Module Code: CSE401

Session 06:Pre-processor Directives and Memory Management

Session Speaker :

Prema Monish
premamonish.tlll@msruas.ac.in

Session Objectives

- To learn about Preprocessor directives
- To understand concept of File Inclusion
- To understand concept of Conditional Compilation
- To understand about memory management in C

Session Topics

- Library functions and header files
- #defined and #include directives
- Macro substitution
- Makefile in linux
- Conditional compilation directives
- Memory management in C

Preprocessor Directives

- C statements start with # symbol are processed by the preprocessor before compilation.
- The result of preprocessor is expanded source, which will be input to the compiler.
- compiler, which identifies the errors and warnings if any.
- Types of preprocessor directives:
  1. Macro substitution
  2. File inclusions
  3. Conditional inclusion/Compilation
  4. pragma directive

Preprocessor Directives

Rules

- We must follow certain rules while writing preprocessor statement, Some of these rules are
- All preprocessor directives must be started with # symbol
- Every preprocessor statement may be started from the first column (Optional)
- There shouldn’t be any space between # and directive
- A preprocessor statement must not be terminated with a semicolon
- Multiple preprocessor statements must not be written in a single line

Macro

- Macro substitution has a name and replacement text, defined with #define directive.
- The preprocessor simply replaces the name of macro with replacement text from the place where the macro is defined in the source code.
- A macro can be undefined using #undef

Syntax:

```
#define  name  replacement text
```

Macro

```
#define PI 3.14
#include<stdio.h>
int main()
{
 int rad;
 float area,cir;
 printf("Enter the radios:");
 scanf("%d",&rad);
 printf("Area %f",PI*rad*rad);
 #undef PI;
 printf("\nCircumference %f",2*PI*rad);
 return 0;
}
```

Execution:
Enter the radios: 15
Area 706.500000
Error undefined symbol PI

Macros with arguments

- even a macro can be defined with arguments. It has the name, arguments and replacement text.

Syntax:

```
#define  name (parameters)  replacement text
```

Macros with arguments

```
#include<stdio.h>
#define MAX(x,y) x>y?x:y
#define MIN(x,y) x<y?x:y
int main()
{
 int a,b,big,small;
 printf("Enter two numbers:\n");
 scanf("%d%d",&a,&b);
 if(a==b)
   printf("Equal");
 else
 {
   big=MAX(a,b);
   small=MIN(a,b);
   printf("Biggest number %d",big);
   printf("\nSmallest number %d",small);
 }
 return 0;
}
```

Execution:
Enter two numbers:
45
45
Equal

Differences: Macros & Functions

Macros	***Functions***
• They are expanded at ***pre-compile*** time i.e. preprocessing phase • They are expanded by the ***preprocessor***. • They do not follow any rules. It is merely a ***replacement.*** • Increases the performance of program • Macro parameters are ***generic***	• They are expanded at ***compile time***. • They are parsed by the ***compiler***. • They follow ***all the rules*** enforced on functions. • Comparatively ***Less performance*** • Function parameters i.e. actual and formal parameters are ***not generic***

Linking Multiple Files in Linux

- If we want to develop any big application, then we divide different sections of application into different C files and then link their object files to build a single executable file.
- consider an example defining different functions in different files and using all of them in another file, compiling and linking all together.

Linking Multiple Files in Linux

- Step 1: Create function libraries and save them with .c extension
  - Defining a function add() in a file “addition.c”, that takes two integers and returns sum of them
  - $vim addition.c
  - Defining a function sub() in a file “subtraction.c”, that takes two integers and returns subtraction of them
  - $vim subtraction.c
  - Writing a program in a file “main.c”, that uses both the functions add() and sub() defined in “addition.c”, “subtraction.c”
  - $vim main.c

Linking Multiple Files in Linux

```
/* program in addition.c*/

int add(int var1,int var2)
{
  return var1+var2;
}
```

```
/* program in subtraction.c*/

int sub(int var1,int var2)
{
  return var1-var2;
}
```

```
/* program in main.c*/

#include<stdio.h>
int main()
{
    int a,b,sumresult,subresult;
    printf("Enter two integers:\n");
    scanf("%d%d",&a,&b);
    sumresult=add(a,b);
    subresult=sub(a,b);
    printf("Sum of two numbers %d",
    sumresult);
    printf("\nSubtraction of two numbers
    %d", subresult);
    return 0;
}
```

Linking Multiple Files in Linux

- Step 2: We can compile and link all the files and generate a single executable file using gcc compiler. Here we may give the files in any order but, the gcc generates executable file (.out)
  - $cc –o main main.c one.c two.c
  - $./main
  - Enter two integers:
    12
    10
    Sum of two numbers 22
    Subtraction of two numbers 2

File Inclusion

- The ***#include*** is the directive for file inclusion.
- This directive causes one file to be included in another.
- There are two ways of writing the #include directive.
  - They are :

```
#include "file_name"
#include <file_name>
```

File Inclusion

- Consider an example of three files main.c,addition.c,subtraction.c and header.h
- Step 1: Defining add() function in addition.c
  - $vim addition.c
  - Defining sub() function in subtraction.c
  - $vim subtraction.c
  - Defining main() function in main.c
  - $vim main.c
  - Create a header file with the prototypes of library functions and save with .h file extension
  - $vim header.h

File Inclusion

```
/* program in addition.c*/
#include “header.h”
int add(int var1,int var2)
{
  return var1+var2;
}
```

```
/* program in subtraction.c*/
#include “header.h”
int sub(int var1,int var2)
{
  return var1-var2;
}
```

```
/* program in header.h*/

int sub(int,int);
int add(int,int);
```

```
/* program in main.c*/

#include<stdio.h>
#include “header.h”
int main()
{
    int a,b,sumresult,subresult;
    printf("Enter two integers:\n");
    scanf("%d%d",&a,&b);
    sumresult=add(a,b);
    subresult=sub(a,b);
    printf("Sum of two numbers %d",
    sumresult);
    printf("\nSubtraction of two numbers
    %d", subresult);
    return 0;
}
```

File Inclusion

- A **makefile** is basically a script that guides the make utility to choose the appropriate program files that are to be compiled and linked together
- Step 2 : creating make file makefile and redirecting output to file called three
  - $vim makefile

```
/* program in makefile*/
three : addition.c subtraction.c main.c
        gcc -o three addition.c subtraction.c main.c
```

  - $make
  - ./three

```
Enter two integers:
12
10
Sum of two numbers 30
Subtraction of two numbers 10
```

Conditional compilation

- It is the process of selecting the source code conditionally from the program and sending to the compiler using preprocessor statements like #if, #ifdef, #ifndef, #else, #endif.
- #if preprocessor conditional statement:

Syntax

```
#if <expression>
   ---------------
   ---------------
#else
   ---------------
   ---------------
#endif
```

Conditional compilation

```
/* main.c */
#include <stdio.h>
int main()
{
    int a,b;
    printf("Enter two numbers:\n");
    scanf("%d%d",&a,&b);
    #if 5>10
     printf("Sum %d",a+b);
    #else
     if(a==b)
      printf("Eqauls");
     else if(a>b)
      printf("Biggest number %d",a);
     else
      printf("Biggest number %d",b);
    #endif
    return 0;
}
```

In the example as 5>10 gives an integer constant 0 (false), the preprocessor sends the code between #else and #endif to the compiler. Now we can realize this by looking at it's expanded source

Execution:
Enter two numbers:
45
30
Biggest number 45

Conditional compilation

- defined() preprocessor-It checks whether a macro with the name is defined or not. It returns true if in case macro with the name is defined otherwise returns false

```
#include <stdio.h>
#define MAX 10
int main()
{
    #if defined(MAX)
     printf("Hello");
    #else
     printf("World");
    #endif
     printf("\n");
    #if !defined(MIN)
     printf("Min Not defined");
    #else
     printf("Min defined");
    #endif
    return 0;
}
```

Output:
Hello
Min Not defined

Conditional compilation

- #ifdef, #ifndef preprocessor conditional statements -We can use #ifdef in place of #if defined(), #ifndef in place of #if !defined().

Syntax

```
#ifdef  MACRO
   ---------------
   ---------------
#else
   ---------------
   ---------------
#endif
```

Syntax

```
#ifndef  MACRO
   ---------------
   ---------------
#else
   ---------------
   ---------------
#endif
```

Conditional compilation Usage

- To experiment with the code
- To develop portable applications- selecting proper code while compiling

```
#include<stdio.h>
#define INTEL
int main()
{
 ------------
 ------------   //common code
 ------------
 #ifdef INTEL
 --------------
 -------------- // code for INTEL
 --------------
 #else
 --------------
 -------------- // code for Motorola
 --------------
 #endif
 --------------
 -------------- // common code
 --------------
}
```

Pragma directive

- It is a special directive to the compiler that helps to turn on/off certain features.
- Pragmas are compiler dependents, different pragmas are supported by different compilers.
- If compiler doesn’t recognize a pragma then ignores it without showing any errors or warnings.

Pragma directive

```
#include<stdio.h>
void first();
void last();
#pragma startup first
#pragma exit last
int main()
{
 printf("\nWithin main..");
 return 0;
}
void first()
{
 printf("\nWithin First..");
}
void last()
{
 printf("\nWithin Last..");
}
```

Output:
Within First..
Within main..
Within Last..

Stringizing Operator ‘#’

- The macro parameters in the strings that appear in the macro definitions are not recognized.
- To substitute a macro argument in a string constant, a *special notation ‘#’ in the macro body*.
- When the ‘#’ character precedes a parameter in the macro body, a string constant is replaced for both # and the parameter.
- The notation ‘#’ is known as the Stringizing Operator.

An Example: '#'

```
#define STRING(x,y) #x"developed by"#y
main()
{
        char s[]=STRING(PROGRAM,Priya);
        printf("%s\n",s);
}
```

Output:

PROGRAM developed by Priya

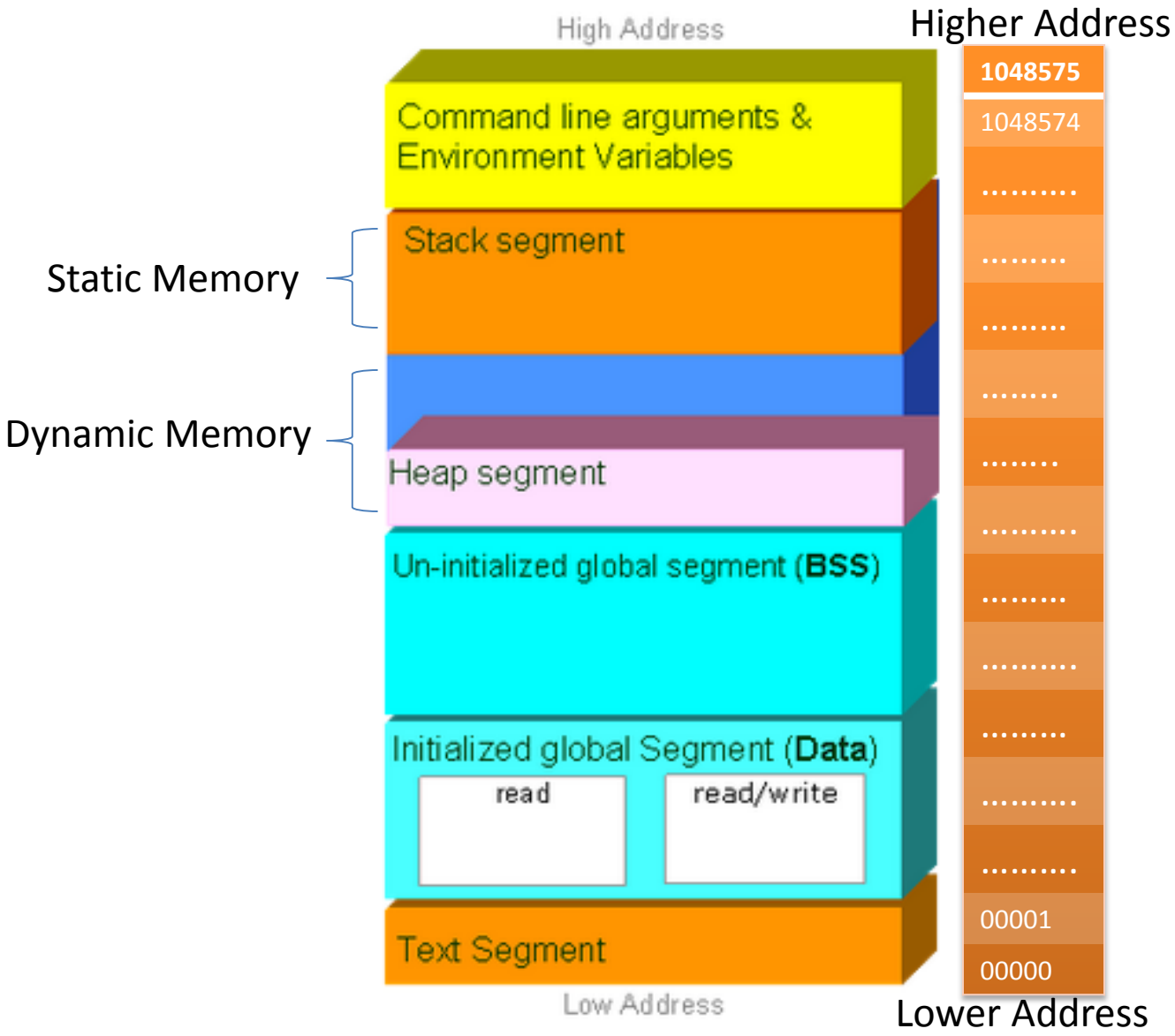
Advantages of Preprocessors

- A preprocessor improves the readability of programs.
- It facilitates easier modifications.
- It helps in writing portable programs.
- It enables easier debugging.
- It helps in developing generalized program.

Memory management in C

- Memory is organized into number of segments on loading a C program. Here program may be a text editor, a compiler or an audio player etc.
- These segments are:
  1. Stack segment
  2. Heap segment
  3. Initialized global segment (Data segment)
  4. Un-initialized global segment (BSS segment)
  5. Text segment (Code segment)

Memory Management in C



Text Segment

- The program loaded to execute is stored in the text segment, which is also called code segment.
- Generally it is set as read only segment to prevent from accidental modifications by other applications.
- A program loaded into this segment is sharable so that single copy loaded into the memory can be used by any number of applications. Say for example dictionary component loaded in the code segment can be used by any application.

Text Segment

- Now let us check how much size a simple C program takes in the text segment using **size** command in Linux.
- /* hello.c */

```
#include<stdio.h>
int main()
{
    int x,y,z;
   x=10;
   y=20;
   z=x+y;
   printf("Sum of two numbers %d",z);
   return 0;
}
```

Text Segment

$ vim hello.c
$cc –o hello hello.c
$ ./hello
Sum of two numbers 30
$ size ./hello

```
Text    data    bss     dec     hex     filename
976     264     8       1248    4e0     ./hello
```

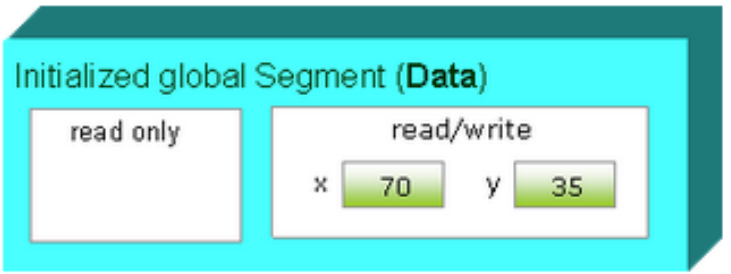
Initialized global segment

- It is generally called as the data segment. All the global and static variables initialized by the program are stored in the Data segment.
- It is further classified into read only segment and read/write segments.
- static and global variables, which are initialized by the programmer are stored in read/write portion of Data segment and their values can be changed in the program.

Initialized global segment

```
#include<stdio.h>
 int y=25;   /* initialized global variable */
int main()
{
      static int x=45; /* initialized static variable */
      x=x+25;
       printf("x=%d",x);
       y=y+10;
       printf("\ny=%d",y);
       return 0;
 }
```

Output



Initialized global segment

- static and global variables defined as const variables are stored in the read-only section of Data segment and initialized values of these variables can't be changed

```
#include<stdio.h>
const int y=25; /* initialized constant global variable */
 int main()
 {
      y=y+10; /* error: assignment of read-only variable 'y' */
     printf("\ny=%d",y);
     return 0;
}
```

Output:

Uninitialized global segment

- It is generally known as BSS segment, named after an ancient assembler operator Block Started by Symbol.
- All the static and global variables, which are not initialized by the programmer are stored in this segment.
- These variables are initialized with zero by kernel of operating system

Uninitialized global segment

```
#include<stdio.h>
 int y; /* uninitialized global variable */
int main()
{
      static int x; /* uninitialized static variable */
      printf("x=%d",x);
      printf("\ny=%d",y);
      return 0;
}
```

Output

Uninitialized global segment

- Let us test some cases using the size command of Linux. The size command gives the size of text, data and BSS segments.
- Here the program name is “demo.c”

```
#include<stdio.h>
int main()
{
 return 0;
}
```

$cc –o demo demo.c
$size ./demo

```
text       data        bss         dec         hex     filename
960        248           8        1216        4c0     demo.c
```

Uninitialized global segment

- Let us add one global variable “x” in the program and check the size of BSS

```
#include<stdio.h>
int x; /* Uninitialized variable stored in bss*/
int main()
{
 return 0;
}
```

$cc –o demo demo.c
$size ./demo

```
text        data        bss         dec         hex     filename
960         248         12          1220        4c4     demo.c
```

Here the size of BSS is incremented by 4 as un-initalized global variable is stored in BSS.

Uninitialized global segment

- Let us add one static variable to

```
#include<stdio.h>
int x; /* Uninitialized variable stored in bss*/
int main()
{
 static int y; /* Uninitialized static variable stored in bss */
 return 0;
}
```

$cc –o demo demo.c
$size ./demo

```
text        data        bss         dec         hex     filename
960         248         16          1224        4c8      demo.c
```

- Here the size of BSS further incremented by 4 as the static un-initalized variable is also stored in BSS

Uninitialized global segment

- if we initialize the global variable & static variable then it is stored in the Data segment instead in BSS

$vim demo.c

```
#include<stdio.h>
int x=20; /* initialized global variable stored in Data segment*/
int main()
{
 static int y=45; /* Initialized static variable stored in Data segment*/
 return 0;
}
```

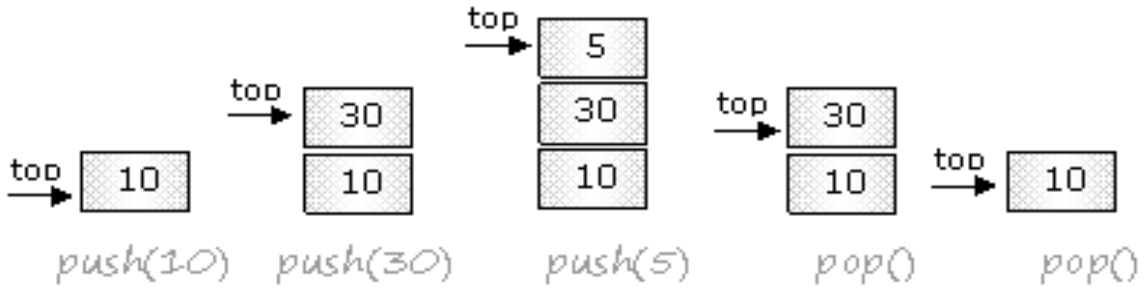
$gcc –o demo demo.c
$size ./demo

```
text        data        bss         dec         hex     filename
960         256          8        1224         4c8     demo.c
```

- Here the size of BSS decremented by 8 as the static & global initialized variable is not stored in BSS

Stack Segment

- If we consider a stack of plates in a cafeteria, we can perform limited operations on the stack of plates like adding a new plate at the top, removing a plate from the top and looking the color of a plate at a position from the to etc.
- Here adding a new plate on the top of plates stack is called push operation,
- removing a plate from the top of the stack is called pop operation.
- Limitation with the stack is that, it works on LIFO principle that is last added plate is first removed (Last In First Out).



Stack Segment

- The stack grows towards the lower part of memory that is towards heap.
- In other hand heap grows towards higher part of memory that is towards stack. If both heap and stack meets at a point then there would be no free memory and the state is called stack overflow.
- Local variables of functions, parameters and addresses of calling functions are pushed on the stack segment. Stack grows as the number of functions called is increased.

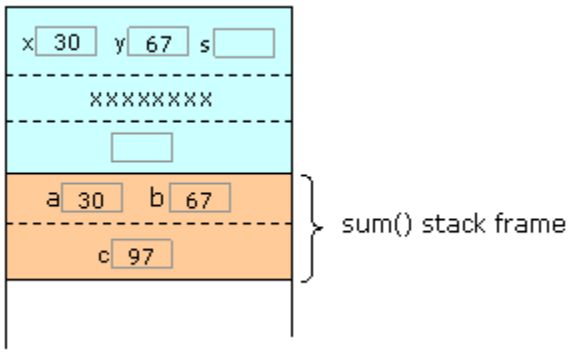
Stack Segment

What happens when a function is called?

1. The address of the instruction next to the function call is pushed onto the stack. This is how the CPU remembers where to go after the completion of function execution.
2. Space is selected on the stack for the function's return type. It is just a placeholder for now.
3. The CPU jumps to the function's definition.
4. Now, the current top of the stack is held in a special pointer called the stack frame. Everything added to the stack after this point is considered local to the function.
5. All function arguments are placed on the stack.
6. The instructions inside the function are executed.
7. Local variables are pushed onto the stack as they are defined.

Stack Segment

```
#include<stdio.h>
int sum(int,int);
int main()
{
        int x,y,s;
        x=30,y=67;
        s=sum(x,y);
        printf("Sum of two numbers %d",s);
        return 0;
}
int sum(int a,int b)
{
        int c;
        c=a+b;
        return c;
}
```



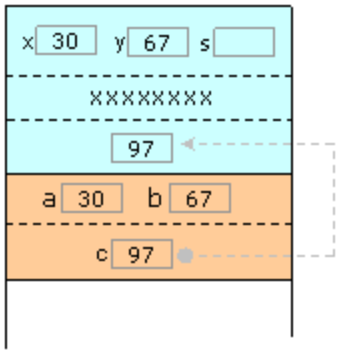
Stack Segment

What happens when a function is terminated?

1. The function's return value is copied into the placeholder that was put on the stack for the purpose.
2. Everything after the stack frame pointer is popped off. This destroys all local variables and arguments.
3. The return value is popped off the stack and is assigned as the value of the function. If the value of the function isn't assigned to anything, no assignment takes place, and the value is lost.
4. The address of the next instruction to execute is popped off the stack, and the CPU resumes execution from that instruction.

Stack Segment

```
#include<stdio.h>
int sum(int,int);
int main()
{
        int x,y,s;
        x=30,y=67;
        s=sum(x,y);
        printf("Sum of two numbers %d",s);
        return 0;
}
int sum(int a,int b)
{
        int c;
        c=a+b;
        return c;
}
```



Summary

- A preprocessor is a facility provided for writing portable programs, easier program modifications and easier debugging.
- A preprocessor processes the source code program before it passes through the compiler.
- A macro is a simple function having its own syntax using #define
- Handling multiple file in linux
- Makefile in linux
- Memory management in C