


A vertical bar on the left side of the slide, composed of a pink segment at the top, a light blue segment, a yellow segment, and a long pink segment at the bottom.

# EL PREPROCESADOR DE C




# El preprocesador

- El preprocesamiento es el primer paso en la etapa de compilación de un programa.
  - Es una característica del compilador de C.
  - Ventajas de usar el preprocesador
    - Programas más fáciles de desarrollar, de leer y de modificar.
    - El código C es más portable entre diferentes arquitecturas de máquinas.
- 



# El preprocesador

- El preprocesador tiene su propio lenguaje el cual puede ser una herramienta muy poderosa para el programador.
  - Formato de las directivas del preprocesador
    - Todas las directivas del preprocesador o comandos inician con un **#**.
    - Delante de la directiva sólo se pueden poner espacios en blanco.
- 




# Preprocesamiento

- Permite
  - Incluir archivos
  - Definir constantes simbólicas y macros.
  - Compilación condicional del código.
  - Ejecución condicional de las directivas del preprocesador.





# Directivas

- **#include**
  - **#define**
    - Sirve para definir **constantes y macros**.
  - **#undef**
  - **#if**
  - **#ifdef y #ifndef**
- 



# #include

- Sirve para insertar archivos externos dentro de nuestro archivo de código fuente.
- El archivo especificado será insertado en el código en el lugar de la directiva.

- **Sintaxis**

`#include <archivo>`     *// directorio predefinido*

`#include "archivo"`     *// directorio corriente*



# #include

## **#include <archivo>**

- Busca el archivo en la librería estándar
- Se utiliza para los arch.de la librería estándar.

## **#include "archivo"**

- Busca primero en el directorio actual y luego en la librería estándar.
- Se utiliza para archivos definidos por el usuario.





# #include

- Usado por
  - Programas formados por varios archivos de código fuente que deben ser compilados juntos.
  - Archivos **headers** que poseen declaraciones y definiciones comunes (clases, estructuras, prototipos de funciones). Habrá una sentencia `#include` en cada archivo.








# #define

Definiendo una constante simbólica

## Sintaxis

**#define *identificador* [valor]**

- La constante puede no tener **valor** asociado. En ese caso el **identificador** será reemplazado con un texto en blanco. Esto se utiliza con **#ifdef** y **#ifndef**.
  - Si un **valor** es provisto, el **identificador** será reemplazado literalmente por **valor** (el resto del texto en la línea).
  - Una constante simbólica no se puede redefinir una vez que ha sido creada.
- 



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

```
#define PI 3.14159
```

```
int main()
```

```
{    float radio;
```

```
    printf("Radio del círculo:");
```

```
    scanf("%f", &radio);
```


```
    printf("Area del círculo = %g\n",  
           3.14159 * radio * radio);
```

```
    printf("Long. de la circunferencia = %g\n",  
           2 * 3.14159 * radio);
```

```
    return 0;
```

```
}
```





```
#include <stdio.h>

#define PI = 3.14159
```

¿Compila ?


```
int main()
{   float radio;

    printf("Radio del círculo:");
    scanf("%f", &radio);

    printf("Area del círculo = %g\n",
           = 3.14159 * radio * radio);

    printf("Long. de la circunferencia = %g\n",
           2 * = 3.14159 * radio);

    return 0;
}
```



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

```
#define begin {
#define end }
```

```
#define FALSE 0
#define TRUE !FALSE
```

```
int main()
{
    char pal[50];
    printf("Sintaxis tipo Pascal!\n");
    scanf("%s", &pal);
    if (strcmp(pal, "FIN") == FALSE)
        begin
            printf("Se ingreso la palabra\n");
            printf("FIN!");
        end
    else begin
        printf("La palabra ingresada\n");
        printf("no es FIN");
    end
    return 0;
}
```

**#define** puede usarse  
para redefinir el lenguaje



# Macro

- Es una operación definida mediante **#define**
- Una macro sin argumentos es tratada como una constante simbólica.
- Una macro con argumentos, al ser expandida, reemplaza sus argumentos con los argumentos reales encontrados en el programa.
- Realiza una sustitución de texto, sin chequeo de tipos.



# Macro. Definición - Ejemplo

La macro

```
#define AREA(x) PI * x * x
```

puede causar que

```
area = AREA(4);
```

se convierta en

```
area = 3.14159 * 4 * 4 ;
```

# Macro. Definición - Ejemplo

```
#include <stdio.h>

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

int main()
{
    int x=10, y=20;

    printf("El minimo es %d\n",
           MIN(x,y) );

    return 0;
}
```

# Macro. Definición - Ejemplo

```
#include <stdio.h>

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

int main()
{
    int x=10, y=20;

    printf("El minimo es %d\n",
           (x < y) ? x : y );

    return 0;
}
```



# Macro. Definición - Ejemplo

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

```
#define DESPL_IZQ_8 <<8
```

```
int main()
```

```
{
```

```
    int x=0xBB, y;
```

```
    y = x DESPL_IZQ_8;
```

```
    printf("x = %x   y = %x\n", x, y);
```

```
    return 0;
```

```
}
```

# Macro. Uso de paréntesis

La macro

```
#define AREA(x) PI * x * x
```

puede causar que

```
area = AREA( c + 1 );
```

se convierta en

```
area = 3.14159 * c + 1 * c + 1 ;
```

¿Cómo se resuelve?

# Macro. Uso de paréntesis

La macro

```
#define AREA(x) (PI * (x) * (x))
```

puede causar que

```
area = AREA( c + 1 );
```

se convierta en

```
area = (3.14159 * (c+1) * (c+1));
```

# El operador #

- Reemplaza el argumento por un texto encerrado entre comillas

```
#include <stdio.h>

#define saludo(x) "Hola " #x "!\n"

int main()
{
    printf("%s", saludo(Ana));

    return 0;
}
```

# El operador #

- Reemplaza el argumento por un texto encerrado entre comillas

```
#include <stdio.h>

#define saludo(x) "Hola " #x "!\n"

int main()
{
    printf("%s", "Hola " "Ana" "!\n");

    return 0;
}
```

printf concatena los strings separados por blancos

## El operador ##

- Concatena dos tokens (secuencias de caracteres sin blancos)
- La sentencia

```
#define TOKENCONCAT( x, y )  x ## y
```

haría que

```
TOKENCONCAT( O, K )
```

se convierta en

```
OK
```

# El operador ##

¿Qué imprime?

```
#include <stdio.h>

#define concatena(x,y) x ## y

int main()
{
    printf("%d", concatena(10,20));

    return 0;
}
```



# Ejercicio

- Defina una macro que reciba tres valores o expresiones numéricas y retorne el menor valor.
  - Utilizando una única instrucción `#define`
  - Definiendo primero una macro que halle el mínimo entre dos valores o expresiones numéricas y luego una segunda macro que la utilice para hallar el mínimo de tres.







# #undef

- La directiva **#undef** «elimina» la definición de una constante simbólica o macro.
- El alcance de una constante o de una macro cubre desde su definición hasta que se elimina con **#undef** o termina el programa.
- La definiciones eliminadas pueden volver a definirse utilizando **#define**.



# #if, #elif, #else y #endif

- Permiten hacer una **compilación condicional** de un conjunto de líneas de código.
- **Sintaxis**

```
#if expresión-constante-1  
<sección-1>  
#elif <expresión-constante-2>  
<sección-2>  
.  
.  
.  
#elif <expresión-constante-n>  
<sección-n>  
<#else>  
<sección-final>  
#endif
```

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

```
#define MEX 0
```

```
#define EUA 1
```

```
#define FRAN 2
```

```
#define PAIS_ACTIVO MEX
```

```
#if PAIS_ACTIVO == MEX  
    char moneda[]="pesos";
```

```
#elif PAIS_ACTIVO == EUA  
    char moneda[]="dolar";
```

```
#else  
    char moneda[]="franco";
```

```
#endif
```

```
int main()  
{
```

```
    printf("Moneda = %s\n",moneda);
```

```
    return 0;
```

```
}
```

← Condición a verificar

← Condición alternativa

← Si ninguna se verifica ...

← Termina con **#endif**

# Ejemplo

```
#if !defined(NULL)
    #define NULL 0
#endif
```

- La expresión **defined(NULL)** se evalúa a 1, si NULL está definido y a 0 si no.
- Por lo tanto **!defined(NULL)** se evalúa a 1 si NULL no está definido y a través de la directiva **#define** se define.
- Cada constructor **#if** termina con **#endif**.

# Compilación condicional

- Note que el siguiente programa no presenta errores de sintaxis

```
int main()  
{  
    #if 0  
        Segmento que no  
        se compila  
    #endif  
  
    return 0;  
}
```

# #ifdef e #ifndef

- Permiten comprobar si un identificador está o no actualmente definido, es decir, si un **#define** ha sido previamente procesado para el identificador y si sigue definido.
- Sintaxis

```
#ifdef <identificador>  
#ifndef <identificador>
```

```

#include <stdio.h>

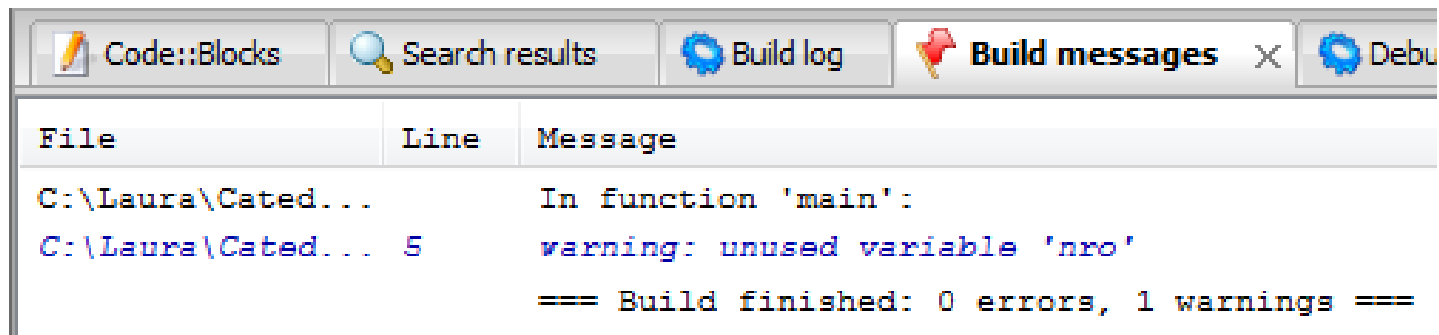
#define EN_PRUEBA
int main()
{
    int nro = 20;

    #ifdef EN_PRUEBA
        printf("Este codigo NO es estable\n"
               "Se verifican llamados a funcion \n");
    #else
        printf("Resultados confirmados! ");
        printf("%d", nro);
    #endif

    return 0;
}

```

- Al compilar aparece esto. Por qué?






# PROGRAMAS FORMADOS POR VARIOS ARCHIVOS







# Programas formados por varios archivos

- Hasta ahora todo el código de nuestra aplicación se encuentra codificado en un único archivo.
  - Esto es poco práctico cuando se trata de
    - Trabajar en grupo
    - Reusar código
    - Utilizar compilación separada
- 




# Escriba este código en un único archivo

```
#include <stdio.h>
void VerTexto(void)
{
    printf("Texto fijo!!!!\n");
}
int main()
{
    printf("Ejecutando...\n");

    VerTexto();

    printf("Terminado.\n");

    return(0);
}
```



Ejecútelo para asegurarse que no tiene errores de sintaxis

```
#include <stdio.h>
// prototipo de la función
void VerTexto(void);
```



```
int main()
{
    printf("Ejecutando...\n");

    VerTexto();

    printf("Terminado.\n");

    return(0);
}
```

```
//-----
void VerTexto(void)
{
    printf("Texto fijo!!!!\n");
}
```

Si usamos una función,  
antes de su definición  
debemos incluir su  
prototipo

Todo el código sigue  
estando codificado  
dentro de un único  
archivo.

# Corte la función VerTexto del programa principal

main.c

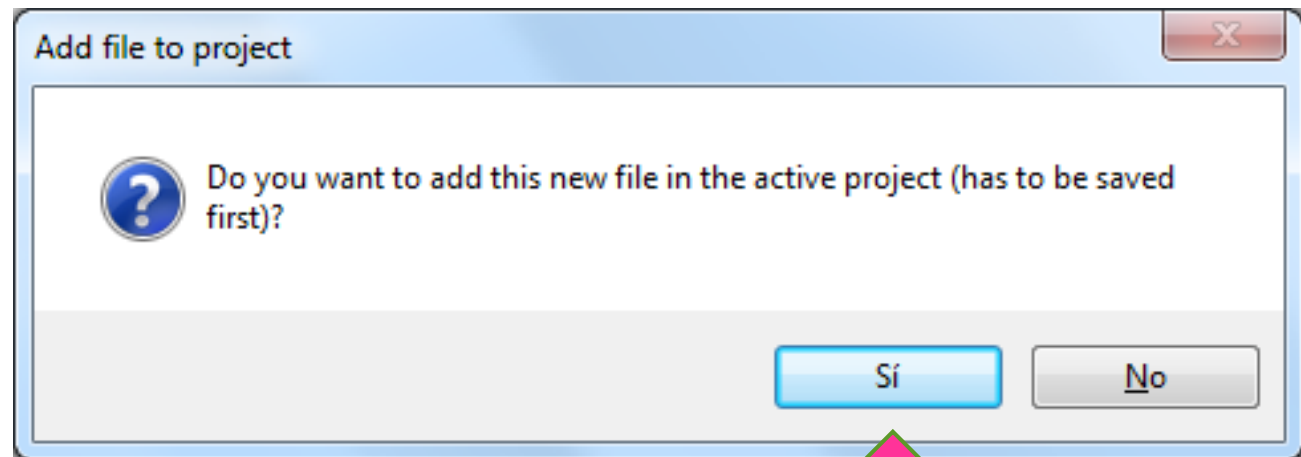
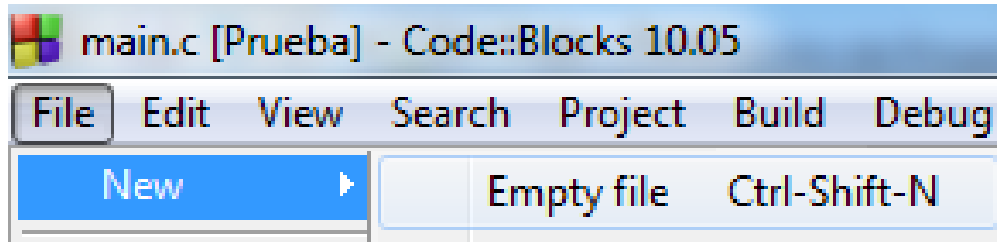
```
#include <stdio.h>
void VerTexto(void);

int main()
{
    printf("Ejecutando...\n");
    VerTexto();
    printf("Terminado.\n");

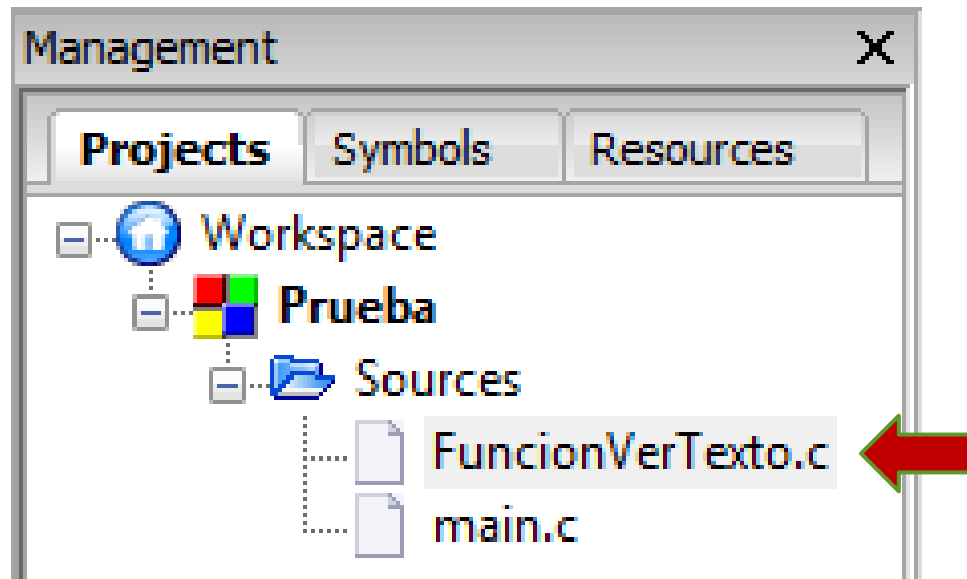
    return(0);
}
```

Agregaremos un **archivo vacío** al proyecto y pegaremos allí el código correspondiente a la función **VerTexto**.

# Agregue un archivo vacío al proyecto

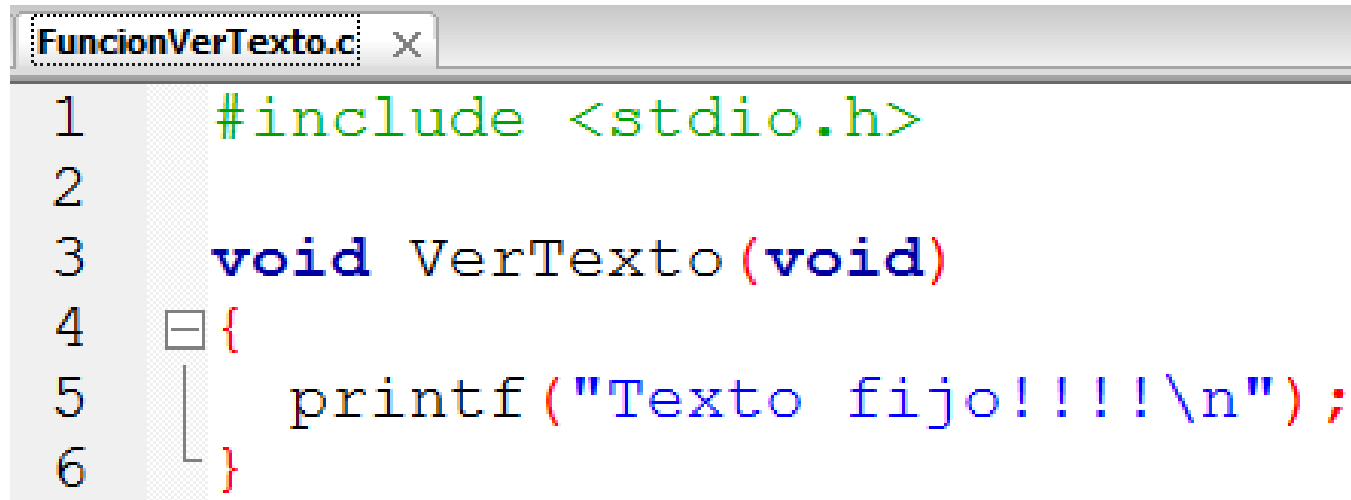


# Agregue un archivo vacío al proyecto



Asígnele un nombre al archivo vacío y pegue en él la función

Ahora la función está en un archivo separado



```
FuncionVerTexto.c x
1  #include <stdio.h>
2
3  void VerTexto(void)
4  {
5      printf("Texto fijo!!!!\n");
6  }
```

Compile y verifique que funciona

# Detalles a tener en cuenta

main.c

```
#include <stdio.h>
void VerTexto(void) ; ←
int main()
{
    printf("Ejecutando...\n");
    VerTexto();
    printf("Terminado.\n");

    return(0);
}
```

Si la función cambia, hay que cambiar el prototipo de la función. Esto puede ocurrir en más de un lugar.

**Es poco práctico**



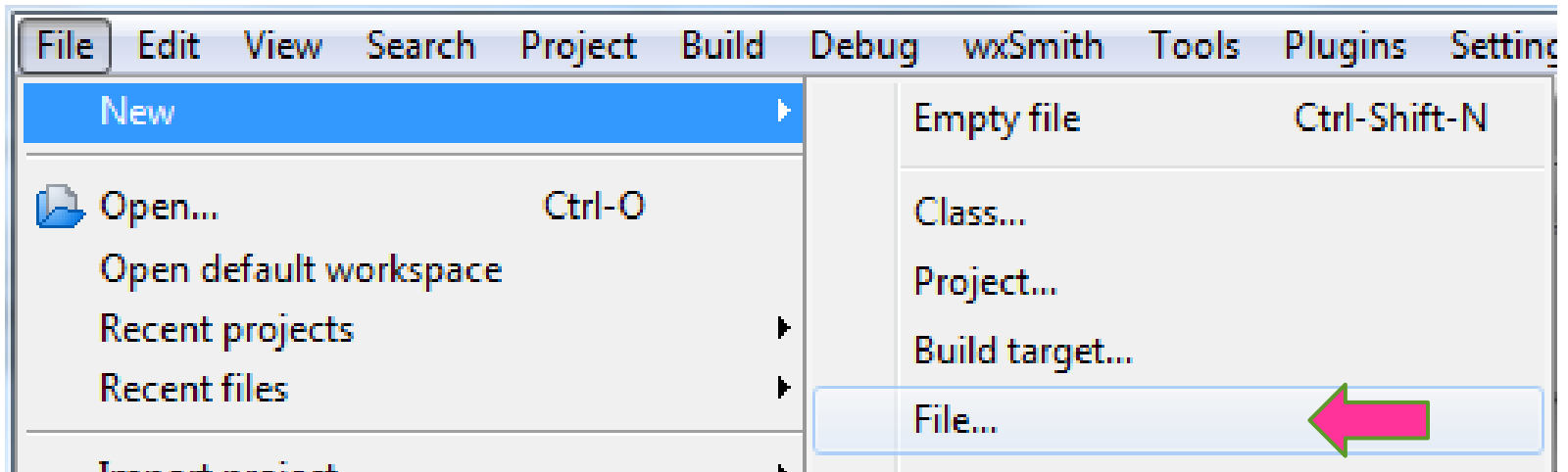
Puede ser peor aun. Si la función cambia y no se actualiza el prototipo se pueden tener resultados inesperados

```
FuncionVerTexto.c x
1  #include <stdio.h>
2
3  void VerTexto(char * cadena)
4  {
5      printf("%s\n", cadena);
6  }
```

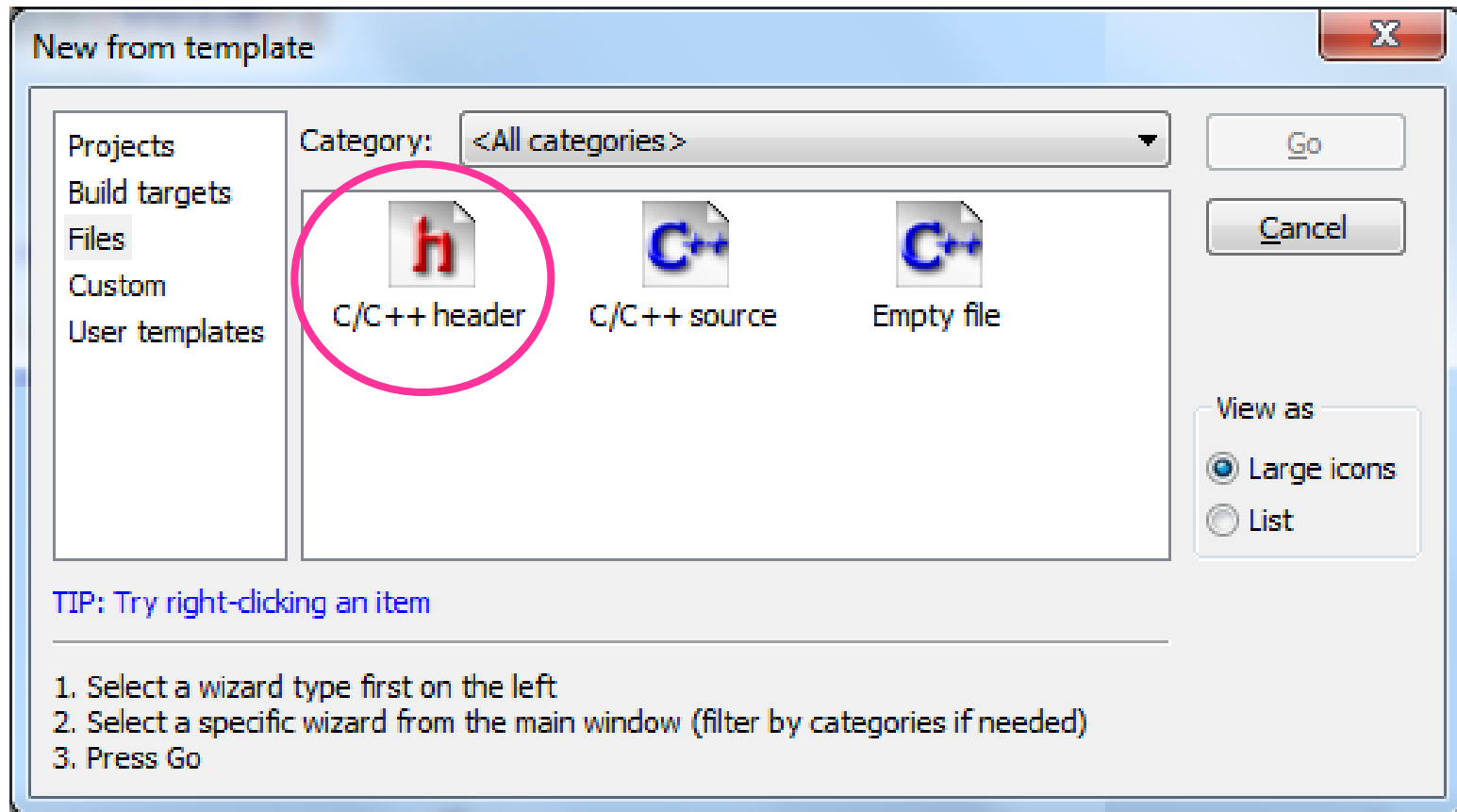
**Compila? Por qué?**

Qué resultados produce?

Para evitar estos problemas se utilizan los archivos cabecera



Para evitar estos problemas se utilizan los archivos cabecera



C/C++ header



Please enter the file's location and name and whether to add it to the active project.

Filename with full path:

2do semestre\Ejemplos\Prueba\cabecera.h

Header guard word:

CABECERA\_H\_INCLUDED

☒ Add file to active project  
In build target(s):

- ☒ Debug
- ☒ Release

All

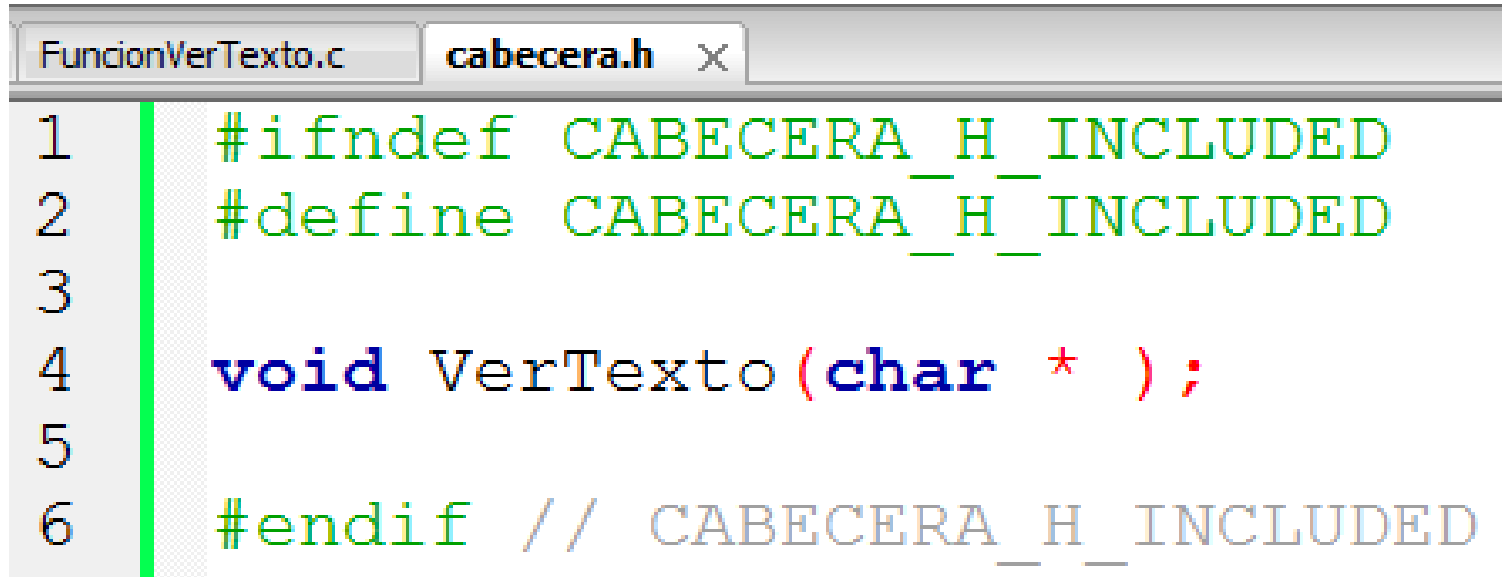
None

< Back

Finish

Cancel

# Archivo cabecera.h

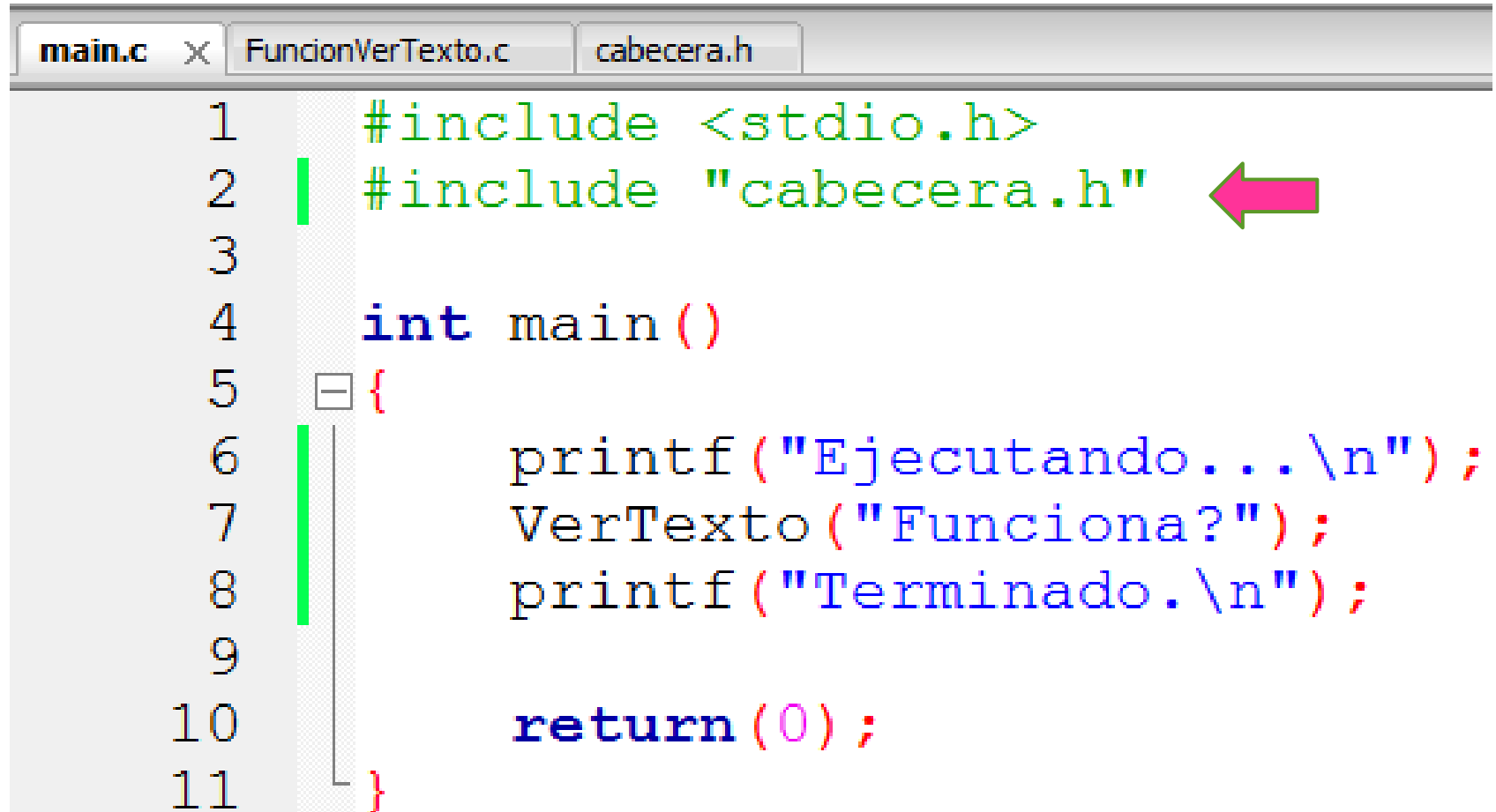


The image shows a code editor window with two tabs: 'FuncionVerTexto.c' and 'cabecera.h'. The 'cabecera.h' tab is active, displaying the following C header file code:

```
1  #ifndef CABECERA_H_INCLUDED
2  #define CABECERA_H_INCLUDED
3
4  void VerTexto(char * );
5
6  #endif // CABECERA_H_INCLUDED
```

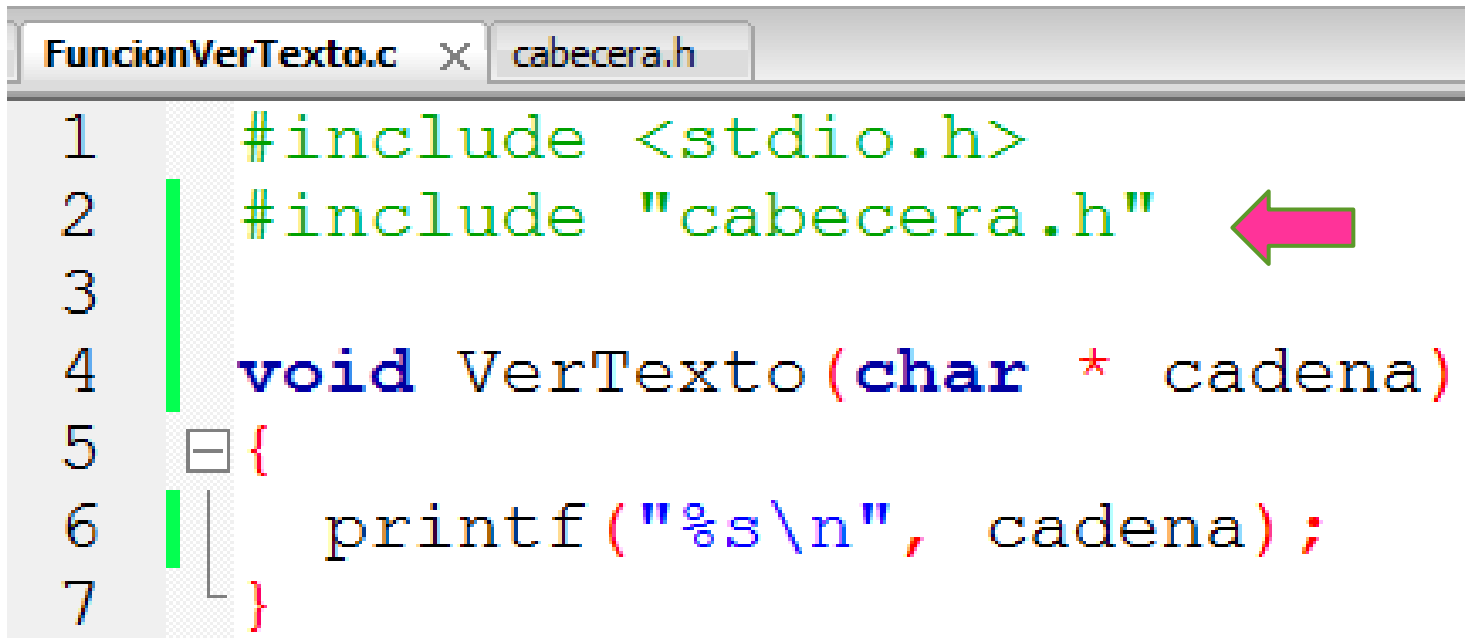
- Este archivo utiliza «include guards» (guardas include) para evitar múltiples definiciones de la función.

# Archivo main.c



```
1  #include <stdio.h>
2  #include "cabecera.h"
3
4  int main()
5  {
6      printf("Ejecutando...\n");
7      VerTexto("Funciona?");
8      printf("Terminado.\n");
9
10     return(0);
11 }
```

# Archivo FuncionVerTexto.c



```
1  #include <stdio.h>
2  #include "cabecera.h"
3
4  void VerTexto(char * cadena)
5  {
6      printf("%s\\n", cadena);
7  }
```

- Puede incluirse la cabecera para chequear consistencia y evitar errores.

**Compile y verifique que funciona**



# Ejercicio

- Defina una biblioteca de funciones para trabajar con una estructura que almacena una hora indicada por el usuario.
- Debe proveer
  - Una función para leerla desde teclado (valide que la hora sea un entero en  $[0,24)$  y los minutos y segundos sean enteros en  $[0,60)$ )
  - Dos funciones para visualizarla: una con los valores cargados y otra como AM-PM (sólo cambia la hora).





# Ejercicio

Este sería el contenido de  
**"horario.h"**

Poner la implementación de  
las funciones en **"horario.c"**

```
struct horario {  
    int hora;  
    int minutos;  
    int segundos;  
};
```

Compile ANTES de escribir  
la función **main.c** para  
verificar la sintaxis

```
void LeerHorario(struct horario *);
```

```
void VerHorario(struct horario);
```

```
void VerHorarioAM_PM(struct horario);
```

# Ejercicio

- Este es el código de **main.c**

```
#include <stdio.h>
#include "horario.h"

int main()
{
    struct horario hs;

    LeerHorario(&hs);

    VerHorario(hs);

    VerHorarioAM_PM(hs);

    return 0;
}
```