1.o file2.o
```

GNU GCC Compiler cont'd

- GCC Compilation Process
  - Pre-processing:

cpp hello.c > hello.i

  - Compilation:

gcc -S hello.i

  - Assembly:

as -o hello.o hello.s

  - Linker:

ld -o hello.exe hello.o ...libraries...

- Default Include-paths, Library-paths and Libraries
  - To look for header files in other directories apart from default locations specify with -I

gcc -o main main.c -I/new/location/include

  - Specify library paths with the -L option

gcc -L/usr/lib -o hello hello.c

  - Link to external library with –l option.

gcc -L/usr/lib -lfile -o program.exe main.c

Creating Libraries

- Creating static libraries
- Extentions – .a(linux), .lib(windows)
- create the object file from source code.

```
gcc –c file.c
```

- archiving to library file.

```
ar rs libfile.lib file.o
```

- Creating shared library
- Extentions– .so(linux), .dll(windows)
- create "position independent code"

```
gcc –c –fPIC file.c
```

- create the shared library file.

```
gcc –shared –o libfile.dll file.o
```

Note: When specifying library file with the –l option the beginning "lib" and the ending extension is left out as it is added by the linker by default .

```
Eg: gcc -o test.exe test.o -L. –ladd
```

Where add is libadd.lib/libadd.dll

Make File

The "make" utility automates building executable from source code.

"make" uses makefile, which contains rules on how to build the executables.

Eg:

```
all: hello.exe

hello.exe: hello.o
        gcc -o hello.exe hello.o

hello.o: hello.c
        gcc -c hello.c

clean:
        rm hello.o hello.exe
```

A rule consists of 3 parts: a target, a list of pre-requisites and a command,

Syntax:

```
target: pre-req-1 pre-req-2 ...
        command
```

Run the "make" utility

➢ Make

Running make without argument starts the target "all" in the makefile.

In order to run make utility for a particular target

> make <target>

Eg:

> make clean

Make File cont’d..: Macros

```
# the compiler: gcc for C program, define as g++ for C++
 CC = gcc
 # compiler flags:
 #  -g    adds debugging information to the executable file
 #  -Wall turns on most, but not all, compiler warnings
 CFLAGS  = -g -Wall
 # the build target executable:
 TARGET = myprog
 all: $(TARGET)

 $(TARGET): $(TARGET).c
             $(CC) $(CFLAGS) -o $(TARGET) $(TARGET).c

 clean:
             $(RM) $(TARGET)
```

Eg:

```
CC = gcc
CFLAGS =  -g -Wall
OBJECTS = main.o foo.o

main.exe : $(OBJECTS)
             $(CC) $(CFLAGS) $(OBJECTS) -o main.exe

main.o : main.c
             $(CC) $(CFLAGS) -c main.c

foo.o : foo.c
             $(CC) $(CFLAGS) -c foo.c
clean:
             rm foo.o main.o
```