Fundamentos de Programación – Programación Estructurada en C:
2.- EL PRECOMPILADOR

Maider Huarte Arrayago

Fundamentos de Programación – Programación Estructurada en C:

2.- EL PRECOMPILADOR

2.- EL PRECOMPILADOR

El precompilador del Lenguaje C permite incluir ficheros, realizar compilaciones condicionales y definir y sustituir **macros**, antes de que se haga la compilación propiamente dicha.

Las macros son Constantes Simbólicas o texto, con o sin parámetros.

2.1.- Comando *#include*

Sintaxis del comando:

> **#include "**<nombre_de_un_fichero>**"**
> **#include <**<nombre_de_un_fichero>**>**

Cuando en un fichero fuente se encuentra una línea con el comando *#include*, el precompilador la sustituye por el contenido del fichero especificado. El contenido de ese fichero debe ser un programa en C, compatible con el programa en el que se incluye.

La 1ª forma sintáctica, indica al precompilador que busque el fichero en el directorio actual primero, y si no está ahí, en el directorio estándar de librerías.
La 2ª forma sintáctica, se busca directamente en el directorio estándar.

El directorio estándar de librerías, en el caso de compiladores Microsoft (que se ejecutan sobre Sistemas Operativos Microsoft Windows-MS DOS), se establece con una variable de entorno del MS-DOS, llamada INCLUDE. En máquinas con Sistema Operativo tipo Linux, el directorio estándar donde se guardan las librerías es el */usr/lib*, y el directorio donde se guardan los ficheros de cabecera relacionados es el */usr/include*.

Hay que tener cuidado de que los ficheros incluidos en la compilación por este comando no contengan, a su vez, los mismos ficheros incluidos. Eso daría un error de compilación, porque los mismos elementos aparecerían declarados varias veces.

Ejemplo:

> Supongamos que un fichero A tiene incluido un fichero B. Si se crea un fichero C, y se incluye el fichero A, hay que tener cuidado y no incluir el fichero B también en C, ya que al estar incluido en A, ya se incluye implícitamente en C.

Este comando de precompilador se suele usar para incluir ficheros de cabecera de librerías en la precompilación, de forma que esto implica a su vez la inclusión de las librerías correspondientes en la fase de enlazamiento.

2.2.- Comando *#define*

Sintaxis del comando:

#define <NOMBRE> <texto a introducir>
#define <NOMBRE**(**<parámetros>**)**> <texto a introducir con parámetros>

La primera forma de sintaxis establece una macro. Así, el precompilador analiza el programa en la precompilación, y cada vez que encuentra el identificador NOMBRE, lo sustituye por *texto a introducir*.

Este mecanismo de sustitución permite definir constantes simbólicas y poder cambiarlas fácilmente, a la vez que el programa se mantiene más legible.

La 2ª forma de sintaxis define **macros con parámetros**.
Ejemplo:

```
#define CUAD(x) ((x)*(x))
```

Se cumple así la labor de una función de una sola instrucción, que se analizará más adelanten, en el tema **6.- FUNCIONES**.

Hay algunas macros ya predefinidas

__DATE__	Fecha de compilación
__FILE__	Nombre del fichero
__LINE__	Número de línea
__TIME__	Hora de compilación

Se pueden definir macros sin *texto a sustituir*, para usarlas como **señal** (para compilaciones condicionales, por ejemplo) a lo largo del programa.

2.3.- Comando #undef

Sintaxis del comando:

#undef <macro>

Anula la definición de una macro anterior. Así, el precompilador, cuando encuentra este comando, deja de sustituir macro por el texto indicado en el comando de definición (en el #define de esa macro).

2.4.- Directrices condicionales

Las directrices condicionales sirven para establecer bloques de compilación opcionales. La compilación condicional se usa con frecuencia para el desarrollo de **código portable** para distintos tipos de computadores. Según de qué computador se trate, se compilan unas líneas u otras, porque en unos computadores las cosas se hacen mejor de una forma y en otros de otra. La diferencia entre computadores, puede ser, por ejemplo, a nivel de Sistema Operativo.

2.4.1.- Comando #if

Sintaxis del comando:

```
#if <condición>
    <sentencias if>
[#else
    <sentencias else>]
#endif
```

Si la *condición* es cierta, se compilan las *sentencias if*. Si no, se compilan las *sentencias else*. *#else* es opcional.

2.4.2.- Comando #ifdef

Sintaxis del comando:

```
#ifdef <macro>
    <sentencias ifdef>
[#else
    <sentencias else>]
#endif
```

Si la *macro* está definida, se compilan las *sentencias_ifdef*. Si no, se compilan las *sentencias_else*. *#else* es opcional.

Ejemplo 1:

```
#include <stdio.h>

#define MARCA

#ifdef MARCA
int main()
{
    printf("\nSe compila esto, cuando MARCA está definido\n\n");

    return 0;
}
#else
int main()
{
    printf("\nSe compila esto, cuando MARCA no está definido\n\n");

    return 0;
}
#endif

/* Resultado de la ejecución de este programa es:
Se compila esto, cuando MARCA estβ definido
*/
```

Ejemplo 2:

```
#include <stdio.h>

//#define MARCA

#ifdef MARCA
int main()
{
    printf("\nSe compila esto, cuando MARCA está definido\n\n");

    return 0;
}
#else
int main()
{
    printf("\nSe compila esto, cuando MARCA no está definido\n\n");

    return 0;
}
#endif

/* El resultado de la ejecución de este programa es:
Se compila esto, cuando MARCA no estβ definido
*/
```

2.4.3.- Comando #ifndef

Sintaxis del comando:

```
#ifndef <macro>
    <sentencias ifndef>
[#else
    <sentencias else>]
#endif
```

Si la *macro* NO está definida, se compilan las *sentencias ifndef*. Si no, se compilan las *sentencias else*. *#else* es opcional.