Module IV

Declaration Initialization -Access of structure Variables- Array of structure – Arrays within structure-Structure within structures- Structures and Functions –Pointer to Structure

STRUCTURE

&

UNION

Data Types

C programming language which has the ability to divide the data into different types. The type of a variable determine the what kind of values it may take on. The various data types are.

- Simple Data type

  → Integer, Real, Void, Char

- Structured Data type

  →Array, Strings

- User Defined Data type

  →Enum, Structures, Unions

Structure Data Type

A structure is a user defined data type that groups logically related data items of different data types into a single unit. All the elements of a structure are stored at contiguous memory locations. A variable of structure type can store multiple data items of different data types under the one name. As the data of employee in company that is name, Employee ID, salary, address, phone number is stored in structure data type.

Defining of Structure

A structure has to defined, before it can used. The syntax of defining a structure is

```
struct <struct_name>

{

<data_type> <variable_name>;

<data_type> <variable_name>;

……..

<data_type> <variable_name>;

};
```

Example of Structure

The structure of Employee is declared as

```
struct    employee
{
int         emp_id;
char        name[20];
float       salary;
char        address[50];
int         dept_no;
int         age;
};
```

Memory Space Allocation

8000	emp_id
8002	name[20]
8022	salary
8024	address[50]
8074	dept_no
8076	age

Declaring a Structure Variable

A structure has to declared, after the body of structure has defined. The syntax of declaring a structure is

Struct <struct_name> <variable_name>;

The example to declare the variable for defined structure "employee"

struct employee e1;

Here e1 variable contains 6 members that are defined in structure.

Initializing a Structure Members

The members of individual structure variable is initialize one by one or in a single statement. The example to initialize a structure variable is

1)struct employee e1 = {1, “Hemant”,12000, “3 vikas colony new delhi”,10, 35);

2)e1.emp_id=1; e1.dept_no=1

e1.name=“Hemant”; e1.age=35;

e1.salary=12000;

e1.address=“ LPU JALANDHAR”;

Accessing a Structure Members

The structure members cannot be directly accessed in the expression. They are accessed by using the name of structure variable followed by a dot and then the name of member variable. The method used to access the structure variables are e1.emp_id, e1.name, e1.salary, e1.address, e1.dept_no, e1.age. The data with in the structure is stored and printed by this method using scanf and printf statement in c program.

Structure Assignment

The value of one structure variable is assigned to another variable of same type using assignment statement. If the e1 and e2 are structure variables of type employee then the statement

e1 = e2;

assign value of structure variable e2 to e1. The value of each member of e2 is assigned to corresponding members of e1.

```
#include<stdio.h>
#include<conio.h>
struct employee
{
int  emp_id;
char name[20];
float salary;
char address[50];
int dept_no;
int age;
};
void main ( )
  { struct      employee    e1,     e2;
                            printf (“Enter the employee id of employee”);
                            scanf(“%d”,&e1.emp_id);
                            printf (“Enter the name of employee”);
                            scanf(“%s”,e1.name);
                            printf (“Enter the salary of employee”);
                            scanf(“%f”,&e1.salary);
                           printf (“Enter the address of employee”);
                           scanf(“%s”,e1.address);
                           printf (“Enter the department of employee”);
                           scanf(“%d”,&e1.dept_no);
                           printf (“Enter the age of employee”);
                           scanf(“%d”,&e1.age);
```

```
printf ("Enter the employee id of employee");
scanf("%d",&e2.emp_id);
    printf ("Enter the name of employee");
    scanf("%s",e2.name);
    printf ("Enter the salary of employee");
    scanf("%f",&e2.salary);
    printf ("Enter the address of employee");
    scanf("%s",e2.address);
    printf ("Enter the department of employee");
    scanf("%d",&e2.dept_no);
    printf ("Enter the age of employee");
    scanf("%d",&e2.age);
    printf ("The employee id of employee is : %d", e1.emp_id);
    printf ("The name of employee is : %s", e1.name);
    printf ("The salary of employee is : %f", e1.salary);
    printf ("The address of employee is : %s", e1.address);
    printf ("The department of employee is : %d", e1.dept_no);
    printf ("The age of employee is : %d", e1.age);
```

Program to implement the Structure

```
printf ("The employee id of employee is : %d",
        e2.emp_id);
printf ("The name of employee is : %s",
        e2.name);
printf ("The salary of employee is : %f",
        e2.salary);
printf ("The address of employee is : %s",
        e2.address);
printf ("The department of employee is : %d",
        e2.dept_no);
printf ("The age of employee is : %d",e2.age);
getch();
}
```

Output of Program

Enter the employee id of employee 1
Enter the name of employee Rahul
Enter the salary of employee 15000
Enter the address of employee 4,villa area, Delhi
Enter the department of employee 3
Enter the age of employee 35
Enter the employee id of employee 2
Enter the name of employee Rajeev
Enter the salary of employee 14500
Enter the address of employee flat 56H, Mumbai
Enter the department of employee 5
Enter the age of employee 30

Output of Program

The employee id of employee is : 1
The name of employee is : Rahul
The salary of employee is : 15000
The address of employee is : 4, villa area, Delhi
The department of employee is : 3
The age of employee is : 35
The employee id of employee is : 2
The name of employee is : Rajeev
The salary of employee is : 14500
The address of employee is : flat 56H, Mumbai
The department of employee is : 5
The age of employee is : 30

Array of Structure

C language allows to create an array of variables of structure. The array of structure is used to store the large number of similar records. For example to store the record of 100 employees then array of structure is used. The method to define and access the array element of array of structure is similar to other array. The syntax to define the array of structure is

Struct <struct_name> <array_name> [<value>];

For Example:-

Struct employee e1[100];

Program to implement the Array of Structure

```
#include <stdio.h>
#include <conio.h>
struct employee
{
int  emp_id;
char name[20];
float salary;
char address[50];
int dept_no;
int age;
};
```

Program to implement the Array of Structure

```
void main ( )
  {
    struct employee e1[5];
    int i;
    for (i=1; i<=100; i++)
    {
    printf ("Enter the employee id of employee");
    scanf ("%d",&e[i].emp_id);
    printf ("Enter the name of employee");
    scanf ("%s",e[i].name);
    printf ("Enter the salary of employee");
    scanf ("%f",&e[i].salary);
```

Program to implement the Array of Structure

```
printf ("Enter the address of employee");
scanf ("%s", e[i].address);
printf ("Enter the department of employee");
scanf ("%d",&e[i].dept_no);
printf ("Enter the age of employee");
 scanf ("%d",&e[i].age);
 }
for (i=1; i<=100; i++)
{
 printf ("The employee id of employee is : %d",
             e[i].emp_id);
 printf ("The name of employee is: %s",e[i].name);
```

Program to implement the Array of Structure

```
  printf (“The salary of employee is: %f”,
                e[i].salary);
  printf (“The address of employee is : %s”,
                e[i].address);
   printf (“The department of employee is : %d”,
                e[i].dept_no);
   printf (“The age of employee is : %d”, e[i].age);
   }
getch();
}
```

Structures within Structures

C language define a variable of structure type as a member of other structure type. The syntax to define the structure within structure is

```
struct <struct_name>{
<data_type> <variable_name>;
                    struct <struct_name>
                                        {    <data_type>
<variable_name>;
                        ……..}<struct_variable>;
<data_type> <variable_name>;
```

Example of Structure within Structure

The structure of Employee is declared as

```
 struct employee
  {  int  emp_id;
     char name[20];
     float salary;
     int dept_no;
     struct date
        {    int day;
             int month;
             int year;
        }doj;
};
```

Accessing Structures within Structures

The data member of structure within structure is accessed by using two period (.) symbol. The syntax to access the structure within structure is

struct _var. nested_struct_var. struct_member;

For Example:-

e1.doj.day;

e1.doj.month;

e1.doj.year;

Pointers and Structures

C language can define a pointer variable of structure type. The pointer variable to structure variable is declared by using same syntax to define a pointer variable of data type. The syntax to define the pointer to structure

```
struct <struct_name> *<pointer_var_name>;
```

For Example:

```
struct employee *emp;
```

It declare a pointer variable “emp” of employee type.

Access the Pointer in Structures

The member of structure variable is accessed by using the pointer variable with arrow operator(→) instead of period operator(.). The syntax to access the pointer to structure.

pointer_var_name→structure_member;

For Example:

emp→name;

Here “name” structure member is accessed through pointer variable emp.

Passing Structure to Function

The structure variable can be passed to a function as a parameter. The program to pass a structure variable to a function.

```
#include <stdio.h>
#include <conio.h>
struct employee
{
int  emp_id;
char name[20];
float salary;
};
```

Passing Structure to Function

```
void main ( )
  {
    struct employee e1;
    printf (“Enter the employee id of employee”);
    scanf(“%d”,&e1.emp_id);
    printf (“Enter the name of employee”);
    scanf(“%s”,e1.name);
    printf (“Enter the salary of employee”);
    scanf(“%f”,&e1.salary);
    printdata (struct employee e1);
    getch();
  }
```

Passing Structure to Function

```
void printdata( struct employee emp)
 {
   printf ("\nThe employee id of employee is :
            %d", emp.emp_id);
   printf ("\nThe name of employee is : %s",
            emp.name);
   printf ("\nThe salary of employee is : %f",
            emp.salary);
 }
```

Function Returning Structure

The function can return a variable of structure type like a integer and float variable. The program to return a structure from function.

```
#include <stdio.h>
#include <conio.h>
struct employee
{
int  emp_id;
char name[20];
float salary;
};
```

Function Returning Structure

```
void main ( )
  {
    struct employee emp;
    emp=getdata();
    printf (“\nThe employee id of employee is :
              %d”, emp.emp_id);
    printf (“\nThe name of employee is : %s”,
              emp.name);
    printf (“\nThe salary of employee is : %f”,
              emp.salary);
    getch();
  }
```

Function Returning Structure

```
struct employee getdata( )
 {
    struct employee e1;
    printf (“Enter the employee id of employee”);
    scanf(“%d”,&e1.emp_id);
    printf (“Enter the name of employee”);
    scanf(“%s”,e1.name);
    printf (“Enter the salary of employee”);
    scanf(“%f”,&e1.salary);
    return(e1);
  }
```

Union Data Type

A union is a user defined data type like structure. The union groups logically related variables into a single unit. The union data type allocate the space equal to space need to hold the largest data member of union. The union allows different types of variable to share same space in memory. There is no other difference between structure and union than internal difference. The method to declare, use and access the union is same as structure.

Defining of Union

A union has to defined, before it can used. The syntax of defining a structure is

```
union <union_name>
{
  <data_type> <variable_name>;
  <data_type> <variable_name>;
   ……..
  <data_type> <variable_name>;
};
```

Example of Union

The union of Employee is declared as

```
union employee
{
int  emp_id;
char name[20];
float salary;
char address[50];
int dept_no;
int age;
};
```

Memory Space Allocation

8000

emp_id, dept_no, age

8002

salary

8004

name

8022

address

Difference between Structures & Union

1)The memory occupied by structure variable is the sum of sizes of all the members but memory occupied by union variable is equal to space hold by the largest data member of a union.

2)In the structure all the members are accessed at any point of time but in union only one of union member can be accessed at any given time.

Application of Structures

Structure is used in database management to maintain data about books in library, items in store, employees in an organization, financial accounting transaction in company. Beside that other application are

1)Changing the size of cursor.

2)Clearing the contents of screen.

3)Drawing any graphics shape on screen.

4)Receiving the key from the keyboard.

Application of Structures

5) Placing cursor at defined position on screen.

6) Checking the memory size of the computer.

7) Finding out the list of equipments attach to computer.

8) Hiding a file from the directory.

9) Sending the output to printer.

10) Interacting with the mouse.

11) Formatting a floppy.

12) Displaying the directory of a disk.

Summary

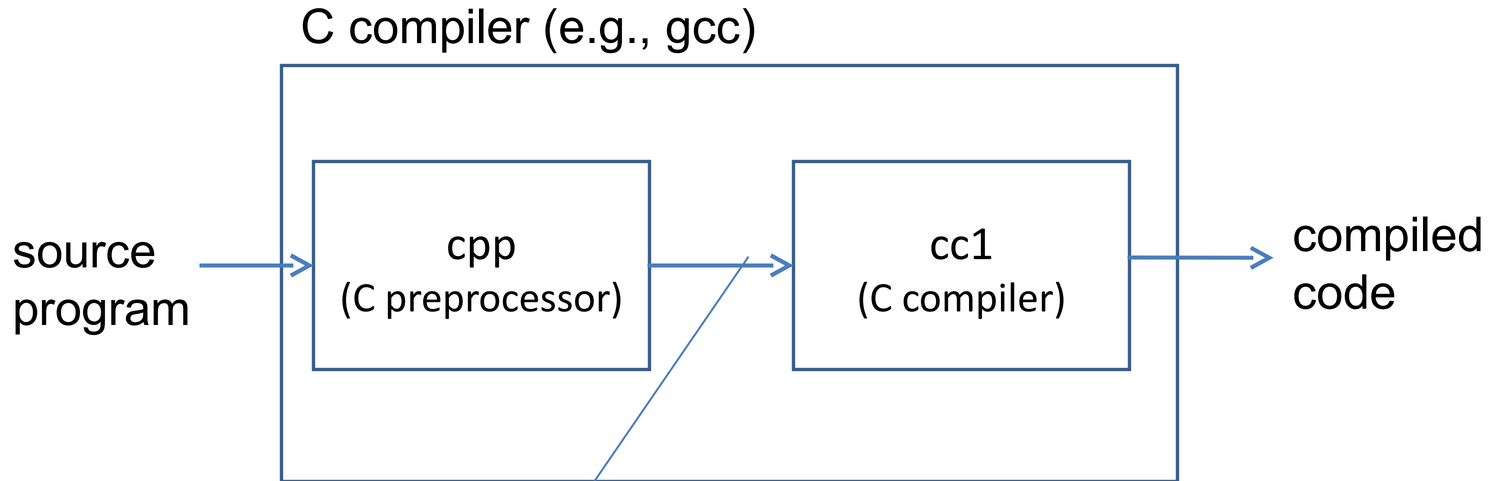
- A structure is a user defined data type that groups logically related data items of different data types into a single unit.
- The elements of a structure are stored at contiguous memory locations.
- The value of one structure variable is assigned to another variable of same type using assignment statement.
- An array of variables of structure is created.
- A variable of structure type is defined as a member of other structure type called nested structure.

Summary

- The member of structure variable is accessed by pointer variable with arrow operator (→).
- The structure variable can be passed to a function as a parameter.
- The function can return a variable of structure type.
- A union is like structure that group logically related variables into a single unit. The union allocate the space equal to space need to hold the largest data member of union.
- Structure used in database management and many more applications.

MACROS & EXAMPLES

The C preprocessor and its role



expanded code

- **expand some kinds of characters**
- **discard whitespace and comments**
  - each comment is replaced with a single space
- **process directives:**
  - file inclusion (**#include**)
  - macro expansion (**#define**)
  - conditional compilation **(#if**, **#ifdef**, …)

#include

- **Specifies that the preprocessor should read in the contents of the specified file**
  - usually used to read in type definitions, prototypes, etc.
  - proceeds recursively
    - **#includes in the included file are read in as well**
- **Two forms:**
  - #include <*filename*>
    - **searches for filename from a predefined list of directories**
    - **the list can be extended via "gcc –I *dir*"**
  - #include "*filename*"
    - **looks for *filename* specified as a relative or absolute path**

#include : Example

hed: /cs/www/classes/cs352/spring10/Code/ex.1.Preprocessor

```
% cat hello_2.c
#include <stdio.h>

int main( void )
{
  printf("hello\n");
  return 0;
}
%
% gcc -Wall hello_2.c
%
% ./a.out
hello
%
%
```

where does it come from?

– man 3 printf :

hed: /cs/www/classes/cs352/

```
PRINTF(3)                    Linux Programmer's

NAME
       printf,  fprintf,  sprintf, snprintf,
       matted output conversion

SYNOPSIS
       #include <stdio.h>

       int printf(const char *format, ...);
       int fprintf(FILE *stream, const char *
       int sprintf(char *str, const char *for
       int snprintf(char *str, size_t size, c

       #include <stdarg.h>

       int vprintf(const char *format, va_lis
       int vfprintf(FILE *stream, const char
       int vsprintf(char *str, const char *fo
       int vsnprintf(char *str, size_t size,
```

a predefined include file that:

- comes with the system
- gives type declarations, prototypes for library routines (printf)

#include: cont'd

- **We can also define our own header files:**
  - a header file has file-extension '**.h**'
  - these header files typically contain "public" information
    - **type declarations**
    - **macros and other definitions**
    - **function prototypes**
  - often, the public information associated with a code file **foo.c** will be placed in a header file **foo.h**
  - these header files are included by files that need that public information

    #include "myheaderfile.h"

Macros

- **A macro is a symbol that is recognized by the preprocessor and replaced by the macro body**

  - Structure of simple macros:

    ```
    #define  identifier  replacement_list
    ```

  - Examples:

    ```
    #define   BUFFERSZ    1024
    #define   WORDLEN   64
    ```

Using simple macros

- **We just use the macro name in place of the value, e.g.:**

```
#define  BUFLEN  1024
#define  Pi   3.1416
...
char buffer[BUFLEN];
...
area = Pi * r * r;
```

NOT:

```
#define  BUFLEN = 1024
#define  Pi  3.1416;
```

Example 1

hed: /cs/www/classes/cs352/spring10/Code/ex.7.Preprocessor

```
% cat preproc_0.c
/*
 * File: preproc_0.c
 *
 * A very simple use of the preprocessor.
 */

#include <stdio.h>

#define N  5
#define M 10
#define FOO printf

int main( )
{
  FOO("%d x %d = %d\n", M, N, M*N);

  return 0;
}
% gcc -Wall preproc_0.c
% ./a.out
10 x 5 = 50
%
```

Example 2

hed: /cs/www/classes/cs352/spring10/Code/ex.7.Preprocessor

```
% cat preproc_1.c
/*
 * File: preproc_1.c
 *
 * A simple use of the preprocessor: 2
 */

#include <stdio.h>

#define N  5
#define M 10
#define FOO printf

int main( )
{
  FOO("M = %d, N = %d, %d x %d = %d\n", M, N, M, N, M*N);
  printf("the decimal representation of %x [octal %o] is %d\n", 1000, 1000, 1000);

  return 0;
}
% gcc -Wall preproc_1.c
%
% ./a.out
M = 10, N = 5, 10 x 5 = 50
the decimal representation of 3e8 [octal 1750] is 1000
%
```

we can "macroize" symbols selectively

Parameterized macros

- **Macros can have parameters**
  - these resemble functions in some ways:
    - **macro definition ~ formal parameters**
    - **macro use ~ actual arguments**
  - Form:

    #define macroName(arg$_1$, …, arg$_n$) replacement_list
  - Example:

```
#define deref(ptr)       *ptr
#define MAX(x,y)         x > y ? x : y
```

no space here!
(else preprocessor will assume we're defining a simple macro

Example

hed: /cs/www/classes/cs352/spring10/Code/ex.7.Preprocessor

```
% cat preproc_2.c
/*
 * File: preproc_2.c
 *
 * A simple use of the preprocessor: 3
 * This example shows macros that take arguments
 */

#include <stdio.h>

#define double(x)  x+x

int main( )
{
  printf("double %d is %d\n", 10, double(10));

  return 0;
}
% gcc -Wall preproc_2.c
% ./a.out
double 10 is 20
%
%
```

Macros vs. functions

- **Macros may be (slightly) faster**
  - don't incur the overhead of function call/return
  - however, the resulting code size is usually larger
    - **this can lead to loss of speed**
- **Macros are "generic"**
  - parameters don't have any associated type
  - arguments are not type-checked
- **Macros may evaluate their arguments more than once**
  - a function argument is only evaluated once per call

Macros vs. Functions: Argument Evaluation

- **Macros and functions may behave differently if an argument is referenced multiple times:**
  - a function argument is evaluated once, before the call
  - a macro argument is evaluated each time it is encountered in the macro body.
- **Example:**

int dbl(x) { return x + x;} **…** **u = 10; v = dbl(u++);** **printf("u = %d, v = %d", u, v);** *prints*: **u = 11, v = 20**	**#define Dbl(x) x + x** **…** **u = 10; v = Dbl(u++);** **printf("u = %d, v = %d", u, v);** *prints*: **u = 12, v = 21**

Dbl(u++)
expands to:
u++ + u++

Properties of macros

- **Macros may be nested**
  - in definitions, e.g.:

```
#define  Pi          3.1416
#define  Twice_Pi    2*Pi
```

  - in uses, e.g.:

```
#define  double(x)    x+x
#define  Pi  3.1416
…
if ( x > double(Pi) ) …
```

- **Nested macros are expanded recursively**

Header Files

- **Have a file extension ".h"**
- **Contain shared definitions**
  - typedefs
  - macros
  - function prototypes
- **referenced via "#include" directives**

Header files: example

hed: /cs/www/classes/cs352/spring10/Code/ex.7.Preprocessor

```
% cat preproc_5.c
/*
 * File: preproc_5.c
 *
 * A simple use of the preprocessor: header files
 */

#include "preproc_5.h"

int main( )
{
  int x = 3;
  printf("double of %d squared is: %d\n", x, double(square(x)));
  printf("square of %d doubled is: %d\n", x, square(double(x)));

  return 0;
}
%
%
% cat preproc_5.h
/*
 * File: preproc_5.h
 */

#include <stdio.h>

#define double(x)  (x)+(x)
#define square(x)  (x)*(x)

%
% gcc -Wall preproc_5.c
%
% ./a.out
double of 3 squared is: 18
square of 3 doubled is: 36
%
```

typedefs

- **Allow us to define *aliases* for types**
- **Syntax:**

  typedef old_type_name new_type_name;

  - **new_type_name becomes an alias for old_type_name**
- **Example:**
  - typedef int BasePay;
  - typedef struct node {

```
typedef struct node {
    int value;
    struct node *next;
} node;
```

Example

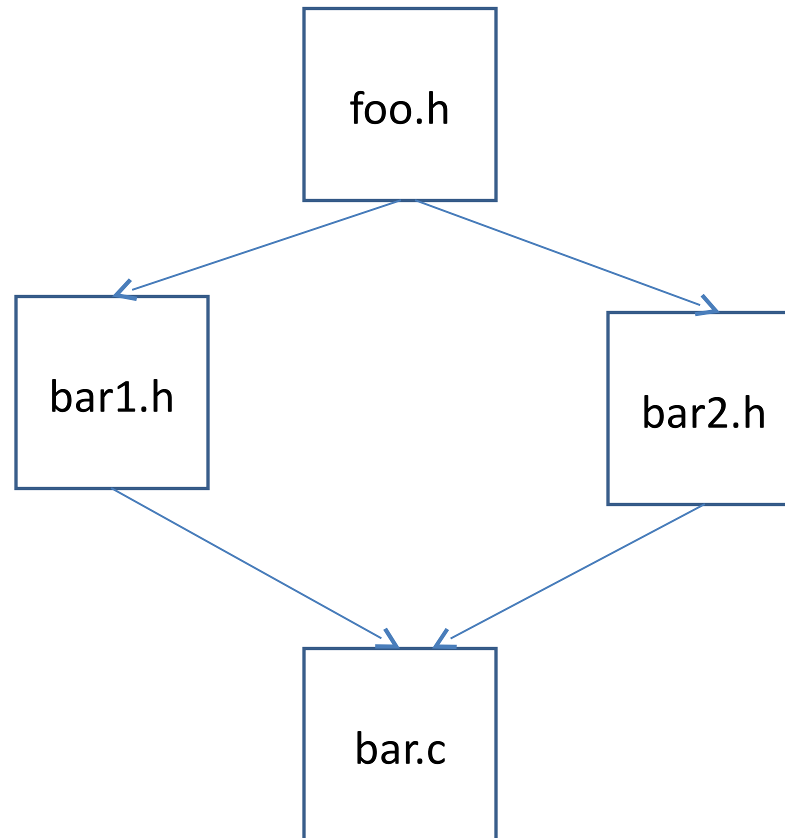
```
hed: /cs/www/classes/cs352/spring10/Code/ex.7.Preprocessor
% cat wordcount.h
/*
 * File: wordcount.h
 */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h>

typedef struct wc {
  char *word;
  int count;
  struct wc *next;
} wcnode;

% grep wcnode wordcount.c
wcnode *listHd = NULL;
  wcnode *wtmp;
    wtmp = malloc(sizeof(wcnode));
  wcnode *wtmp0, *wtmp1;
  wcnode *wtmp;
%
% gcc -Wall wordcount.c
% ./a.out < wordcount.h | head -7
char 1
count 1
ctype 1
file 1
h 6
include 5
int 1
%
```

defines “wcnode” as an alias for “struct wc”

we can use “wcnode” in place of“struct wc”

but not here, since “wcnode” has not yet been defined

What if a file is #included multiple times?



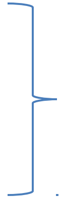
Conditional Compilation: #ifdef

#ifdef *identifier*

line$_1$

…

line$_n$



line1 … line$_n$ will be included if *identifier* has been defined as a macro; otherwise nothing will happen.

#endif

- *macros can be defined by the compiler:*
  - **gcc –D** *macroName*
  - **gcc –D** *macroName=definition*
- *macros can be defined without giving them a specific value, e.g.:*
  - *#define macroName*

Conditional Compilation: #ifndef

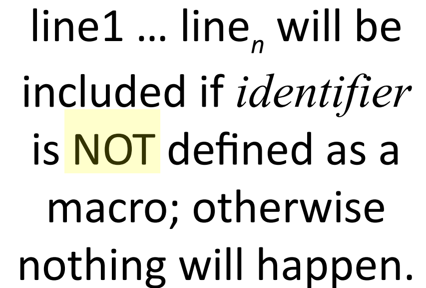
#ifndef *identifier*

line1

...

line$_n$

#endif



Solution to multiple inclusion problem

The header file is written as follows:

```
#ifndef file_specific_flag
#define file_specific_flag
...contents of file...
#endif
```

indicates whether or not this file has been included already

- ***file_specific_flag*** **usually constructed from the name of the header file:**

  E.g.: file = **foo.h** ⇒ flag = **_FOO_H_**

  – try to avoid macro names starting with '_'

Another use of #ifdefs

- **They can be useful for controlling debugging output**
  - Example 1: guard debugging code with #ifdefs:

```
#ifdef   DEBUG
…debug message…
#endif
```

straightforward, but needs discipline to use consistently

  - Example 2: use the debug macro to control what debugging code appears in the program:

```
#ifdef   DEBUG
#define  DMSG(msg)    printf(msg)        // debugging output
#else
#define DMSG(msg)     {}                 // empty statement
#endif
```

Generalizing #ifdef

#if ***constant-expression***

line$_1$

…

line$_n$

#endif

> **⇒ line$_1$ … line$_n$ included if *constant-expression* evaluates to a non-zero value**

Common uses:
- #if 1

or
- #if 0

Predefined Macros

`__LINE__`	*current line number of the source file*
`__FILE__`	*name of the current source file*
`__TIME__`	*time of translation*
`__STDC__`	*1 if the compiler conforms to ANSI C*

```
printf("working on %s\n", __FILE__);
```