Unit : 8

Pre Process Directives and DMA

Introduction

- Preprocessor directives are lines included in a program that begin with the character #, which make them different from a typical source code text. They are invoked by the compiler to process some programs before compilation. Preprocessor directives change the text of the source code and the result is a new source code without these directives.

#define

- In the C Programming Language, the #define directive allows the definition of macros within your source code. These macro definitions allow constant values to be declared for use throughout your code.
- Macro definitions are not variables and cannot be changed by your program code like variables. You generally use this syntax when creating constants that represent numbers, strings or expressions.

Syntax

- #define CNAME value

- #define CNAME (expression)

Example

- #define AGE 10

- #define AGE (20 / 2)

#undef

- In the C Programming Language, the #undef directive tells the preprocessor to remove all definitions for the specified macro. A macro can be redefined after it has been removed by the #undef directive.

Syntax

- undef macro

Example

- #undef AGE

#ifndef

- In the C Programming Language, the #ifndef directive allows for conditional compilation. The preprocessor determines if the provided macro does **not** exist before including the subsequent code in the compilation process.

Syntax

- #ifndef *macro_definition*

Example

```
#include <stdio.h>
#define YEARS_OLD 12
#ifndef YEARS_OLD
#define YEARS_OLD 10
#endif
void main()
{
   printf("TechOnTheNet is over %d years old.\n",
   YEARS_OLD);
}
```

#if

- In the C Programming Language, the #if directive allows for conditional compilation. The preprocessor evaluates an expression provided with the #if directive to determine if the subsequent code should be included in the compilation process.

Syntax

- #if *conditional_expression*

Example

```
#include <stdio.h>
#define WINDOWS 1
void main()
{
    printf("TechOnTheNet is a great ");
   #if WINDOWS
   printf("Windows ");
   #endif
   printf("resource.\n");
}
```

#elif

- in the C Programming Language, the #elif provides an alternate action when used with the #if, #ifdef, or #ifndef directives. The preprocessor will include the C source code that follows the #elif statement when the condition of the preceding #if, #ifdef or #ifndef directive evaluates to false and the #elif condition evaluates to true.

Syntax

- #elif *conditional_expression*

Example

```
#include <stdio.h>
#define YEARS_OLD 12
void main()
{
    #if YEARS_OLD <= 10
   printf("TechOnTheNet is a great resource.\n");
   #elif YEARS_OLD > 10
   printf("TechOnTheNet is over %d years old.\n",
   YEARS_OLD);
   #endif
}
```

#else

- In the C Programming Language, the #else directive provides an alternate action when used with the #if, #ifdef, or #ifndef directives. The preprocessor will include the C source code that follows the #else statement when the condition for the #if, #ifdef, or #ifndef directive evaluates to false.

Syntax

- #else

Example

```
#include <stdio.h>
#define YEARS_OLD 12
void main()
{
   #if YEARS_OLD < 10
   printf("TechOnTheNet is a great resource.\n");
   #else
   printf("TechOnTheNet is over %d years old.\n",
   YEARS_OLD);
   #endif
}
```

#include

- In the C Programming Language, the #include directive tells the preprocessor to insert the contents of another file into the source code at the point where the #include directive is found. Include directives are typically used to include the C header files for C functions that are held outsite of the current source file.

Syntax

- #include <*header_file*>

- #include "*header_file*"

Example

- #include<conio.h>

#warning

- In the C Programming Language, the #warning directive is similar to an #error directive, but does not result in the cancellation of preprocessing. Information following the #warning directive is output as a message prior to preprocessing continuing.

Syntax

- #warning *message*

Example

```
#include <stdio.h>
void main() {
    /* The age of TechOnTheNet in seconds */
   int age;
   #warning The variable age may exceed the size of a 32 bit
   integer
   /* 12 years, 365 days/year, 24 hours/day, 60 minutes/hour,
   60 seconds/min */
   age = 12 * 365 * 24 * 60 * 60;
   printf("TechOnTheNet is %d seconds old\n", age);
}
```

#error

- In the C Programming Language, the #error directive causes preprocessing to stop at the location where the directive is encountered. Information following the #error directive is output as a message prior to stopping preprocessing.

Syntax

- #error *message*

Example

```
#include <stdio.h>
#include <limits.h>
/*
Calculate the number of milliseconds for the provided age in
   years * Milliseconds = age in years * 365 days/year * 24
   hours/day, 60 minutes/hour, 60 seconds/min, 1000
   millisconds/sec
*/

#define MILLISECONDS(age) (age * 365 * 24 * 60 * 60 *
   1000)
```

```
void main()
{
   /* The age of TechOnTheNet in milliseconds */
   int age;
   #if INT_MAX < MILLISECONDS(12)
   #error Integer size cannot hold our age in milliseconds
   #endif
   /* Calculate the number of milliseconds in 12 years */
   age = MILLISECONDS(12);
   printf("TechOnTheNet is %d milliseconds old\n", age);
}
```

Dynamic Memory Allocation

- As you know, you have to declare the size of an array before you use it. Hence, the array you declared may be insufficient or more than required to hold data. To solve this issue, you can allocate memory dynamically.
- Dynamic memory management refers to manual memory management. This allows you to obtain more memory when required and release it when not necessary.

- Although C inherently does not have any technique to allocate memory dynamically, there are 4 library functions defined under <stdlib.h> for dynamic memory allocation.

Functions

- 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() deallocate the previously allocated space.
- realloc() Change the size of previously allocated space

malloc()

- The name malloc stands for "memory allocation".
- The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

Syntax

- ptr = (cast-type*) malloc(byte-size)

- ptr = (int*) malloc(100 * sizeof(int));

calloc()

- The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax

- ptr = (cast-type*)calloc(n, element-size);

- ptr = (float*) calloc(25, sizeof(float));

free()

- Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. You must explicitly use free() to release the space.

Syntax

- free(ptr);

Example 1

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int num, i, *ptr, sum = 0;
   printf("Enter number of elements: ");
   scanf("%d", &num);
   ptr = (int*) malloc(num * sizeof(int));
   //memory allocated using malloc
   if(ptr == NULL)
```

```
{
    printf("Error! memory not allocated.");
}
printf("Enter elements of array: ");
for(i = 0; i < num; ++i)
{
    scanf("%d", ptr + i);
    sum += *(ptr + i);
}
printf("Sum = %d", sum);
free(ptr);
}
```

Example 2

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int num, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &num);
    ptr = (int*) calloc(num, sizeof(int));
    if(ptr == NULL)
    {
     printf("Error! memory not allocated.");
    }
```

```
printf("Enter elements of array: ");
for(i = 0; i < num; ++i)
{
        scanf("%d", ptr + i);
        sum += *(ptr + i);
}
   printf("Sum = %d", sum);
   free(ptr);
}
```

realloc()

- If the previously allocated memory is insufficient or more than required, you can change the previously allocated memory size using realloc().

Syntax

- ptr = realloc(ptr, newsize);

Example 3

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
   int *ptr, i , n1, n2;
   printf("Enter size of array: ");
   scanf("%d", &n1);
   ptr = (int*) malloc(n1 * sizeof(int));
```

```
printf("Address of previously allocated memory: ");

 for(i = 0; i < n1; ++i)
{
    printf("%u\t",ptr + i);
}
printf("\nEnter new size of array: ");
scanf("%d", &n2);
ptr = realloc(ptr, n2);
for(i = 0; i < n2; ++i)
{
        printf("%u\t", ptr + i);
}
}
```