# Tut

Date - 25-10-24

TA - Harshvardhan

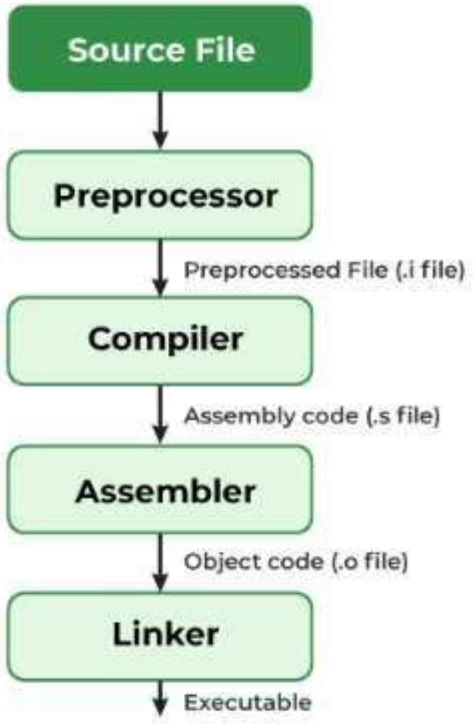
# What is Multi-File Programming?

- In C programming, multi-file programming refers to organizing your code across multiple files, which can help improve the modularity, readability, and maintainability of a project.

- Instead of writing all the code in one large file, you split it into separate files with each file responsible for a specific part of the program (e.g., function definitions, data structures, or global variables).

- The general practice is to separate your code into **header files (.h)** and **source files (.c)**.

# Compilation Process (Revisit)



Source File

Preprocessor

Preprocessed File (.i file)

Compiler

Assembly code (.s file)

Assembler

Object code (.o file)

Linker

Executable

Link - https://www.geeksforgeeks.org/compiling-a-c-program-behind-the-scenes/

# Header File ( math_ops.h )

here

i) #ifndef MATH_OPS_H

ii) #define MATH_OPS_H

iii) int add(int a, int b);

iv) int subtract(int a, int b);

v) #endif

# Why the Macros ?

The #ifndef, #define, and #endif directives are include guards to prevent multiple inclusions of the same header file

# Function Declaration ( math_ops.c)

```
// math_ops.c
vi) #include "math_ops.h"

vii) int add(int a, int b) {
     return a + b;
      }

Viii) int subtract(int a, int b) {
          return a - b;
       }
```

# Main Function ( main.c )

```
ix)        #include <stdio.h> //Include  -- stdio.o
x)         #include "math_ops.h“ //
xi)        #include "math_ops.h“ (again)

xi)   int main() {
xii)  int result1 = add(10, 5);
xiii) int result2 = subtract(10, 5);

xiv) printf("Addition: %d\n", result1);
xv)  printf("Subtraction: %d\n", result2);

xvi) return 0;
       }
```

# Now how to compile ?

```
gcc main.c math_ops.c -o program
```

Breakup of what is happening ?

```
(gcc -c math_ops.c

gcc -c main.c
gcc -o program main.o math_ops.o)
```

# Advantages

- **Modularity**: Each module (or functionality) of your program is isolated into its own file, making it easier to maintain and develop.
- **Reusability**: You can reuse the same code across different projects by simply including the relevant header and source files.
- **Teamwork**: Different team members can work on different parts of the project without interfering with each other.
- **Faster Compilation**: When making changes, only the modified source files need to be recompiled, not the entire program.

# Make

```
CC = gcc
# Compiler flags
CFLAGS = -Wall -Wextra -std=c99
# Target executable name
TARGET = program
# Source files
SRCS = main.c math_ops.c
# Object files (derived from source files)
OBJS = $(SRCS:.c=.o)
# Default target to build the executable
$(TARGET): $(OBJS)
$(CC) $(CFLAGS) -o $(TARGET) $(OBJS)
# Rule to compile .c files into .o files
%.o: %.c
$(CC) $(CFLAGS) -c $< -o $@
# Clean up object files and the executable
.PHONY: clean
clean:
rm -f $(OBJS) $(TARGET)
```

# File I/O

```
FILE *fp;
fp = fopen("filename.txt", "mode");
```

# Ahh Pointers….

**Now …**

**Difference Between :**

**Const int * ptr;**

**v/s**

**Int * const ptr;**

# Ahh Pointers….

**Now …**

**Difference Between :**

**Const int * ptr;**

**v/s**

**Int * const ptr;**

# Ahh Pointers….

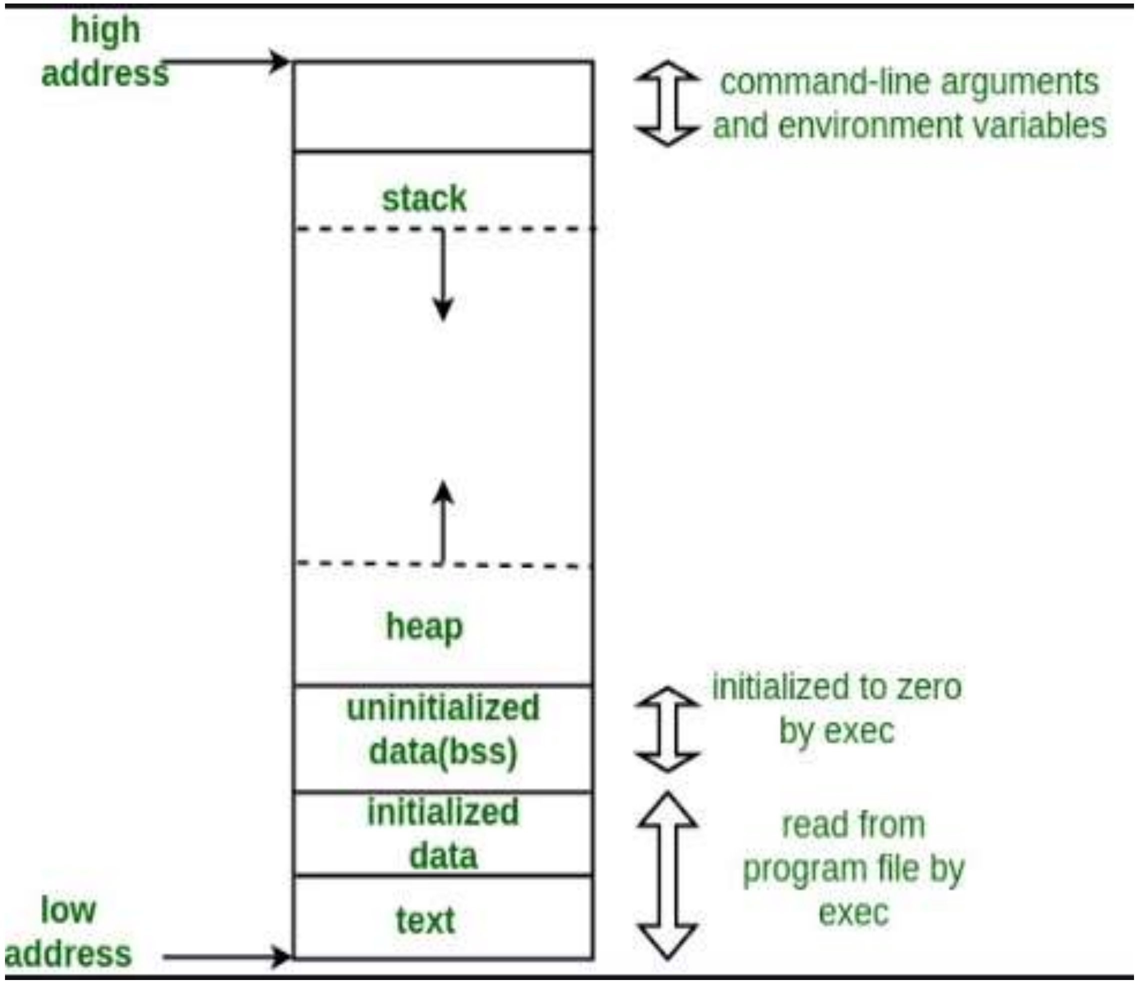
```
int x = 10;
int y = 20;

// const int * ptr: Can modify the pointer, but not the integer
const int * ptr1 = &x;
ptr1 = &y; // OK
// *ptr1 = 30; // Error: Cannot modify the integer

// int * const ptr: Cannot modify the pointer, but can modify the integer
int * const ptr2 = &x;
// ptr2 = &y; // Error: Cannot modify the pointer
*ptr2 = 30; // OK
```

How the const knows this ?

# Memory Layout revisited


high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

# File I/O

"r": Read mode (opens the file for reading).

"w": Write mode (creates a new file or overwrites an existing one).

"a": Append mode (appends data to the end of an existing file).

"rb": Read binary mode (for reading binary data).

"wb": Write binary mode (for writing binary data).

"ab": Append binary mode (for appending binary data).

# File I/O

fscanf: Reads formatted data from the file.

fgetc: Reads a single character from the file.

fgets: Reads a line from the file.

fputc: Writes a single character to the file.

fputs: Writes a string to the file.

fprintf: Writes formatted data to the file.

# File I/O

Code Demo