Miscellaneous

Structured Programming Language (CSE-1271)

Course Instructor : Mohammed Mamun Hossain
Assistant Professor, Dept. of CSE, BAUST

Outline

1. Storage class
2. scope of variables
3. Typedef
4. Preprocessors
5. Memory Management

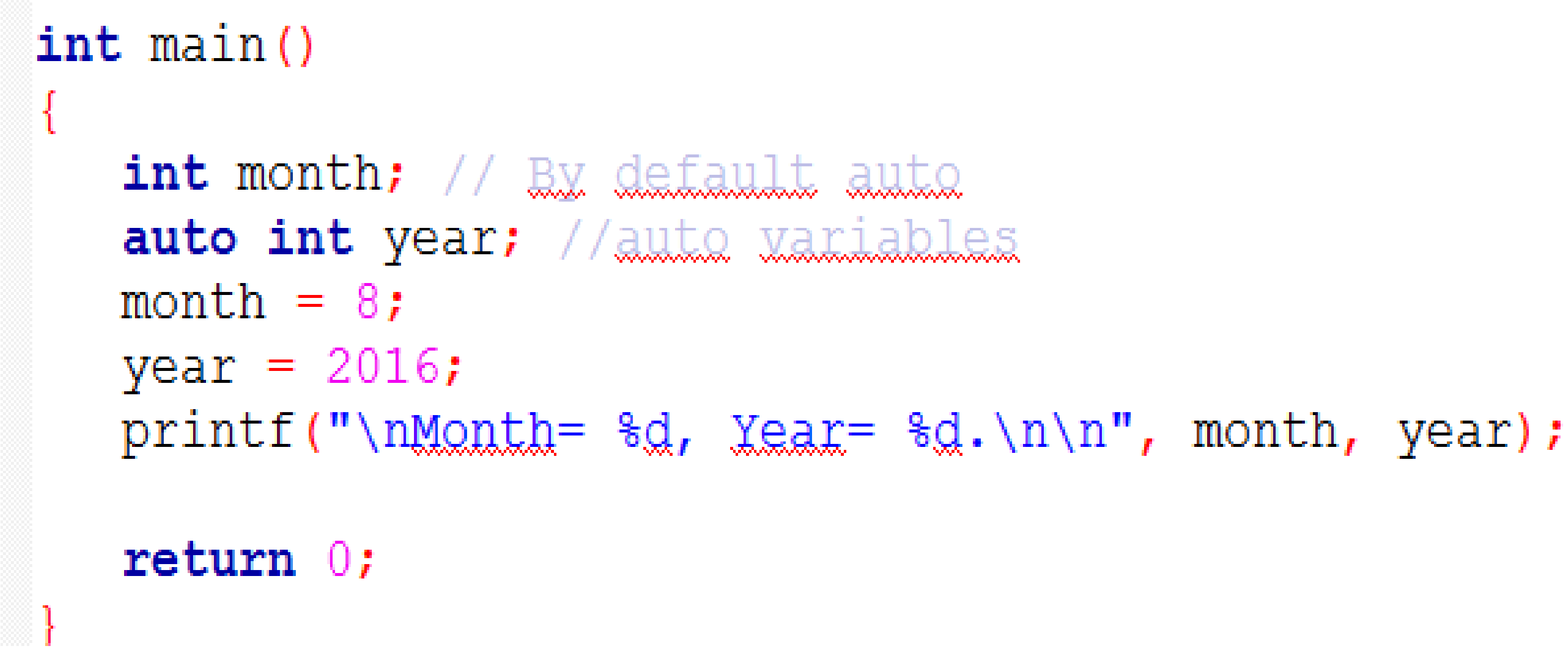
Storage Class

- Storage class define
  - ✓ Scope,
  - ✓ Visibility and
  - ✓ Lifetime of the variable

- Storage classes are
  - ✓ Automatic (auto)
  - ✓ Static variables (static)
  - ✓ Register variables (register)
  - ✓ External variables (extern)

Storage Class | Automatic variables

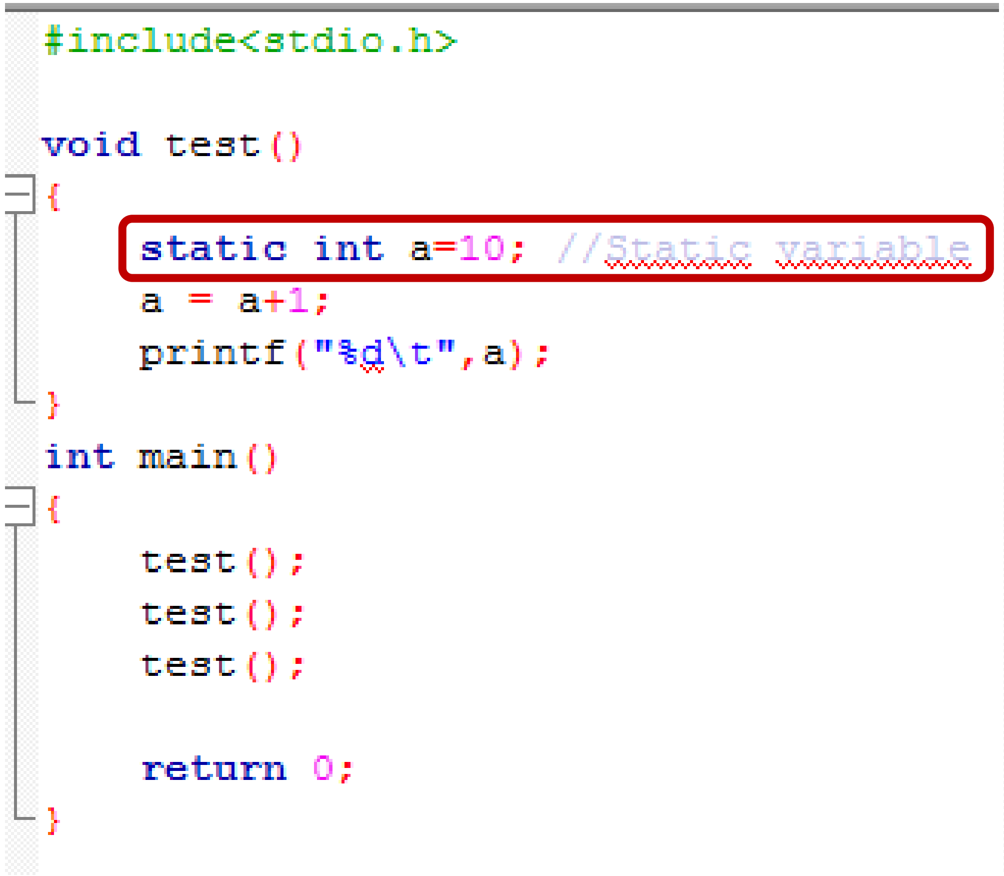
- Declared inside a function only
- It’s default variable
- Created when a function is called
- Destroyed automatically when the function exits
- It’s called local variables
- By default they are assigned garbage value by the compiler



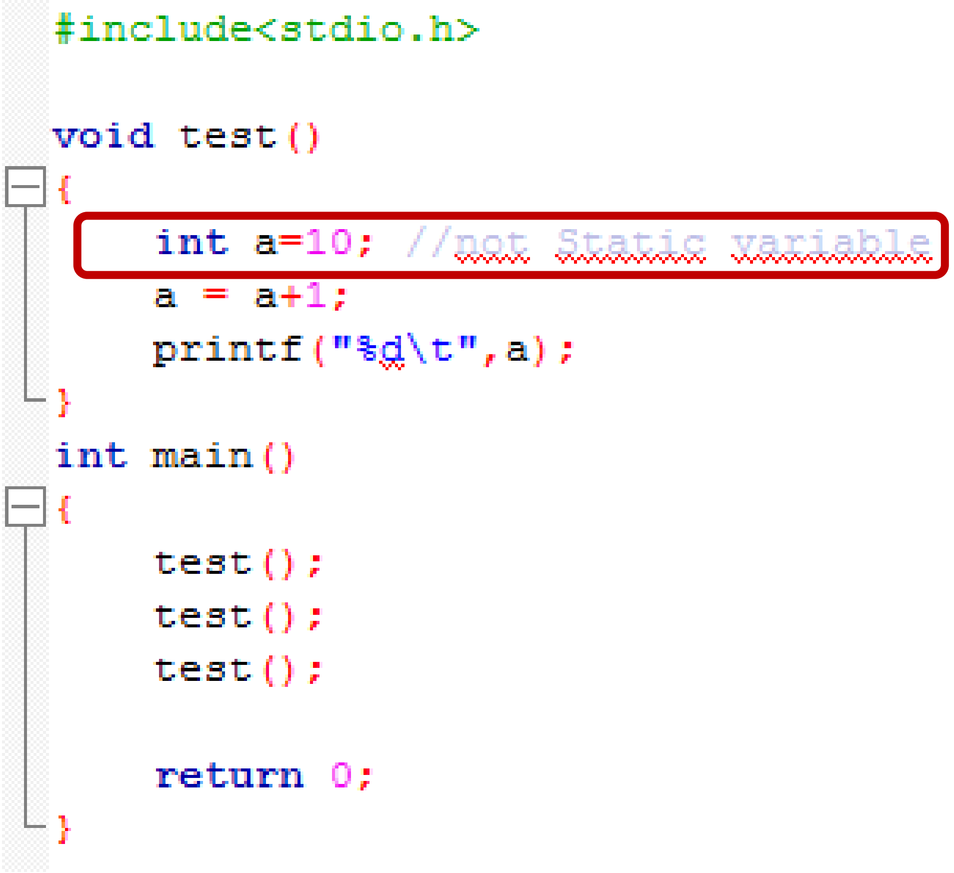
```
int main()
{
    int month; // By default auto
    auto int year; //auto variables
    month = 8;
    year = 2016;
    printf("\nMonth= %d, Year= %d.\n\n", month, year);

    return 0;
}
```

Storage Class | Static Variables

- ❖ Tells the compiler to persist the variable until the end of program.
- ❖ A static variable can either be internal or external
- ❖ They are assigned 0 (zero) as default value by the compiler.



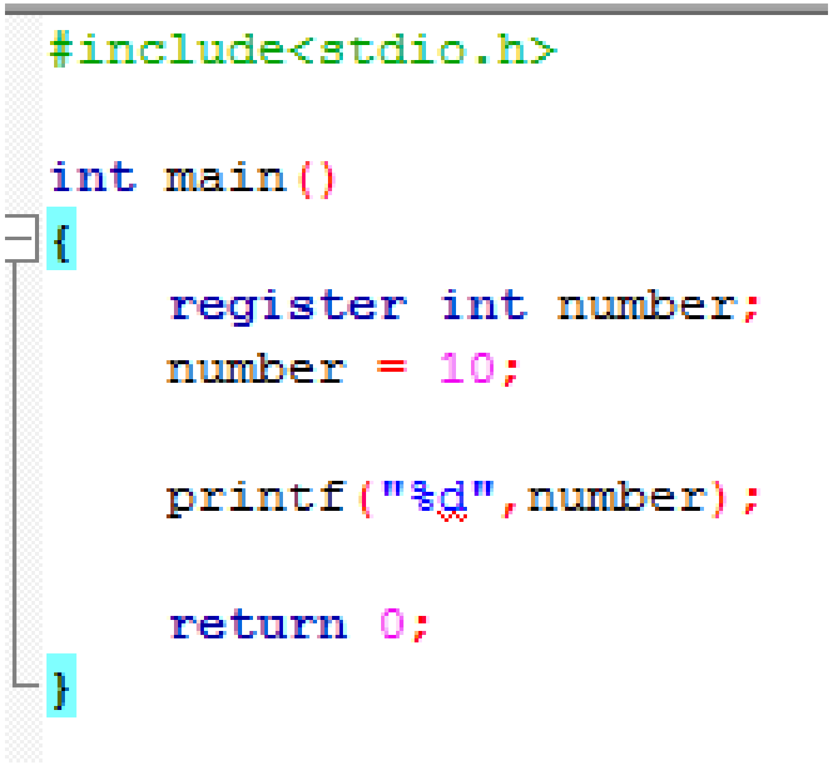
```
#include<stdio.h>

void test()
{
    static int a=10; //Static variable
    a = a+1;
    printf("%d\t",a);
}
int main()
{
    test();
    test();
    test();

    return 0;
}
```



```
#include<stdio.h>

void test()
{
    int a=10; //not Static variable
    a = a+1;
    printf("%d\t",a);
}
int main()
{
    test();
    test();
    test();

    return 0;
}
```

Storage Class | Register Variables

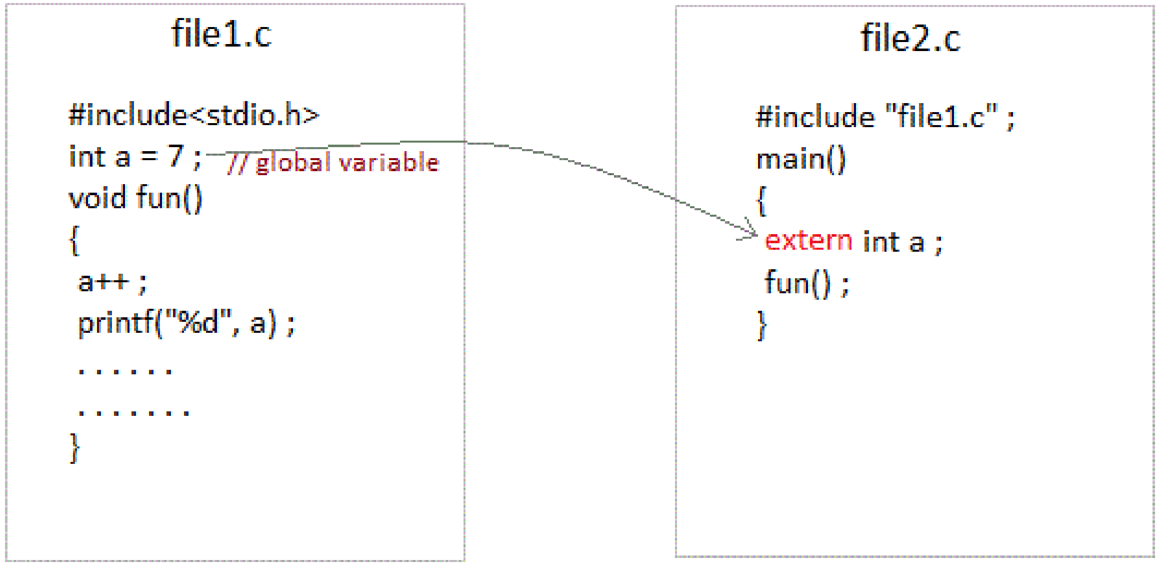
- To store the variable in register instead of memory
- Register variable has faster access than normal variable
- Only few variables can be placed inside register
- We can never get the address of such variables



```c
#include<stdio.h>

int main()
{
    register int number;
    number = 10;

    printf("%d",number);

    return 0;
}
```

Storage Class | External Variables

- The extern keyword is used before a variable to inform the compiler that this variable is declared somewhere else.
- extern is used to declare a global variable or function in another file.
- When use 'extern', the variable cannot be initialized
- The extern declaration does not allocate storage for variables.



file1.c

```
#include<stdio.h>
int a = 7 ; // global variable
void fun()
{
 a++ ;
 printf("%d", a) ;
 . . . . . .
 . . . . . . .
}
```

file2.c

```
#include "file1.c" ;
main()
{
 extern int a ;
 fun() ;
}
```

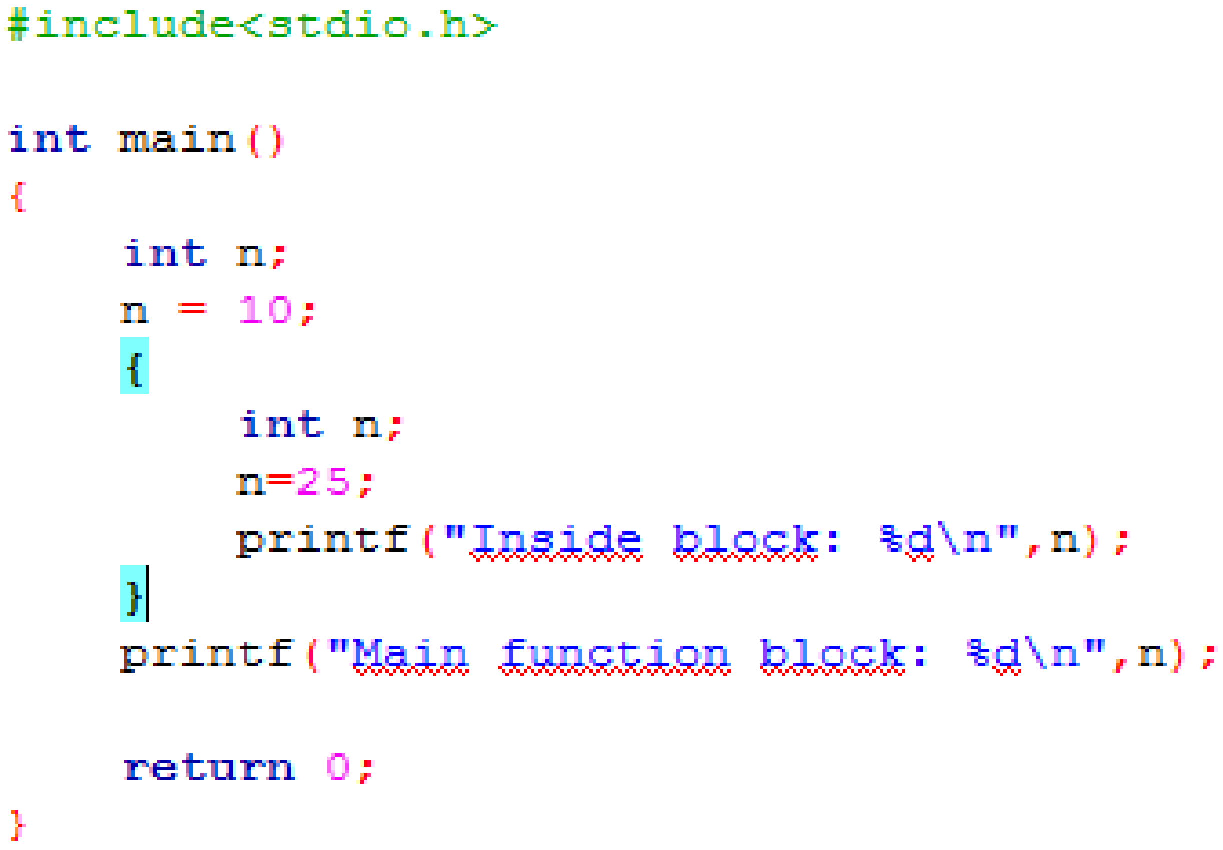
global variable from one file can be used in other using **extern** keyword.

Scope of Variables

- A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed.
- There are three places where variables can be declared:
  - ✓ Inside a function or a block (local variables).
  - ✓ Outside of all functions (global variables).
  - ✓ In the definition of function parameters (formal parameters).

Scope of Variables | Local Variables

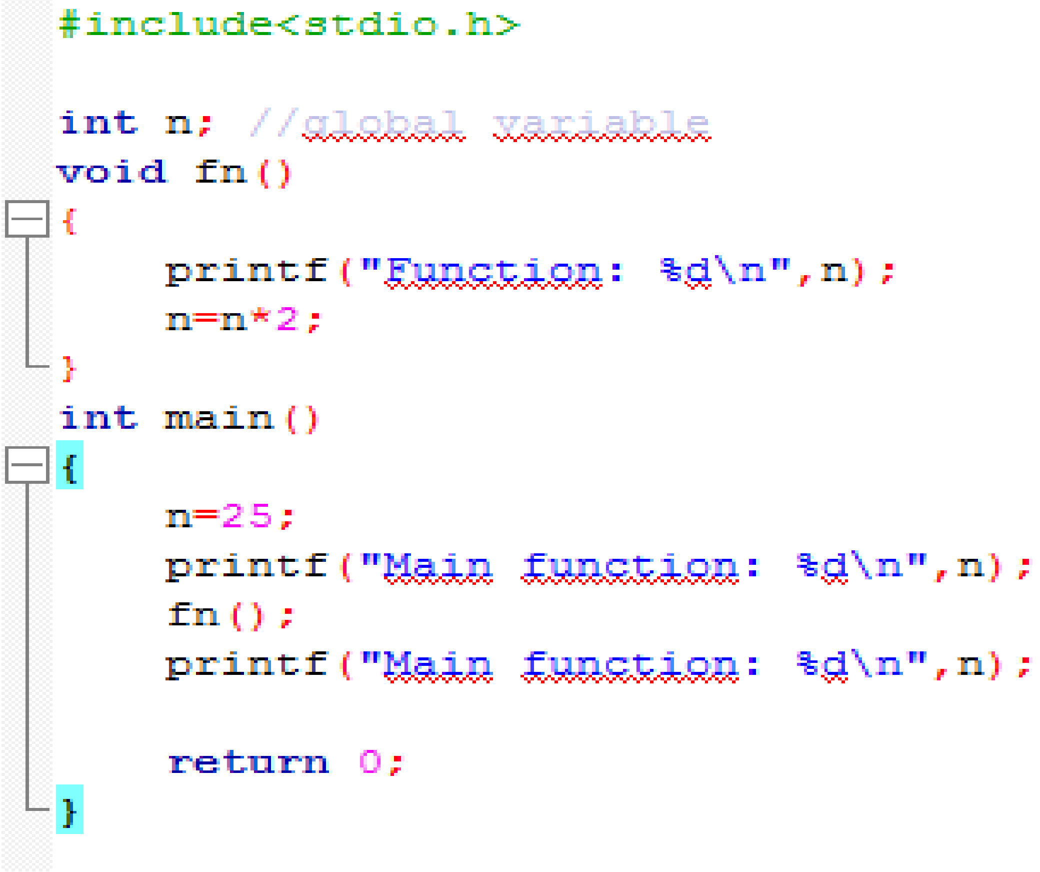
- Declared inside a function or block.
- Can be used only inside that function or block.
- Not known to functions outside their own.



```c
#include<stdio.h>

int main()
{
    int n;
    n = 10;
    {
        int n;
        n=25;
        printf("Inside block: %d\n",n);
    }
    printf("Main function block: %d\n",n);

    return 0;
}
```

Scope of Variables | Global Variables

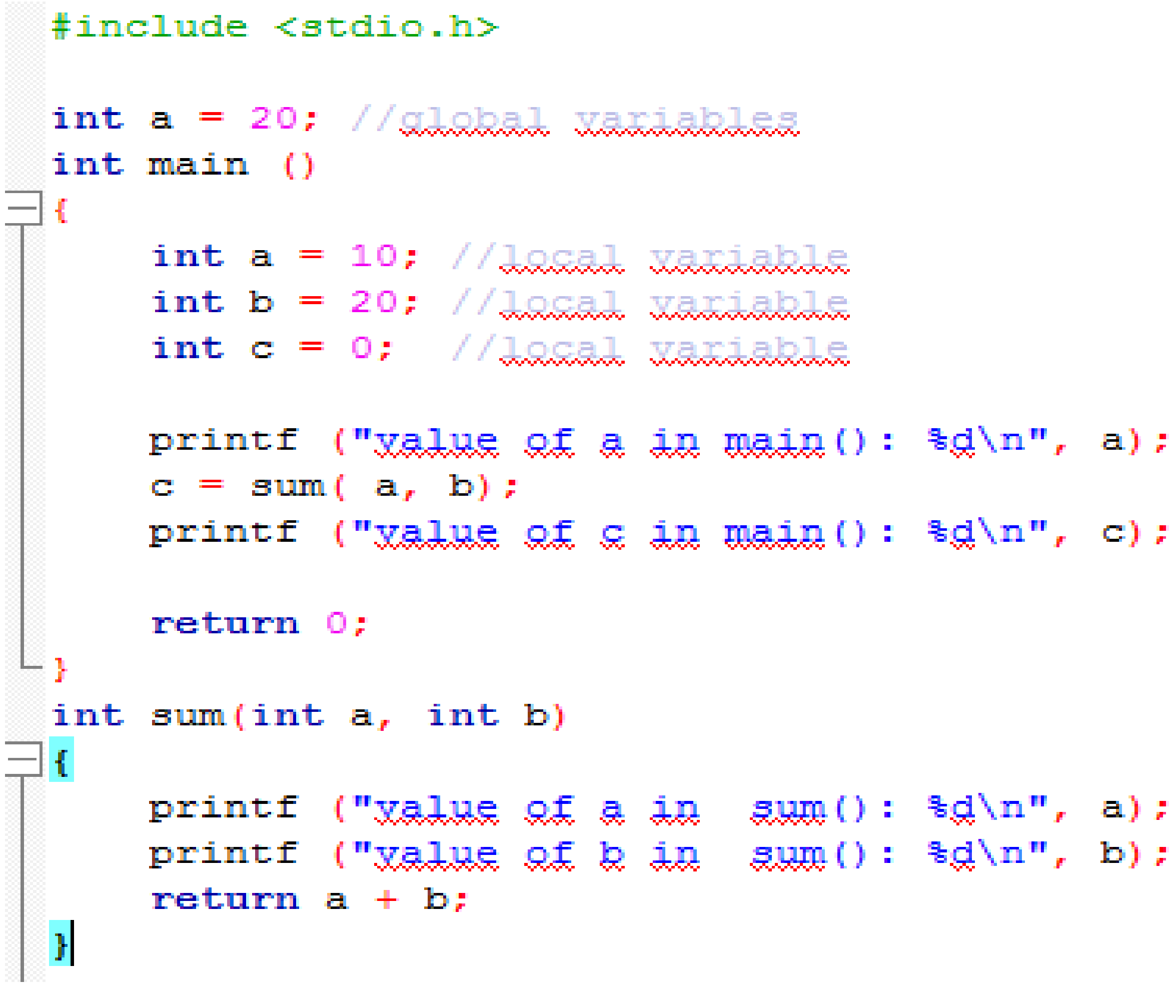
- Defined outside a function, usually on top of the program.
- Hold their values throughout the lifetime of program.
- Available for use throughout entire program after its declaration.



```
#include<stdio.h>

int n; //global variable
void fn()
{
    printf("Function: %d\n",n);
    n=n*2;
}
int main()
{
    n=25;
    printf("Main function: %d\n",n);
    fn();
    printf("Main function: %d\n",n);

    return 0;
}
```

Scope of Variables | Formal Parameters

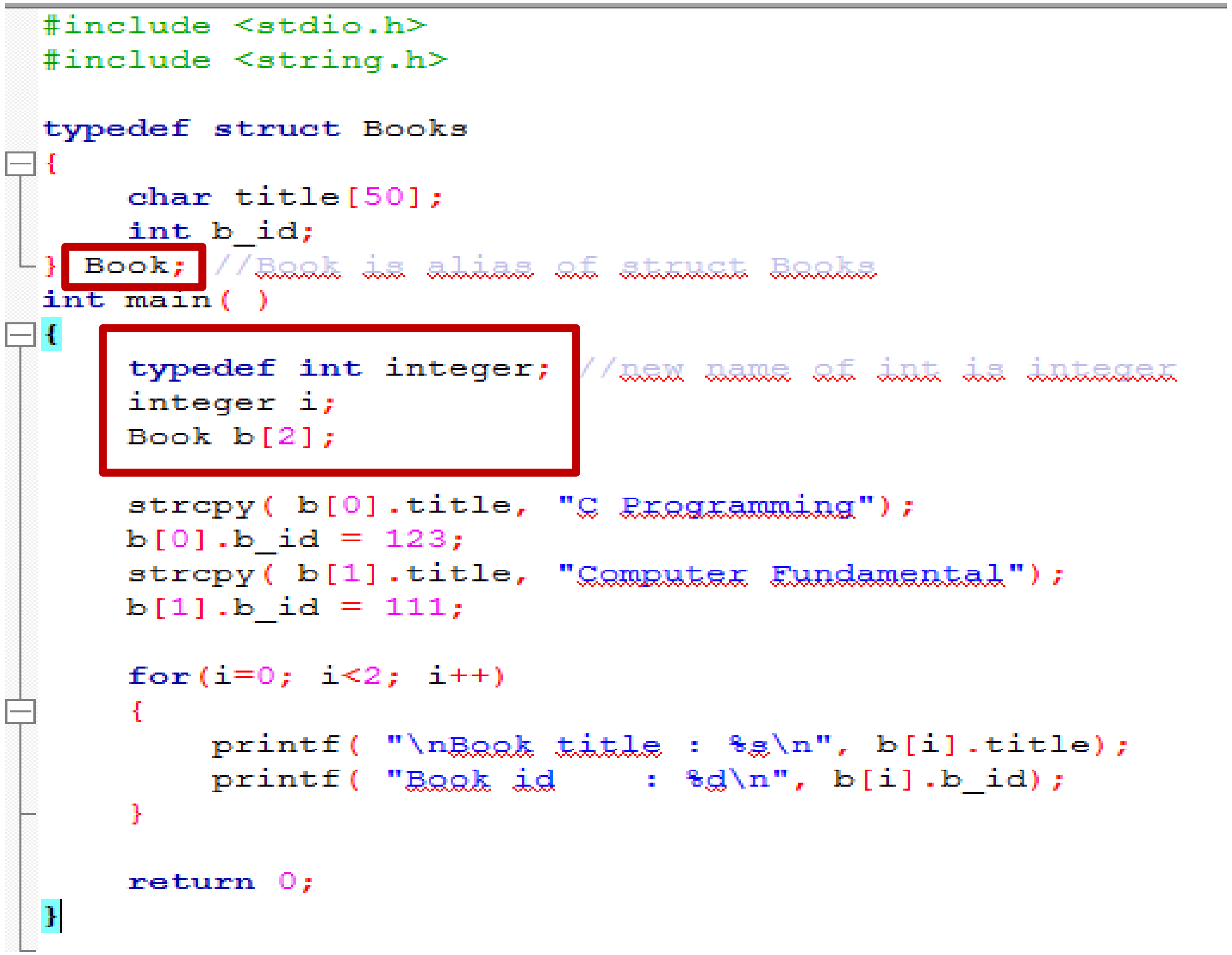
- ❖Treated as local variables with-in a function
- ❖They take precedence over global variables



```
#include <stdio.h>

int a = 20; //global variables
int main ()
{
    int a = 10; //local variable
    int b = 20; //local variable
    int c = 0;  //local variable

    printf ("value of a in main(): %d\n", a);
    c = sum( a, b);
    printf ("value of c in main(): %d\n", c);

    return 0;
}
int sum(int a, int b)
{
    printf ("value of a in  sum(): %d\n", a);
    printf ("value of b in  sum(): %d\n", b);
    return a + b;
}
```

Typedef

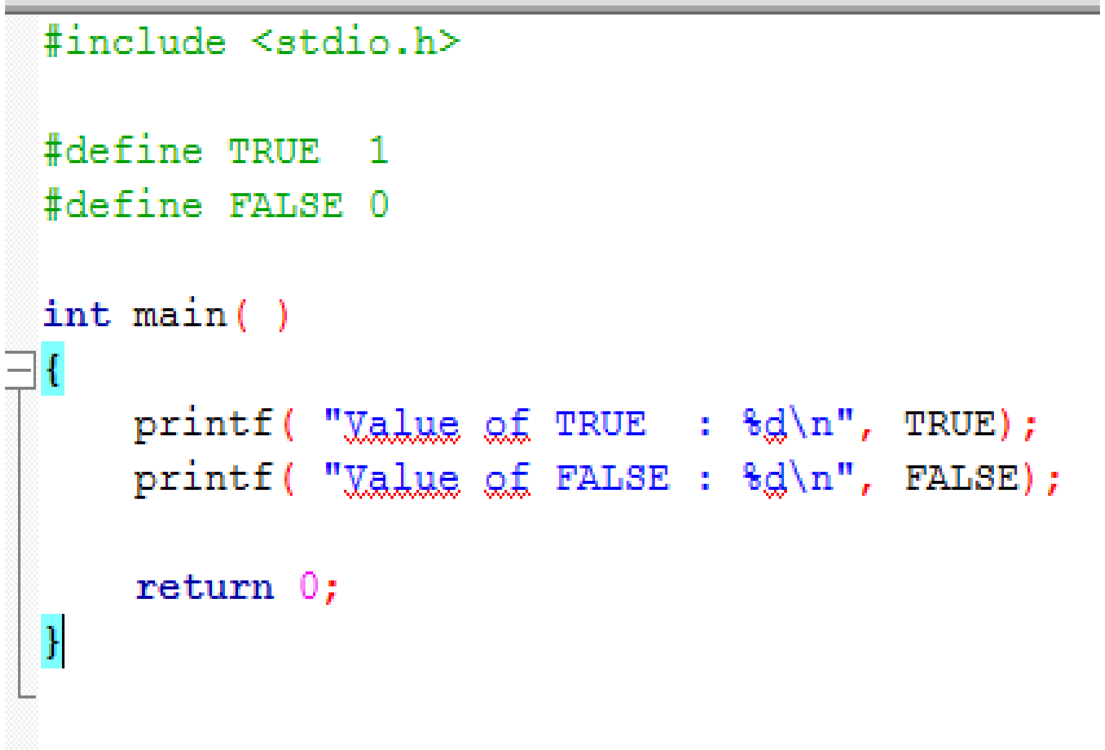
- typedef is a keyword used in C language to assign alternative names to existing types. Its mostly used with user defined data types, when names of data types get slightly complicated.



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

typedef struct Books
{
        char title[50];
        int b_id;
} Book; //Book is alias of struct Books
int main( )
{
        typedef int integer; //new name of int is integer
        integer i;
        Book b[2];

        strcpy( b[0].title, "C Programming");
        b[0].b_id = 123;
        strcpy( b[1].title, "Computer Fundamental");
        b[1].b_id = 111;

        for(i=0; i<2; i++)
        {
                printf( "\nBook title : %s\n", b[i].title);
                printf( "Book id     : %d\n", b[i].b_id);
        }

        return 0;
}
```

Typedef vs #define

- typedef is limited to giving symbolic names to types only where as #define can be used to define alias for values.
- typedef interpretation is performed by the compiler whereas #define statements are processed by the pre-processor.



```c
#include <stdio.h>

#define TRUE  1
#define FALSE 0

int main( )
{
    printf( "Value of TRUE  : %d\n", TRUE);
    printf( "Value of FALSE : %d\n", FALSE);

    return 0;
}
```

Preprocessor

- C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation.

Directive	Description
#define	Substitutes a preprocessor macro.
#include	Inserts a particular header from another file.
#undef	Undefines a preprocessor macro.
#ifdef	Returns true if this macro is defined.
#ifndef	Returns true if this macro is not defined.
#if	Tests if a compile time condition is true.
#else	The alternative for #if.
#elif	#else and #if in one statement.
#endif	Ends preprocessor conditional.
#error	Prints error message on stderr.
#pragma	Issues special commands to the compiler, using a standardized method.

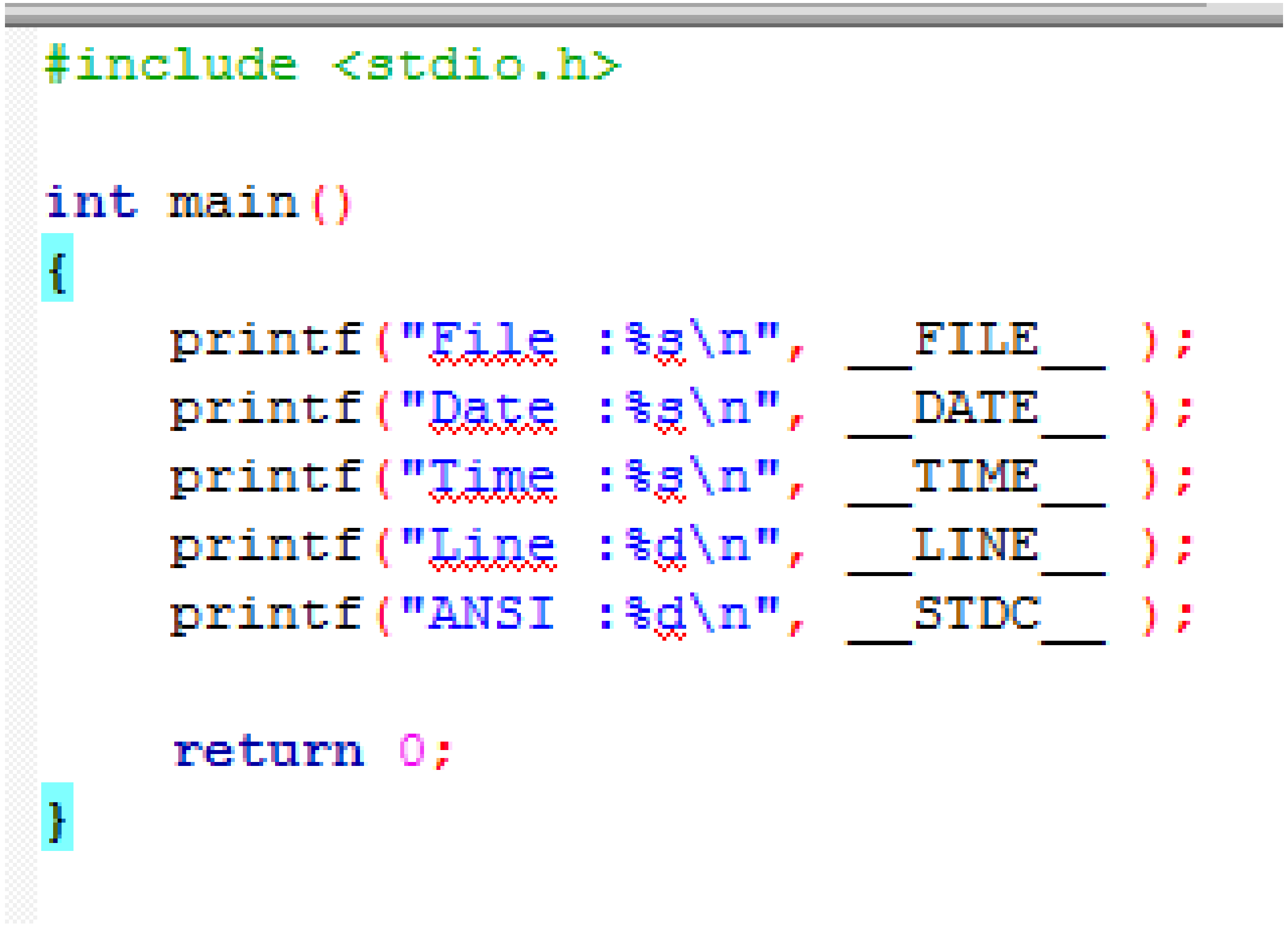
Preprocessor

- Predefined Macros: ANSI C defines a number of macros. Although each one is available for use in programming, the predefined macros should not be directly modified.

Macro	Description
`__DATE__`	The current date as a character literal in "MMM DD YYYY" format.
`__TIME__`	The current time as a character literal in "HH:MM:SS" format.
`__FILE__`	This contains the current filename as a string literal.
`__LINE__`	This contains the current line number as a decimal constant.
`__STDC__`	Defined as 1 when the compiler complies with the ANSI standard.

Preprocessor

- Predefined Macros: ANSI C defines a number of macros. Although each one is available for use in programming, the predefined macros should not be directly modified.



```c
#include <stdio.h>

int main()
{
    printf("File :%s\n", __FILE__ );
    printf("Date :%s\n", __DATE__ );
    printf("Time :%s\n", __TIME__ );
    printf("Line :%d\n", __LINE__ );
    printf("ANSI :%d\n", __STDC__ );

    return 0;
}
```

Preprocessor

- Below is the list of preprocessor directives that C language offers.

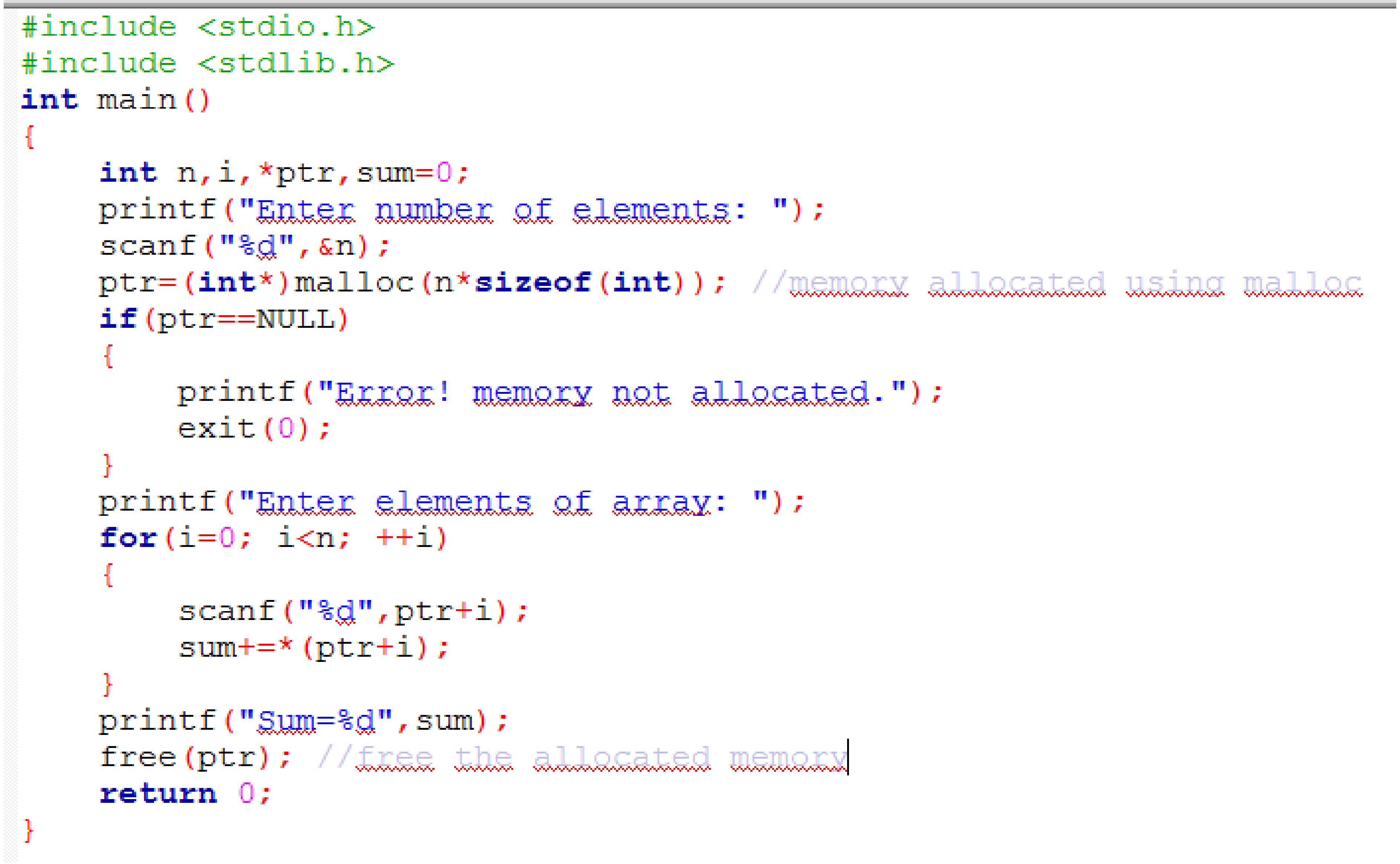
SN.	Preprocessor	Syntax	Description
1	Macro	`#define`	This macro defines constant value and can be any of the basic data types.
2	Header file inclusion	`#include<file_name>`	The source code of the file “file_name” is included in the main program at the specified place
3	Conditional compilation	`#ifdef, #endif, #if, #else, #ifndef`	Set of commands are included or excluded in source program before compilation with respect to the condition
4	Other directives	`#undef, #pragma`	#undef is used to undefine a defined macro variable. #Pragma is used to call a function before and after main function in a C program

Memory Management

- The C programming language provides several functions for memory allocation and management. These functions can be found in the <stdlib.h> header file.

Function	Use of Function
`malloc()`	Allocates requested size of bytes and returns a pointer first byte of allocated space
`calloc()`	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
`free()`	Dellocate the previously allocated space
`realloc()`	Change the size of previously allocated space

Memory Management



```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)malloc(n*sizeof(int)); //memory allocated using malloc
    if(ptr==NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0; i<n; ++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr); //free the allocated memory
    return 0;
}
```

Thank You.

Questions and Answer

References

Books:

1. Programming in ANSI C *By E. Balagurusamy*
2. Teach yourself C. *By Herbert Shield*
3. Programming With C. *By Byron Gottfried*

Web:

1. www.wikbooks.org

and other slide, books and web search.