C Inline Functions, Static Assertions, Structures, Keywords

Lecture 05

Inline Functions:

The inline function can be substituted at the place where the function call is happening. Function substitution is always compiler choice.

- In an inline function, a function call is replaced by the actual program code.
- Most of the Inline functions are used for small computations. They are not suitable for large computing.
- An inline function is similar to a normal function. The only difference is that we place a keyword inline before the function name.

Syntax:

Inline functions are created with the following syntax −

```
inline function_name (){
   //function definition
 }
```

Example:

Following is the C program for inline functions −

```
#include<stdio.h>
inline int mul(int a, int b) //inline function declaration{
   return(a*b);
}
int main(){
   int c;
   c=mul(2,3);
   printf("Multiplication:%d\n",c);
   return 0;
}
```

Output

When the above program is executed, it produces the following result “6” .

Static Assertions:

Static assertions are used to check if a condition is true when the code is compiled. If it isn't, the compiler is required to issue an error message and stop the compiling process.

A static assertion is one that is checked at compile time, not run time. The condition must be a constant expression, and if false will result in a compiler error. The first argument, the condition that is checked, must be a constant expression, and the second a string literal.

Unlike assert, `_Static_assert` is a keyword. A convenience macro `static_assert` is also defined in `assert.h` header file. Static assertion is only available in C11 version of C.

Syntax:

```
static_assert(expression, message)
"or"
_Static_assert(expression, message)
```

Parameters:

- *expression* - expression of scalar type.
- *message* - string literal to be included in the diagnostic message.

Example:

```
#include <assert.h>

enum
{
    N = 5
};

_Static_assert(N == 5, "N does not equal 5");
static_assert(N > 10, "N is not greater than 10");  /* compiler error */
```

Structures:

Structure in c is a user-defined data type that enables us to store the collection of different data types. Each element of a structure is called a member. Structures ca; simulate the use of classes and templates as it can store various information

The ,**struct** keyword is used to define the structure.

Syntax:

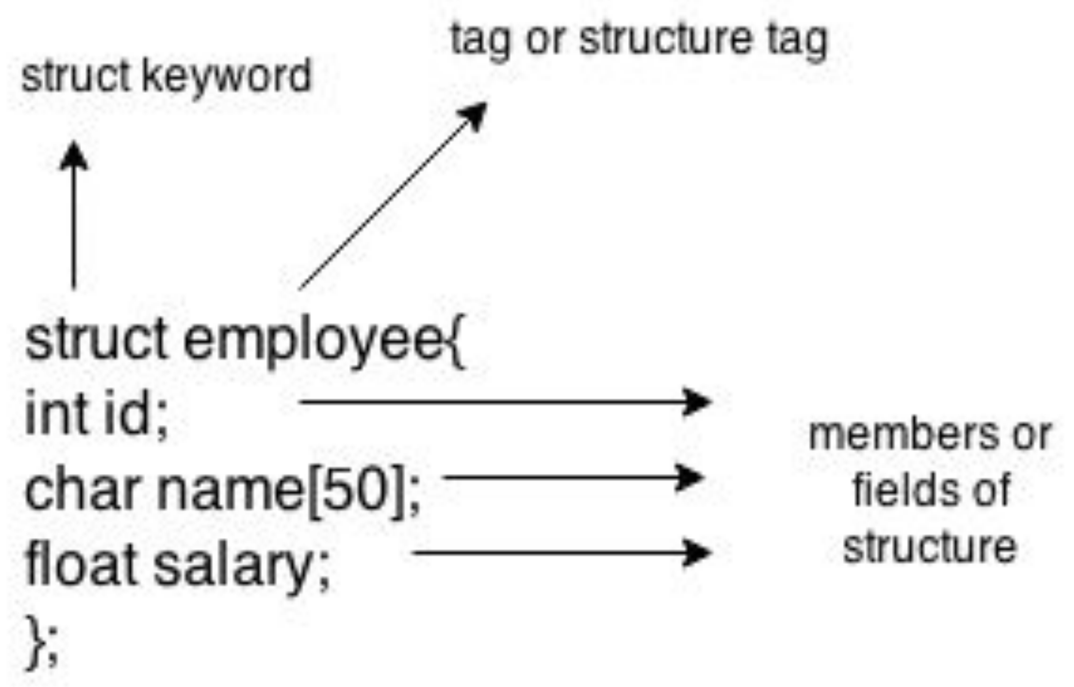
```
struct structure_name

{

   data_type member1;

   ….

   data_type memeberN;

};
```

Example:

```
struct employee
{   int id;
    char name[20];
    float salary;
};
```

Here, **struct** is the keyword; **employee** is the name of the structure; **id**, **name**, and **salary** are the members or fields of the structure. Let's understand it by the diagram given below

Further Elaboration:



```
struct employee{
int id;
char name[50];
float salary;
};
```

Static Keyword:

Static is a keyword used in C programming language. It can be used with both variables and functions, i.e., we can declare a static variable and static function as well. An ordinary variable is limited to the scope in which it is defined, while the scope of the static variable is throughout the program.

Usage:

Static keyword can be used in the following situations:
Static global variable
Static function
Static local variable
Static member variables
Static method

Example:

Without Static:

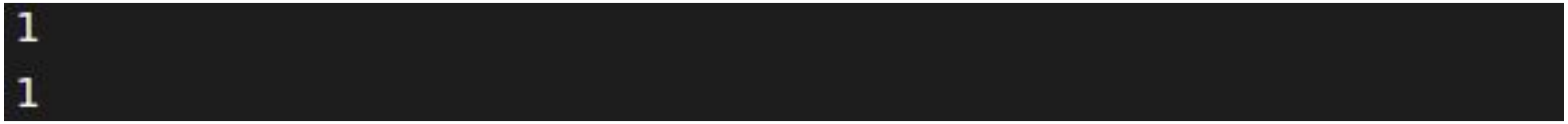
```
#include <stdio.h>

int main()

{

 printf("%d",func());

printf("\n%d",func());

 return 0;

}
```

Example Continued:

```
int func()

{

    int count=0; // variable initialization

    count++; // incrementing counter variable


   return count; }
```

Output:

The output would be as given below.



```
1
1
```

Example:

With Static:

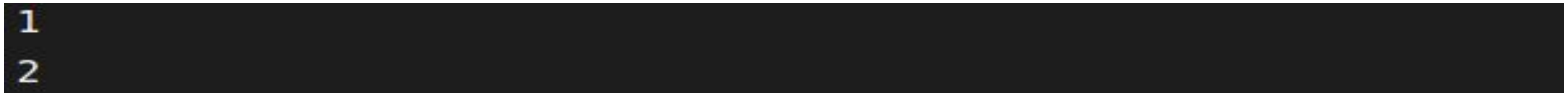
```
#include <stdio.h>

int main()

{

 printf("%d",func());

printf("\n%d",func());

 return 0;

}
```

Example Continued:

```
int func()

{

    static int count=0; // variable initialization

    count++; // incrementing counter variable


   return count; }
```

Output:

The output would be as given below.



```
1
2
```

Volatile Keyword:

A volatile keyword in C is nothing but a qualifier that is used by the programmer when they declare a variable in source code. It is used to inform the compiler that the variable value can be changed any time without any task given by the source code. Volatile is usually applied to a variable when we are declaring it.

Syntax:

```
volatile data_type variable_name ;
volatile data_type  *variable_name ;
```

Example:

Without Volatile:

```
#include<stdio.h> // C header file for standard input and output

int a = 0 ; // initilaizing and declaring the integer a to value

0.

int main ()  // main class

{

if ( a == 0 )  //  This condition will be true

{

printf ( " a = 0  \n " ) ;

}
```

Example Continued:

```
else                          // Else part will be optimized
{
printf ( " a ! = 0  \n " ) ;
}
return 0 ; // returning value
}
```

Output:

```
a=0
```

Example:

With Volatile:

```
#include<stdio.h>
volatile int a ;      /* volatile Keyword used before declaration of
integer variable a */
int main() // main class
{
a = 0 ;   // initializing the integer value to 0
if (a == 0)  // applying if condition
{
printf ( " a = 0 \n " ) ;
}
else// Now compiler never optimize else part because the variable is
declared as volatile
{
printf ( " a ! = 0  \n " ) ;
}
```

Example Continued:

```
return 0 ;
}
```

Output:

```
a=0
```

Extern Keyword:

Extern is a keyword in C programming language which is used to declare a global variable that is a variable without any memory assigned to it. It is used to declare variables and functions in header files. Extern can be used access variables across C files.

Syntax:

```
extern <data_type> <variable_name>;
// or
 extern <return_type> <function_name>(<parameter_list>);
```

Example:

```
extern int opengenus;
// or
 extern int opengenus_fn(int, float );
```

Const Keyword:

The qualifier const can be applied to the declaration of any variable to specify that its value will not be changed (Which depends upon where const variables are stored, we may change the value of const variable by using pointer).

Example:

For example, pointer to a const variable is given as below.

```
const int *ptr;
```

Define Keyword:

The #define preprocessor directive is used to define constant or micro substitution. It can use any basic data type.

Syntax:

```
#define token value
```

Example:

#define to define a constant.

```
#include <stdio.h>

#define PI 3.14

main() {
        printf("%f",PI);
      }
```

Output:

```
3.140000
```

The #define preprocessor directive is used to define constant or micro substitution. It can use any basic data type.

Syntax:

```
#define token value
```

Example:

#define to create a macro.

```
#include <stdio.h>

#define MIN(a,b) ((a)<(b)?(a):(b))

void main() {

   printf("Minimum between 10 and 20 is: %d\n", MIN(10,20));

}
```

Output:

```
Minimum between 10 and 20 is: 10
```

Compiler Attributes:

Attributes allow us **to tell the compiler some specific details about how to compile certain parts of code**. These attributes can be applied to variables, structures, structure members and functions. Attributes are a feature of the compiler; they are not part of the C standard language.
OR
Attributes are **a mechanism by which the developer can attach extra information to language entities with a generalized syntax**, instead of introducing new syntactic constructs or keywords for each feature.

Example:

aux

The aux attribute is used to directly access the ARC's auxiliary register space from C. The auxiliary register number is given via attribute argument.