# 1 Instrumentación basada en PC

## *1.1 ANSI C*

### 1.1.1 Introducción

El American National Standards Institute (ANSI) estableció un comité para elaborar una definición del lenguaje C, no ambigua e independiente de la máquina, siendo su resultado lo que se conoce como ANSI C. Otro estándar ha sido el lenguaje C definido en la referencia [1] que se conoce como C Kernighan & Ritchie.

C es un lenguaje de programación de propósito general que nació con una estrecha relación con el sistema operativo UNIX (escrito en C), y con fines de proveer una plataforma común para todas las máquinas. Pese a ello, el lenguaje C no está ligado a ninguna máquina ni sistema operativo, y se utiliza con igual profusión para escribir aplicaciones de bajo nivel (aplicaciones de sistemas y sistemas operativos), como para aplicaciones de usuario final. Debido a la acción de diversos fabricantes, se han producido sagas que incorporan funcionalidades que no son compatibles entre sí, perdiéndose así parte de la portabilidad que caracterizaba la idea original. Hoy en día son conocidas los entornos de desarrollo de Microsoft VisualC, Borland C, UNIX y la aplicación que aquí contemplamos en mayor detalle, LabWindows CVI, orientada a aplicaciones de instrumentación. El lenguaje ANSI C podría definirse como en núcleo común de compatibilidad entre todos ellos. Un desarrollo posterior son los lenguajes orientados a objetos que dan lugar a las versiones de C++.

C es un lenguaje que cubre un conjunto de niveles muy amplio, pero en su base puede decirse que es de un bajo nivel relativo. Esto es, está más cerca de la máquina que del usuario final. En parte debido a esta circunstancia, el lenguaje básico carece de sentencias de entrada / salida (READ / WRITE), ni de acceso a archivos. Se consideran mecanismos de alto nivel, y se implementan a través de llamadas explícitas a funciones.

Todo ello proporciona un núcleo de desarrollo relativamente pequeño que permite su aprendizaje rápido y sencillo, al menos en los aspectos básicos.

#### 1.1.1.1 Ejemplo de un programa en C

```
/* Programa de prueba */
/* prueba.c */
#include <stdio.h>              /* Se incorpora la biblioteca estándar(std) */
                                /* de entrada / salida (io)                 */
main()                          /* Programa principal = función main        */
{                               /* Comienzo de main                         */
printf("Hola  \n");             /* Cuerpo de main = llamada a la función    */
                                /* de la biblioteca stdio llamada           */
                                /* printf (imprimir).                       */
                                /* Se imprime la palabra Hola, y un         */
                                /* retorno de carro (\n).                   */
                                /* Final de main                            */
}
```

Obsérvese que cada línea de sentencia termina con un punto y coma, y que los comentarios se introducen entre /* y */ (// es una variante introducida por Microsoft).

```
/* comentario */          es equivalente a
// comentario
```

#### 1.1.1.2 Estructura de un programa en C

El siguiente esquema muestra las partes fundamentales de un programa escrito en lenguaje C.

| Num. | Sección | Contenido |
|---|---|---|
| 1 | `Includes` | directivas de preprocesado para las cabeceras (headers) de ficheros `#include` |
| 2 | `Defines` | directivas de preprocesado para definir constantes simbólicas y macros `#define` |
| 3 | `Typedefs` | declaraciones para nuevos nombres de tipos de datos |
| 4 | `Prototypes` | prototipos de funciones |
| 5 | `Globals` | declaración de variables globales |
| 6 | `main()` | cuerpo principal del programa (`main`) |
| | `{` | |
| | `locals` | variables locales |
| | `statics` | variables estáticas |
| | `statements` | sentencias del programa |
| | `}` | |
| 7 | `Functions` | definiciones de funciones |
| | `func1` | función `func1` |
| | `{` | |
| | `}` | |
| | `func2` | función `func2` |
| | `{` | |
| | `}` | |

El conjunto de elementos que definen una función (incluida la función `main`), tiene contenidas las sentencias que la forman entre los símbolos { }.

Cada sentencia se termina con el símbolo punto y coma (;).

Los comentarios se escriben entre los símbolos /* y */ , o bien // y //, según el compilador.

A continuación se incluye un programa que sirve de ejemplo:

#### 1.1.1.3 Ejemplo de programa en CVI

```
/* Ejemplo de programa en CVI */
/* ejemplo.c */

/* INCLUDES */
#include <ansi_c.h>
#include <userint.h>
#include <dataacq.h>
#include <daq.h>

/* DEFINES */
#define TRUE  1
#define FALSE 0
#define POINTS 1000

/* TYPEDEF */
/* no hay sentencias de typedef */

/* PROTOTIPO DE FUNCIONES */
int Media
(double data_array[], int num_elemetos, double *average);

/* GLOBALES */
static short buffer[POINTS];
static double waveform[POINTS];
int pnl_handle;
double samp_rate;

main()
{
pnl_handle = LoadPanel (0, "daq.uir", PNL);
DisplayPanel (pnl_handle);
RunUserInterface ();
}

/* DEFINICION DE FUNCIONES DE USUARIO */
int Media
(double data_array[], int num_elementos, double *average)

{
int i;
double array_sum = 0;

if (num_elementos == 0)
return (-1);  /* error */

else
{
for (i=0, i < num_elementos; i++)
array_sum += buffer[i];
*average = array_sum/num_elementos;
}
}
```

```
/* DEFINICION DE FUNCIONES CALLBACK */
int CVICALLBACK acquire
(int panel, int control, int event, void *callbackData, int eventData1, int
eventData2)

{
double avg;

if (event == EVENT_COMMIT)
{
GetCtrlVal (pnl_handle, PNL_RATE, ¬samp_rate);
DAQ_OP (1, 1, 1, buffer, POINTS, samp_rate);
DAQ_VScale (1, 1, 1, 1.0, 0.0, POINTS, bufffer, waveform);

Media (waveform, POINTS, &avg);
SetCtrlVal (pnl_handle, PNL_AVERAGE, avg);

DeleteGraphPlot (pnl_handle, PNL_GRAPH, -1, 1);
Plot Waveform
(pnl_handle,   PNL_GRAPH,   waveform,   POINTS,   VAL_double,   1.0,   0.0,   0.0,
1/samp_rate, VAL_THIN_LINE, VAL_EMPTY_SQUARE, VAL_SOLID, 1, VAL_RED);
}
return (0);
}

int Quit
(int panel, int control, int event, void *callbackData,
int eventData1, int event Data2)
{
if (event == EVENT_COMMIT)
QuitUserInterface (0);
return (0);
}
```

#### 1.1.1.4 Directivas de preprocesado

Se definen al comienzo del programa mediante las expresiones:

```
#include
```

```
#define
```

La directiva `#include` provoca que el contenido del fichero que se especifica a continuación, se incluye en el código fuente antes de la compilación, en la posición marcada por la directiva.

Los ficheros especificados en esta directiva se denominan ficheros de encabezados (header files), y se designan mediante la extensión .h. El contenido de estos ficheros está compuesto por definición de constantes, declaraciones de variables, y declaraciones de prototipos de funciones.

Los nombres de los ficheros se encierran entre los símbolos < y > si el fichero se localiza en el directorio declarado para los ficheros de encabezado en el compilador, o entre " y " si se incluye en otro directorio:

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

incorpora el fichero de encabezado que incluye las rutinas estándar (std) de entrada y salida de datos (io).

```
#include "c:\c\headers\header1.h"
```

### 1.1.2 Tipos de datos

Los tipos de datos más comunes son:

| | |
|---|---|
| `char` | un solo byte = un carácter = 8 bits |
| `int` | entero |
| `float` | flotante de precisión normal |
| `double` | flotante de doble precisión |

calificadores

| | |
|---|---|
| `short` | 16 bits |
| `long` | 32 bits |

se aplican a enteros, por ejemplo:

```
short int a;
long int b;
```

| | |
|---|---|
| `signed` | con signo |
| `unsigned` | sin signo |

se aplican a char, int, por ejemplo:

`unsigned char` 0 a 255 ($2^n$ n = número de bits, 8 bits = 1 byte)

`signed char` -128 a 127

| Los posibles tipos de datos son: | | | | rango | bits | dig |
|---|---|---|---|---|---|---|
| Simples | Enteros (`int`) | corto (`short`) | sin signo (`unsigned`) | 0 a 65355 | 16 | |
| | | | con signo (`signed`) | -32768 a 32767 | 16 | |
| | | largo (`long`) | sin signo | 0 a 4.29e+09 | 32 | |
| | | | con signo | ±2.15e+09 | 32 | |
| | | Carácter (`char`) | sin signo | 0 a 255 | 8 | |
| | | | con signo | -128 a 127 | 8 | |
| | Reales | | simple (`float`) | ±3.4e±38 | 32 | 7 |
| | | | doble (`double`) | ±1.7e±30864 | 15 | |
| | | largo | doble (`long double`) | ±1.2e±4932 | 128 | 19 |
| Referenciados | Punteros | | | | | |
| Estructurados | Arrays | vectores | nombre[n] | | | |
| | | matrices | nombre[n] [m] | | | |

Registros

Uniones

#### 1.1.2.1 Constantes

Se pueden definir constantes, constantes simbólicas y variables con los tipos de datos básicos (enteros, reales, carácter). Los sufijos pueden ser mayúsculas o minúsculas.

| | | | |
|---|---|---|---|
| Entera | corta | sin signo | 1234U |
| | | con signo | 1234 (número decimal) |
| | larga | sin signo | 1234UL |
| | | con signo | 1234L |
| | octal | | O1234 (prefijo = O) |
| | hexadecimal | | Ox4D2 (prefijo = Ox) |
| Carácter (string) | | | "Juan"<br>"\n"<br>"Texto en una línea\y en otra" |
| Real | simple | notación punto flotante | 1234. (. = real)<br>-12.34<br>-12.34f |
| | | notación científica | -1.234e+01 |

#### 1.1.2.2 Constantes simbólicas

Permiten definir variables con valores asignados, de manera que se comporten como constantes. Se suelen escribir en mayúsculas.

La directiva `#define` facilita una manera de sustituir una secuencia de caracteres por un valor especificado:

```
#define TRUE 1
#define FALSE 0
#define PI 3.14
```

Las constantes y variables definidas en la cabecera del programa, fuera de los bloques definidos por { }, actuan como elementos comunes a todo el programa, es decir, son constantes y variables globales, por lo que son accesibles desde cualquier parte del programa.

La directiva `typedef` permite asignar sus propios nombres a tipos de datos. Funciona de manera similar a la directiva `#define`, excepto que es compilada en vez de manejada por el preporcesador, y sólo se puede emplear para definir tipos de datos:

```
#typedef unsigned char BYTE;
BYTE ch1;
BYTE ch2;
```

son equivalentes a:

```
unsigned char ch1;
unsigned char ch2;
```

en donde se definen dos variables cha1 y cha2 como variables de tipo carácter (`char`) sin signo (`unsigned`).

```
/* Imprime la tabla de conversión de grados Farenheit a Celsius */
/* celsius.c */

#include <stdio.h>

#define LOWER 0
#define UPPER 300
#define STEP 20

main()
{
int fahr, celsius;

for (fahr = LOWER; fahr <= UPPER; fahr =fahr + STEP)
      printf("%3d %6.lf \n", fahr, (5.0/9.0)*(fahr-32));
}
```

#### 1.1.2.3 Constantes de carácter

Son secuencias de uno o más caracteres encerrados entre apóstrofos, como 'a'
El valor de una constante de carácter con un sólo carácter, es el valor numérico del carácter en el conjunto de caracteres de la máquina.

Por medio de constantes de carácter pueden escribirse secuencias de escape, es decir, códigos que representan funciones específicas en el dispositivo de salida (por ejemplo: \n representa el salto a una nueva línea, LF LineFeed en la representación tradicional, o LF y CR en la representación de MS-DOS, LineFeed seguido de CarriageReturn)

| Secuencias de escape | | |
|---|---|---|
| nueva línea | NL | \n |
| tabulador horizontal | HT | \t |
| tabulador vertical | VT | \v |
| retroceso | BS | \b |
| retorno de carro | CR | \r |
| avance de página | FF | \f |
| señal audible | BEL | \a |

| Caracteres especiales | | |
|---|---|---|
| diagonal inversa | \ | \\ |
| interrogación | ? | \? |
| apóstrofo | ' | \' |
| comillas | " | \" |
| número octal | ooo | \ooo |
| número hexadecimal | hh | \xhh |
| carácter NUL | NUL | \0 |

#### 1.1.2.4 Variables y expresiones aritméticas

Los nombres de variables se componen de letras y dígitos, debiendo ser una letra el primer carácter. Se distingue entre mayúsculas y minúsculas, siendo habitual usar las minúsculas para variables y las mayúsculas para constantes simbólicas. Son significativos los primeros 31 caracteres del nombre de una variable.

Se deben cuidar los siguientes detalles:

- No pueden tener nombres reservados por el compilador de C:
  `main, for, while, switch, ...`
- El primer carácter debe ser una letra o el carácter de subrayado (_)
- Se recomienda usar con precaución las variables que comienzan por _, pues es el carácter utilizado para definir variables en bibliotecas.
- Los restantes caracteres pueden ser letras, dígitos o subrayados
- Se distingue entre mayúsculas y minúsculas
- No se debe sobrepasar el número máximo de caracteres definido por el compilador

Las variables que se emplean en los programas escritos en C deben declararse obligatoriamente al comienzo del mismo, indicando el nombre y tipo de la variable. En el momento de su declaración, pueden también inicializarse. Se recomienda no trabajar con una variable que no haya sido inicializada, bien en su declaración, o en el cuerpo del programa.

```
int a = 0;
int x_array ^3* = {25, 50, 75}
float xy_array [2] [3] = {  {1.25, 2.5, 2.75} {3.14, 6.28, 9.42} }
char c_array [] = "Hola"
```

Los vectores y matrices se pueden dimensionar en la misma sentencia de declaración de tipo

```
int x_array [20];
int xy_array [10] [10];
char c_array [80];
```

La definición de variables puede hacerse antes del comienzo del función `main`(), o dentro de un función, y al comienzo del mismo (tras en primer {). En el primer caso se trabaja con *variables globales*, cuyos valores son visibles desde cualquier función, en el segundo caso se trabaja con *variables locales* que sólo son visibles en el función en el que han sido declaradas.

```
/* Programa de definición de variables */
/* variable.c */

#include <stdio.h>

int numero_de_matricula;
char nombre[80];

main()
{
int numero_de_puntos;
printf("Numero %d %s %d \n", numero_de_matricula, nombre, numero_de_puntos);
}
```

#### 1.1.2.5 Reglas de cambio de tipo

La regla general establece que si un operador (+,*, ...) tiene operandos de diferentes tipos, el tipo "menor" es promovido al tipo "superior", antes de proceder con la operación, y el resultado se da en forma del tipo superior.

Las reglas básicas son las siguientes (y en el orden indicado):

- si el operando de mayor nivel es `long double`, el otro operando se convierte a `long double`

- si el operando de mayor nivel es `float`, el otro operando se convierte a `float`

- si el operando de mayor nivel es `short`, el otro operando se convierte a `int`

- si el operando de mayor nivel es `char`, el otro operando se convierte a `int`

- si el operando de mayor nivel es `long`, el otro operando se convierte a `long`

En general, las funciones matemáticas de <`math.h`> utilizarán doble precisión.

Las conversiones también tienen lugar en las asignaciones: el valor del lado derecho es convertido al tipo del lado izquierdo, que es el tipo del resultado.

los `char` se convierten a `int` (c=i)
los `long int` se convierten a `char` o `int`, desechando los bits de orden más alto (i=l)
los `float` se convierten a `int` desechando la parte fraccionaria por truncamiento (x=i)

#### 1.1.2.6 Conversiones explícitas (método cast)

Por ejemplo, `sqrt`() definida en <`math.h`> requiere un argumento en doble precisión, si lo que se desea es calcular la raíz cuadrada de un entero, se puede hacer mediante la declaración explícita de tipo:

```
sqrt( (double) n )
```

donde (`double`) es el 'cast' del `int n` que lo presenta como `double`

*cast: (nombre_de_tipo) expresion*

Si el prototipo de la función declara argumentos, la llamada a la función provoca la conversión forzada de argumentos:

si se declara `sqrt2` como: `double sqrt2(double);`

la llamada: `raiz2 = sqrt2(2);`

produce la conversión forzada del `int 2` al `double 2.0` sin necesidad de ningún cast.

Ejemplos con conversiones de tipo:

```
/* cuenta digitos, espacios en blanco y otros caracteres */
/* cuedig.c */
#include <stdio.h>
main()
{
int ndigit[10];
int c, i, nwhite, nother;
nwhite=nother=0;
for(i=0; i<=10; ++i) ndigit[i]=0;

while( (c=getchar() ) != EOF)
      if( c>='0' && c<='9' ) ++ndigit[c-'0']; /* ver nota 1 */
      else if( c==' ' || c=='\n' || c=='\t' ) ++nwhite;
      else ++nother;

printf( "digitos =");
for( i=0; i<10; ++i) printf(" %d", ndigit[i]);
printf(", especios en blanco = %d, otros caracteres = %d \n", nwhite, nother);
}
```

(1)
if( c>='0' && c<='9' )
comprueba si c es un dígito,
si lo es, su valor numérico será: c-'0'
las variables y constantes char son idénticas a las int en expresiones aritméticas

```
/* stoi: convierte un caracter alfanumerico (s) en un entero (i) */
int stoi( char s[])
{
int i,n;
n=0;
for(i=0; s[i]>='0' && s[i]<='9'; ++i) n=10*n+(s[i]-'0');
return n;
}


/* lower: convierte un caracter ASCII a minuscula */
int lower(int c)
}
if( c>='A' && c<='Z' ) return c+'a'-'A'
else return c;
}

en <ctype.h> se definen funciones para cambios de tipo:
```

así por ejemplo,
tolower(c) regresa el valor en minusculas de la variable c (equivalente a la función lower)
isdigit(c) regresa el valor numérico de la variable c (equivalente a if( c>='0' && c<='9' ) )

### 1.1.3 Sentencia printf

Sirve para dirigir la información hacia el dispositivo estándar de salida (la pantalla).
Tiene en siguiente formato:

```
printf( "Texto Formatos Secuencias_de_escape", variables )
```

Formatos

| | |
|---|---|
| %s | carácter |
| %30s | carácter con 30 posiciones |
| | |
| %i | entero |
| %d | entero decimal |
| %6d | entero decimal con 6 posiciones |
| %u | entero sin signo |
| | |
| %f | real (punto flotante) |
| %6.2f | real con 6 posiciones enteras y 2 decimales |
| %e | real (punto fijo = notación científica) |
| %g | real con redondeo del resultado |
| | |
| %o | octal sin signo |
| %x | hexadecimal (dígitos en minúscula) |
| %X | hexadecimal (dígitos en mayúscula) |
| | |
| %n | puntero a un entero |
| %p | dirección completa (segmento : desplazamiento) |

Prefijos

| | | |
|---|---|---|
| h | con d, i, o, x, X | entero corto |
| | con u | sin signo |
| l | con d, i, o, x, X | entero largo |
| | con e, g | doble |
| L | con d, i,, o, x, X | entero largo doble |

Secuencias de escape:

| | |
|---|---|
| \ | indica el comienzo de una secuencia de escape: |
| \n | retorno de carro |

Por ejemplo:
printf ( “%d \t %d \n “, var1, var2 );

Ejemplo de programa en C

```
/* Ejemplo de programa en C */
/* celsius2.c */

#include <stdio.h>
/* Este programa imprime la tabla de conversión de grados Farenheit a Celsius
para fahr = 0, 20, ... 300 */

main()
{
int fahr, celsius;
```

```
int lower, upper, step;

lower = 0;  /* límite inferior */
upper = 300;      /* límite superior */
step = 20;  /* incremento */

fahr = lower;     /* inicializo fahr */

while (fahr <= upper)   /* bucle para hacer la tabla */
{
celsius = 5*(fahr-32)/9;      /* calculo celsius */
printf("%d \t %d \n", fahr, celsius);
      /* imprimo fahr y celsius */
      /* %d formato de fahr = decimal */
      /* \t tabulador */
      /* %d formato de celsius */
      /* \n salto a línea nueva */
fahr = fahr + step;     /* incremento fahr con step */
}
}
```

### 1.1.4 Definición completa de printf

```
int printf(char *format, arg1, arg2, ...)
```

Convierte los valores de las variables en caracteres, los formatea y los dirige a la salida estándar.
Regresa el número de caracteres impresos, o un valor negativo si se produce un error.
La cadena de formato contiene dos tipos de objetos: caracteres ordinarios, que son copiados al flujo de salida; y especificaciones de conversión, para la conversión e impresión de los valores de los argumentos.
Cada especificación de conversión comienza por % y termina con un carácter de conversión, y entre ambos pueden especificarse:

- un signo menos: ajusta a la izquierda del campo el argumento convertido, o
  un signo mas: se imprimirá siempre el signo, o
  un espacio: si el primer carácter no es un signo, se imprimirá un espacio
  un cero: se rellena con ceros la parte izquierda del campo
  un #: especifica una forma alterna de salida:
    o: el primer dígito será cero
    x: los valores distintos de cero se imprimen con 0x
    X: los valores distintos de cero se imprimen con 0X
    e: la salida tendrá siempre un punto decimal
    f: la salida tendrá siempre un punto decimal
    g: no se eliminan los ceros acarreados
- un número: especifica el ancho mínimo del campo (carácter de relleno por defecto = blancos)
- un punto: separa el ancho del campo de la precisión
- un número: especifica el número de caracteres de la parte fraccionaria (reales), el número mínimo de dígitos impresos (enteros), el número máximo de caracteres que serán impresos (cadenas).
- una h si se imprime un entero como short, o una l (ele) si se imprime como long

Los caracteres de conversión pueden ser:

| | | |
|---|---|---|
| d | int | número decimal |
| i | int | número decimal |
| o | int | número octal sin signo |
| x | int | número hexadecimal sin signo y en minusculas |
| X | int | número hexadecimal sin signo y en mayúsculas |
| u | int | número decimal sin signo |
| | | |
| c | int | carácter sencillo |
| s | char | imprime los caracteres de una cadena hasta el carácter \0 |
| | | |
| f | double | [-]m.dddddd<br>número de decimales por defecto = 6 |
| e | double | [-]m.ddddddde±xx |
| E | double | [-]m.ddddddE±xx |
| g | double | se usa %e si el exponente es menor de -4 o mayor que la precisión<br>se usa %f en caso contrario<br>no se imprime el punto y los ceros finales |
| G | double | se usa %E o %f (igual que %g) |
| | | |
| p | void | apuntador |
| | | |
| % | - | se imprime el carácter % |

La amplitud o precisión se puede especificar mediante *, con lo que el valor se calcula convirtiendo el siguiente argumento (que debe ser int):

```
int max;
char s[10];
printf("Valores de la cadena = %.*s", max, s);
```

Imprime max caracteres de la cadena s.

Ejemplo: en la cadena s se almacena el valor "hola, mundo" (11 caracteres).
Los resultados con distintas especificaciones son:

```
:%s:        :hola, mundo:
:%10s:      :hola, mundo:
:%.10s:     :hola, mund:
:%-10s:     :hola, mundo:
:%.15s:     :hola, mundo:
:%-15s:     :hola, mundo    :
:%15.10s:   :      hola, mund:
:%-15.10s:  :hola, mund     :
```

### 1.1.5 Ámbito de las variables

El ámbito de una variable es el conjunto de sentencias desde las cuales se puede acceder a esa variable para obtener su valor (también se denomina visibilidad de la variable).

Un programa en C se divide en partes y en módulos. Las partes están constituidos por las diversas funciones (o subprogramas) definidas por el programador junto con la función `main`(), y por las bibliotecas que se incorporan al mismo. Los módulos son el conjunto de ficheros en donde están escritas las partes o sus componentes.

La definición de variables puede hacerse antes del comienzo del función `main`(), o dentro de un función, y al comienzo del mismo (tras en primer {). En el primer caso se trabaja con *variables globales*, cuyos valores son visibles desde cualquier función, en el segundo caso se trabaja con *variables locales* que sólo son visibles en el función en el que han sido declaradas.

Los valores de las variables definidas en un función, permanecen visibles mientras dura la ejecución de ese función, liberándose después para permitir la utilización de la memoria por ellas ocupada, por otras variables de otros función.

Si se desea que no se libere la memoria, y por tanto su valor permanezca para su utilización posterior al función que las define, es necesario declararlas *estáticas*:

```
static int indice;
```

Las variables definidas al comienzo del programa, fuere de los limitadores { }, son globales y estáticas.

Las variables que se desee acceder desde otros módulos o ficheros enlazados con el programa, deben declararse *externas*.

```
extern int indice;
```

Los tres tipos de accesibilidad de las variables son:

*Globales* (external linkage): son variables accesibles en cualquier parte dentro de un módulo y dentro de otros módulos. Se definen al comienzo del módulo, y fuera de la función `main`.

*Globales de módulo* (file linkage): son variables globales en las que se emplea la declaración `static` en su definición de tipo. La variable es accesible desde cualquier parte dentro del mismo módulo, y no puede ser accedida desde cualquier otro módulo.

*Locales*: son variables accesibles sólo dentro de la función en la que están declaradas, siendo invisibles desde otras funciones o desde otros módulos.

Ejemplo:

```
/* Ejemplo de funcion */
/* funcion.c */

double setup [100]; /* global a todos los módulos */
static reading [100]; /*global sólo para este módulo */

main()
{
int count; /* local dentro de main */
count=0;   /* resto de sentencias de la función main */
}

Mifuncion()
{
int i; /* local dentro de Mifuncion */
i=3;   /* resto de sentencias de Mifuncion */
return i;
}
```

### 1.1.6 Variables estructuradas

**Array (Matrices)**

Son aquellas en las que un conjunto de datos se llama por medio de un único identificador. Son conocidas por vectores (una dimensión) o matrices (varias dimensiones). La identificación de cada elemento de la matriz se realiza por medio de sus índices.

definición:

```
int datos[10];
```

identificación del elemento número 3:

```
dato3 = datos[3];
```

En la definición se indica el número máximo de elementos que puede tener la matriz. Normalmente se comienza el conteo por el número 0, por lo que el vector datos[10] tendrá 11 elementos.

Matrices de 2 dimensiones:

```
Int matriz [2] [3] = { { 11, 12, 13 }, { 21, 22, 23 } }
```

define la matriz:

| | | |
|---|---|---|
| 11 | 12 | 13 |
| 21 | 22 | 23 |

### 1.1.7 Variables de registro

Permiten identificar un conjunto de datos con tipos diferentes mediante un único identificador.

Se define primero la estructura de la variable de registro:

```
struct ANOTACION
{
char nombre[10];
char apellidos[30];
```

```
char calle[30];
shot numero;
float deuda;
}
```

A continuación se definen las variables de registro:

```
struct ANOTACION agenda1;
struct ANOTACION agenda2;
```

De esta forma la variable agenda1 estará formada por las variables simples:

```
agenda1.nombre
agenda1.apellido
agenda1.calle
agenda1.numero
agenda1.deuda
```

a las cuales se puede acceder como a cualquier otra variable en lenguaje C.

Las variables de registro pueden anidarse o unirse entre sí, dando lugar a estructuras complejas.

### 1.1.8 Variables enumeradas

Son listas de entidades que tiene un único identificador, y todos los elementos de la lista son del mismo tipo, y están ordenados.

```
enum DIAS_SEMANA
{
LUN, MAR, MIE, JUE, VIE, SAB, DOM
}dia;
```

### 1.1.9 Operadores

Las expresiones se definen por medio de operandos y operadores, y producen un único valor que es el resultado asignado a una variable o función.

Tipos de operadores

| | |
|---|---|
| Aritméticos | realizan operaciones aritméticas |
| Asignación | transfieren datos de una a otra variable |
| Relacionales | comparan números o caracteres |
| Lógicos | realizan operaciones lógicas (Booleanas) |
| Direccionales | trabajan con direcciones de variables |

| Operadores aritméticos | Nombre | Ejemplo |
|---|---|---|
| * | multiplicación | a*b |
| / | división | a/b |
| % | resto de la división | a%b |
| + | suma | a+b |
| - | resta | a-b |
| () | expresión | (a+b)/c |
| - | cambio de signo | -a |
| ++ | incremento | a++ |
| -- | decremento | c-- |
| Operadores de asignación | | |
| = | directa | x=b |
| -- | decremento sobre x | x-- |
| | x=--b | decrementa b, asigna el resultado a x |
| | x=b-- | asigna b a x, decrementa b |
| ++ | incremento sobre x | x++ |
| += | incremento x en el valor dado | x += 5 |
| -= | incremento en el valor dado | x -= 5 |
| *= | multiplica por el valor dado | x *= 5 |
| /= | divide por el valor dado | x /= 5 |

Ejemplos

x=3

b=6

| | |
|---|---|
| x++ | incrementa el valor de x en una unidad: resultado x=4 |
| x=b++ | asigna a x el valor de b, y luego incrementa n: resultado x=6, b=7 |
| x=--b | disminuye b, asigna el valor resultante a x: resultado b=5, x=5 |
| x+=2 | incrementa x en 2: resultado x=5 |

| Operadores relacionales | |
|---|---|
| comparación | >, <, <=, >= |

| | |
|---|---|
| igualdad, desigualdad | ==, != |

Comparan dos variables, produciendo un resultado lógico:

| | | |
|---|---|---|
| verdadero | true | 1 |
| falso | false | 0 |

```
if (tecla != 27) printf( "no ha pulsado la tecla Esc" );
```

| Operadores lógicos y de comparación de bits | |
|---|---|
| negación | ! |
| y (AND) | & |
| o (OR) | \| |
| o exclusivo (XOR) | ^ |
| desplazamiento hacia la izquierda | << |
| desplazamiento hacia la derecha | >> |
| y (AND) | && |
| o (OR) | \|\| |

Ejemplos

```
a>b && c<d
```

```
int valor=0;
if( valor ) printf( "valor es cero" );
```

```
int valor=0;
if( !valor ) printf( "valor no es cero" );
```

| Operadores direccionales (para punteros) | |
|---|---|
| dirección | & |
| valor | * |
| tamaño de una variable | sizeof |

### 1.1.10 punteros

Las variables de caracteres inicializadas usando punteros (*), es posible que no sean imprimibles:

```
char p1[] = "abcdef";    /* p1[i] es siempre imprimible   */
char *p2  = "abcdef";    /* p2[i] puede no ser imprimible */
```

Los punteros son indicaciones que se incluyen en la utilización de variables de manera que sus valores se asocian a posiciones de memoria.

```
int var =100;                    /* define la variable var */
int *puntero;                    /* define el puntero de var*/

puntero = &var;                  /* puntero obtiene la dirección de var */
```

```
longitud = sizeof(int);          /* determina el tamaño de int */

printf( " var = %d \n", var );   /* da el valor de la variable var */
printf( "*puntero =  %d \n", *puntero );
                                 /* da el valor de var gracias a *puntero */
printf( "&var = %u \n", var );   /* da la dirección de la variable var */
printf( " puntero = %d \n", puntero );
                                 /* da la dirección de var gracias a puntero */
```

### 1.1.11 Precedencia de la ejecución de operadores

La precedencia preestablecida se altera con el empleo de paréntesis ( ) :

```
5 - 3 * 4 = +7
(5 - 3 ) * 4 = +8
```

Tabla de prioridades

| Operador | Denominación | Asociatividad |
|---|---|---|
| ( ) : > [ ] . - | expresión | izq a der |
| - ~ ! *punt & ++ -- sizeof | unitarios | der a izq |
| * / % | multiplicativos | izq a der |
| + - | aditivos | izq a der |
| >> << | desplazamiento de bits | izq a der |
| < <= > >= == != | relacionales | izq a der |
| & ^ \| && \|\| | binarios | izq a der |
| ? : | condicionales | der a izq |
| = *= /= += -= | asignación | der a izq |
| , | evaluación secuencial | izq a der |

### 1.1.12 Sentencias de control del flujo de ejecución

#### 1.1.12.1 Operador condicional

Es una forma de incluir en una única línea una bifurcación del tipo if ... else
La sintaxis es:

*expresión_lógica ? si_expresión_es_verdadera : si_expresion_es_falsa*

Ejemplo:

```
valor_minimo= ( (x<y) ? x : y) )
```

#### 1.1.12.2 Sentencias de bifurcación

#### 1.1.12.3 If

La sentencia if se utiliza para evaluar una expresión de control y ejecutar las sentencias especificadas si la expresión es cierta

*if (expresion_de_control)*
*sentencias*

```
if (c>100) printf ("el valor supera el límite");
```

La sentencia if ... else permite definir dos conjuntos de sentencias, uno a ejecutar si la expresión es verdadera, y el otro si es falsa.

*if (expresion_de_control)*
*sentencias 1*
*else*
*sentencias 2*

```
if (c>100) printf("el valor supera el límite");
else
printf("el valor es inferior a %i", c);
```

o bien:

```
if (c>100)
{
printf("/a");
printf("el valor supera el límite");
}
else
{
printf("/a");
```

```
printf("el valor es inferior a %i", c);
}
```

#### 1.1.12.4 Switch

La sentencia switch permite crear múltiple caminos de bifurcación, en función del resultado de la expresión de control, que necesariamente debe ser un entero o carácter.

*switch (expresion_de_control)*
*{*
*case variable-expresion:*
*sentencias*
*break;*

*case variable-expresion:*
*sentencias*
*break;*

*....*

*case variable-expresion:*
*sentencias*
*break;*
*}*

```
switch (expresion)
{
case 0:
resp = op1 + op2;
break;

case 1:
resp = op1 - op2;
break;

case 2:
resp = op1 * op2;
break;

case 3:
resp = op1 / op2;
break;

default:
printf ("operación erronea");
break;
}
```

La sentencia `break` produce la terminación de la ejecución del bucle interior, pasando el control al siguiente nivel de bucle exterior, a partir del punto de terminación del bucle interior.

#### 1.1.12.5 Sentencias para ejecución de bucles

Se ejecutan las sentencias del bucle hasta que se cumple que una expresión alcanza un valor.

#### 1.1.12.6 For

En esta sentencia el bucle se ejecuta un determinado número de veces.
Es adecuada cuando se conoce el número de veces que se debe ejecutar el bucle.

*for (valor_inicial; valor_límite; incremento)*
*sentencias*

```
for (i=0; i<10; i++)
printf (""el valor es: %i",i);
```

Ejemplo:

```
/* Imprime la tabla de conversión de grados Farenheit a Celsius */
/* celsius3.c */

#include <stdio.h>

main()
{
int fahr;

for (fahr = 0; fahr <= 300; fahr =fahr + 20)
      printf("%3d %6.lf \n", fahr, (5.0/9.0)*(fahr-32));
}
```

#### 1.1.12.7 Do

En la sentencia do la comprobación tiene lugar al final del bucle.
Es adecuada para ejecutar el bucle al menos una vez, y hasta que se alcance una condición.

*do*
*sentencias*
*while (expresion_de_control);*

```
do
{
x++;
printf ("%i\n", x);
}
while (x <= 1000);
```

#### 1.1.12.8 While

En la sentencia `while` la comprobación tiene lugar al comienzo del bucle.
Es adecuada para ejecutar el bucle sólo si se cumple una condición.

*while (expresion_de_control)*
*sentencias*

```
while (a <= 'z')
{
printf ("%c", a++);
}
```

#### 1.1.12.9 Sentencias para terminación de bucles

`break`

`continue`

`goto`

#### 1.1.12.10 Etiquetas

Las etiquetas tienen la misma forma que los nombres de variable, y se escriben terminándolas en dos puntos (:).

Se emplean con la sentencia `goto`
Ejemplo: Salir de una estructura anidada en muchos niveles

```
for (...)
{
     for(...)
     {
          if(desastre) goto error:
     }
}
error:
```

### 1.1.13 Funciones

Las funciones son los elementos constructivos básicos de los programas en C.

Las funciones son subprogramas identificados por un nombre, que producen un resultado, y que son ejecutadas cuando son llamadas por una sentencia contenida dentro de otra función del programa. La única función que siempre debe estar presente dentro de un programa en C, es la función `main`, que debe incluir todo el código que vaya a ser ejecutado.

Las funciones se comunican entre sí a través de las variables de entrada (datos) que recibe la función llamada de la llamante, y de la variable de salida (resultado) que proporciona la función llamada a la llamante. Las variables de entrada se denominan argumentos. La variable de salida está asociada al nombre de la función.

Las funciones pueden estar definidas en el propio programa, por medio de las sentencias escritas por el programador, o tomarse de bibliotecas (libraries, librerias en la traslación directa del término inglés). Ejemplos de estas últimas son `printf`, `sin`, `scanf`, para imprimir con formato, calcular el seno, o examinar un elemento.

Mediante el empleo de funciones se obtienen las siguientes ventajas:

El código empleado en las funciones es reutilizable: esto es, se puede llamar a la misma función desde distintas partes del programa, sin necesidad de volver a escribir el mismo código en cada una de las partes del programa en donde se necesita.

La división del programa en funciones, permite escribir éste en pequeños bloques, consiguiéndose una compilación más rápida.

El programa se divide en bloque más pequeños y manejables.

Las funciones se denominan como las variables, y también deben ser definidas de la misma forma. La definición de la función se denomina función prototipo.

La función prototipo asigna el nombre de la función, los tipos de los argumentos, y el tipo del valor retornado por la función.

La inclusión en la definición de la palabra `void` indica que no se utilizan argumentos, o que no se devuelve un resultado.

La sintaxis de definición de una función es:

*tipo_de_retorno  nombre_de_la_funcion (  lista_de_parametros  );*

Tras la definición de la función, se escribe a continuación el código a ella asociado. Este código se escribe a continuación de la línea de identificación de la función (igual que la línea de declaración pero sin el ; final), y encerrado entre { }.

*tipo_de_retorno  nombre_de_la_funcion (  lista_de_parametros  )*

*{*
*/* sentencias de código de la función */*
*/* la función termina al llegar al símbolo }, o al llegar a una sentencia return */*
*/* pueden existir varias sentencias return en el código de la función */*
*}*

Los elementos empleados en la función son:

*tipo_de_retorno*
indica el tipo de la variable que recoge los resultados de la función
si no se retorna ningún valor, se especifica `void`

*nombre_de_la_funcion*
indica el nombre el que se identifica la función (mismas reglas que para variables)

*lista_de_parametros*
nombres y tipos de las variables empleadas como argumentos

Las funciones que no retornan ningún valor al programa que las llama, son conocidas como funciones tipo `void` o procedimientos.

`valor = cos( angulo );` retorna en cos( ) el valor del coseno de la variable ángulo

`exit(=);` no retorna ningún valor

Es más sencillo realizar funciones que retornen un único valor a través del identificador (nombre de la función), evitando retornar valores mediante argumentos y bloque comunes. Este último tipo de funciones se les denomina procedimientos.

Argumentos de una función son las variables en el programa principal, o en el procedimiento, que hace la llamada a dicha función. En el ejemplo siguiente, en la función *esfera* su único argumento es la variable *radio2*.

Parámetros de una función son las variables que aparecen en su definición. En el ejemplo siguiente, en la función *esfera*, su único parámetro en la variable *radio*.

Los nombres de los argumentos y de los parámetros pueden ser distintos, pero deben tener ser del mismo tipo, y ocupar la misma posición, en su correspondencia uno a uno.

```
Llamada:    argumentos =    arg1,  arg2,  ...    argn
                            |      |             |
Definición: parámetros =    par1,  par2,  ...    parn
```

Los argumentos deben ser declarados solamente en la función que realiza la llamada.

Ejemplo:

```
/* Calculo del volumen de una esfera */
/* esfera.c */

#include <stdio.h>
```

```
#define PI 3.14

float esfera( int radio );

main()
{
float volumen;
int radio2 = 4;
volumen = esfera ( radio2 );  /* Llamada a la función esfera */
printf( "Volumen: %f\n", volumen);
}

float esfera( int radio )    /* Definición de la función esfera */
{
float result;
result = 4*PI*radio*radio*radio/3;
return result;
}
```

La transferencia de los valores de las variables desde la función llamante a la función llamada, puede realizarse pro medio de dos formas:

#### 1.1.13.1 Paso de argumentos por valor

La variable original en la función llamante no sufre cambio de valor. El resultado se envía al nombre de la función. Se transfiere el valor de la variable del programa llamante al llamado, y no se transfiere su dirección de memoria (referencia).

```
/* Calculo de x elevado a y (y = entero) */
/* pasoval.c */

#include <stdio.h>

int xay( int x, int y);

main()
{
int x=2;
int y=8;

printf(" \n Datos de entrada: x = %i \ y = %i", x, y );
printf("\n %i elevado a %i =", x, y, xay(x,y) );
printf("\n Valores finales:  x = %i \ y = %i", x, y );
}

int xay( intx, int y);
int p;
for( p=1; y>0; -y) p= p*x;

printf("n Valores en la funcion xay: x= %i y = %i", x, y);

return(p);
}
```

Resultado:

| | |
|---|---|
| Datos de entrada: x = 2 y = 8 | función main |
| Valores en la funcion xay: x = 2 y = 0 | función xay |
| 2 elevado a 8 = 256 | función main: x,y antes de llamar a xay |

Valores finales: x = 2 y = 8 función main: x,y después de llamar a xay

#### 1.1.13.2 Paso de valores por referencia

La función llamante pasa a la función llamada, la dirección de la variable que contiene el valor de entrada para el cálculo a realizar en la función llamada. Una vez realizado dicho cálculo, el resultado se almacena en la dirección proporcionada, por lo que el valor de la variable de la función llamante se habrá modificado, tomando como valor el resultado de dicho cálculo.

Para recuperar la posición de memoria de una variable, se debe declarar una variable puntero como uno de los parámetros en la definición de la función y en la llamada.

```
/* Paso de argumentos por referencia */
/* pasoref.c */

#include <stdio.h>

int zay( int x, int *y);              /* se usa *y  */

main()
int x=2;
int y=8;
int *y;                               /* puntero de y = valor de y          */
y=&z;                                 /* en y se coloca la dirección de z   */

printf(" \n Datos de entrada: x = %i \ y = %i", x, *y );     /* se usa *y  */
printf("\n %i elevado a %i =", x, *y, xay(x,y) );            /* se usa *y  */
printf("\n Valores finales:  x = %i \ y = %i", x, *y );      /* se usa *y  */
}

int xay( intx, int *y);
int z,p;

z=*y;                                 /* z toma el valor de y */

for( p=1; z>0; -z) p= p*x;

*y = 3

printf("n Valores en la funcion xay: x= %i y = %i", x, *y);  /* se usa *y  */

return(p);
}
```

Resultado:

Datos de entrada: x = 2 y = 8 función main
Valores en la funcion xay: x = 2 y = 3 función xay
2 elevado a 8 = 256 función main: x,y antes de llamar a xay
Valores finales: x = 2 y = 3 función main: x,y después de llamar a xay

### 1.1.14 Recomendaciones para la escritura de programas

1.
Utilice programación estructurada: utilice funciones, dedicando cada función a resolver una determinada necesidad, concreta y bien definida, y con el menor ámbito que sea posible. Cree un programa complejo sobre un conjunto lo más amplio posible de programas sencillos e independientes.

2.
Identifique mediante comentarios cada función:

- nombre de la función (corto y autoexplicativo)
- nombre del fichero en donde reside
- parámetros de entrada
- parámetros de salida
- descripción de la tarea que realiza
- requisitos para su ejecución

3.
Escriba el programa principal (`main`) lo más corto posible.

4.
Emplee código ya escrito, y escriba código que le permita volver a utilizarlo. Esto se traduce en la utilización lo más amplia posible de bibliotecas, y la creación de nuevas bibliotecas (1) con código depurado y suficientemente probado.

5.
Procure mantener siempre un flujo de ejecución del programa de arriba hacia abajo.
En una función se entra por arriba y se sale por abajo. Evite la sentencia `goto`.

6.
Utilice variables locales.
Pase los valores mediante argumentos, evite la utilización de bloques comunes.

7.
Escriba funciones simétricas y neutras:
Si en una función se abre un fichero, debe cerrarse antes de salir de la misma (simetría).
Si en una función se entra con un determinado color del fondo de pantalla, dejar activo ese color antes de salir de la función (neutralidad).

(1)
Una librería es un conjunto de código compilado que puede incorporarse a un programa si éste lo necesita. De esta forma se puede reutilizar código ya escrito y depurado, se incorpora compilado, lo que ahorra tiempo de compilación del programa completo, e impide conocer los algoritmos empleados en su programación.

### 1.1.15 Entrada y salida a periféricos

Periféricos estándar definidos en C

| nombre | código | descripción |
|---|---|---|
| stdin | 0 | teclado |
| stdout | 1 | pantalla |
| stderr | 2 | pantalla |
| stdaux | 3 | puerto serie |
| stdprn | 4 | puerto paralelo |

Funciones de entrada y salida (en `stdio.h`)

| | |
|---|---|
| `printf` | imprime en pantalla |
| `scanf` | lee del teclado |
| `fprintf` | imprime en fichero |
| `fscanf` | lee de fichero |
| `sprintf` | imprime a buffer de memoria una cadena de caracteres (string) |

Además de estas funciones, la biblioteca `stdio` incorpora otras funciones de entrada y salida:

`c = getchar()`
Lee el siguiente carácter desde el dispositivo de entrada (teclado)

`putchar(c)`
Escribe el contenido de la variable entera c como un carácter en el dispositivo de salida (pantalla)

```
/* Copia un carácter de la entrada a la salida */
/* copcar.c */

#include <stdio.h>

main()
{
int c;

c = getchar();    /* Lee un carácter */
while (c != EOF)  /* Sigue leyendo hasta el fin de fichero */
{     /* Comienzo del ámbito del bucle while */
putchar(c); /* Escribe el carácter leido */
c = getchar();    /* Vuelve a leer otro carácter */
}     /* Fin del ámbito del bucle while */
}

/* Cuenta los caracteres entrados */
/* cuecar.c */
#include <stdio.h>

main()
{
long nc;
```

```
nc = 0;
while (getchar() != EOF)
++nc; /* equivalente a nc = nc +1 */
printf("%ld \n", nc);
}
```

Con la función `scanf` es posible obtener uno o varios valores a través del teclado, con un determinado formato, y almacenarlos en las correspondientes variables.

La función `fflush` permite borrar el área de memoria del teclado para recibir nuevos datos.
La función `getch` permite solicitar una tecla sin que se muestre el carácter en pantalla.
La función `gets` permite leer una cadena de caracteres desde el teclado.

### 1.1.16 Punteros

Un puntero es una variable que contiene la dirección de memoria de otra variable o de una función.

Los punteros permiten transferir datos de unos procedimientos a otros, sin realizar copia de los mismos, con lo que se ahorra memoria y tiempo de programación y de ejecución.

Por medio de los punteros se realiza una gestión dinámica de memoria, la cual permite definir el espacio a ocupar por los datos en el momento de la ejecución del programa. Por tanto no es necesario reservar más espacio que el que realmente se necesita en función de los datos que aparecen en el momento de la ejecución.

La utilización de los punteros se realiza en los pasos siguientes:

- declaración del puntero especificando su tipo
- inicialización del puntero, asignándole un valor de partida
- utilización del puntero:
  asignándole un valor
  investigando su valor
  accediendo a los datos a los que apunta

La declaración del puntero se realiza en el bloque de declaración de variables:

```
int *puntero;
```

El operador * indica que se trata de un puntero, `int` indica que se trata de un valor entero.

Un puntero se puede inicializar asignándole una dirección de memoria, o el valor nulo (`NULL`):

```
char *puntero2 = NULL;
```

El operador & devuelve la dirección de una variable.

| | |
|---|---|
| `puntero = &valor` | se almacena en el puntero la dirección de la variable `valor` |
| `*puntero` | proporciona el valor almacenado en la dirección de `puntero` |

#### 1.1.16.1 Ejemplo de utilización de punteros:

```
/* Empleo de punteros */
/* puntero1.c */

#include <stdio.h>

main()
{
```

```
int valor = 100;
int *puntero;

puntero = &valor;

printf( " valor = %d \n", valor );
printf( "*puntero = %d \n", *puntero );
printf( "&valor = %u \n", &valor );
printf( " puntero = %d \n", puntero );
}
```

Salida:
Hola
 valor = 100
*puntero = 100
&valor = 6880112
 puntero = 6880112

| nombre de la variable | dirección | valor |
|---|---|---|
| valor | 0012 | 100 |

| nombre de la variable | dirección | valor |
|---|---|---|
| &valor | | 6880112 |

| nombre de la variable | dirección | valor |
|---|---|---|
| puntero | | 6880112 |

| nombre de la variable | dirección | valor |
|---|---|---|
| *puntero | | 100 |

#### 1.1.16.2 Utilización de punteros con matrices

```
/* Utilización de punteros con matrices */
/* puntero2.c */

#include <stdio.h>

int vector[] = { 10, 20, 30, 40, 50, 60 };

main()
{
int *puntero;
int i;

puntero = &vector[0];

for( i=0; i<6; i++)
{
printf( "vector[%d] = %d \n", i, *puntero );
puntero++;  /* se va a la siguiente dirección = */
      /* siguiente elemento de vector[]   */
}
}
```

Salida:
Hola
 valor = 100
*puntero = 100
&valor = 6880112
 puntero = 6880112
vector[0] = 10
vector[1] = 20
vector[2] = 30
vector[3] = 40
vector[4] = 50
vector[5] = 60

#### 1.1.16.3 Acceder al valor almacenado en la dirección que tiene el puntero:

```
/* Empleo de punteros con cadenas de caracteres */
/* puntero3.c */

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

main()
{
int i;
char nombre[] = "Juan";

for( i=0; i<strlen( nombre ); i++ )/* strlen = longitud de la cadena */
 printf( "*(nombre+%d) = %c \n", i, *(nombre+i) );
}
```

Salida:
*(nombre+0) = J
*(nombre+1) = u
*(nombre+2) = a
*(nombre+3) = n

#### 1.1.16.4 Punteros de punteros:

Utilización de un puntero a otro puntero, incializando el segundo puntero al dato al que apunta el primer puntero.

```
/* Punteros de punteros */
/* puntero4.c */

#include <stdio.h>

main()
{
int valor = 501;
int *puntero1 = &valor; /* *puntero1 = dirección de valor */
int **puntero2 = &puntero1; /* **puntero2 = dirección de puntero1 */
```

```
printf( "\n valor = %d ", valor );
printf( "\n puntero1 = %d ", *puntero1 );
printf( "\n puntero2 = %d ", **puntero2);
}
```

Salida:
valor = 501
puntero1 = 501
puntero2 = 501

| nombre | dirección | valor |
|---|---|---|
| valor | 0015 | 501 |
| *puntero1 | 0016 | 0015 |
| **puntero2 | 0017 | 0016 |

#### 1.1.16.5 Paso de argumentos a funciones mediante punteros

Una de las principales aplicaciones de los punteros es la de transferir datos a otras funciones con sólo enviar la dirección de los argumentos.

```
/* Paso de punteros como argumentos de una función */
/* puntero5.c */

#include <stdio.h>

void cambio( int *puntero1, int *puntero2 );

main()
{
int uno = 1, dos = 2;
int *puntero = &dos;

printf( "uno: %d dos: %d \n", uno, *puntero );

cambio( &uno, puntero );

printf( "uno: %d dos: %d \n", uno, *puntero );
}

void cambio( int *puntero1, int *puntero2 )
{
int t;

t = *puntero1;
*puntero1 = *puntero2;
*puntero2 = t;
}
```

Salida:
uno: 1 dos: 2
uno: 2 dos: 1

#### 1.1.16.6 Utilización de punteros a variables de registro, y paso como argumento a una función:

```
/* Puntero a variable de registro */
/* puntero6.c */

#include <stdio.h>

struct tipoagenda
{
char nombre[15];
char apellido1[15];
char apellido2[15];
int edad;
float deuda;
};

void ficha( struct tipoagenda *agenda );

main()
{
static struct tipoagenda agenda1 = { "Juan", "de", "Juanes", 29, 123.45 };

ficha( &agenda1 );
}

void ficha( struct tipoagenda *agenda2 )
{
printf( "Nombre: %s \n", agenda2->nombre );
printf( "Primer apellido: %s \n", agenda2->apellido1 );
printf( "Segundo apellido: %s \n", agenda2->apellido2 );
printf( "Edad: %i \n", agenda2->edad );
printf( "Deuda: %6.2f \n", agenda2->deuda );
}
```

Salida:
Nombre: Juan
Primer apellido: de
Segundo apellido: Juanes
Edad: 29
Deuda: 123.45

### 1.1.17 Manejo de ficheros para entrada/salida

Existen dos grandes grupos de ficheros:

Con formato: tienen una organización interna formada con estructuras repetitivas. Ejemplo de fichero en este grupo es el de una tabla de datos. La información se alacena dividiéndola en registros, cada uno de los cuales a su vez se divide en campos. Cada dato se encuentra almacenado en un campo.

Ejemplo:

| | | | |
|---|---|---|---|
| Fichero: | direcciones | | |
| Registros: | Persona1 | Persona2 | ... |
| Campos: | nombre | nombre | |
| | apellido1 | apellido1 | |
| | apellido2 | apellido2 | |
| | direccion | direccion | |
| | telefono | telefono | |

Sin formato (de bloques o sin tipo): carecen de una estructura repetiviva uniforme a lo largo de toda su extensión. Un ejemplo de fichero de este grupo es el de un fichero binario ejecutable.

Tipos de acceso:

| | |
|---|---|
| secuencial: | se recorren todos los registros hasta llegar al buscado (cinta) |
| directo (o aleatorio): | se direcciona el registro al que se accede (memoria, disco) |

| | |
|---|---|
| lectura | se leen datos |
| escritura | se escriben datos |

| | | | | | |
|---|---|---|---|---|---|
| formato: | texto | código: | ASCII (DOS) | | |
| | | | ANSI (Windows) | | |
| | | | otros | | |
| | números | código: | binario | entero | corto |
| | | | | | largo |
| | | | | real | simple |
| | | | | | doble |
| | | | Gray | | |
| | | | otros | | |

### 1.1.18 Operaciones para trabajo con ficheros

1.

Declarar una variable que identifique el fichero: se la denomina canal de lectura / escritura. Esta variable es la que se usará para identificar el fichero en el programa.

Se pueden tener varios canales abiertos simultáneamente, hasta el número máximo determinado por los ficheros abiertos por otras aplicaciones y por el límite impuesto en el sistema operativo.

Es el identificador interno del fichero.

Se declara como un puntero tipo FILE:

```
FILE *canal_1
```

2.
Declarar una variable que contenga el nombre del fichero en el disco o cinta.
Es el identificador externo del fichero.

3.
Abrir en canal de entrada / salida = asigna el identificador externo (nombre) al identificador interno (número).
Se realiza con la función `fopen`. Retorna el puntero `NULL` si se produce un error.

Modos de apertura:

| Modo | Descripción |
|---|---|
| r | -- |
| read | abre un fichero existente para lectura |
| w | crea y abre un fichero nuevo para escritura |
| write | abre y borra el contenido de un fichero existente para escritura |
| a | crea y abre un fichero nuevo para añadir registros |
| append | abre un fichero existente para añadir registros al final del mismo (escritura) |
| r+ | -- |
| | abre un fichero existente para lectura y escritura |
| w+ | crea y abre un fichero nuevo para lectura y escritura |
| | abre y borra el contenido de un fichero existente para escritura |
| a+ | crea y abre un fichero nuevo para lectura y escritura |
| | abre un fichero existente para lectura y escritura, añadiendo registros al final |

Modificadores (sufijos):

| | |
|---|---|
| b | se abre en modo binario (números) |
| t | se abre en modo texto (caracteres) |

4.
Escribir o leer datos del fichero.

5.
Cerrar el fichero, eliminando el canal abierto en el punto 1.

Ficheros de texto:
Son ficheros de acceso secuencial organizados en registros (líneas) separados por caracteres de retorno de carro (CR), y dentro de cada línea los campos están separados por otro tipo de carácter que puede ser variable (espacio, coma, punto y coma, tabulador, ...).

`fputs`
Escribe en un fichero una cadena de caracteres o una matriz cuyo último elemento sea el carácter nulo. Devuelve un número no negativo o la constante EOF, NULL en caso de error.

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

```
int fputs( char *texto, FILE *puntero_fichero );
```

Ejemplo:

```
/* Crea y escribe un fichero de texto */
/* fichero1.c *

#include <stdio.h>

main()
{
FILE *canal;

if( (canal = fopen( "c:\\fichero.txt", "w")) != NULL )
{
fputs( "Este texto se escribe en el fichero", canal );
fclose( canal );
}
else
printf( "No se ha podido abrir el fichero \n");
}
```

## fgets

Lee un registro de un fichero, definido entre la posición del cursor y el carácter \n, o hasta el número de caracteres especificado en la llamada de la función. Devuelve el texto leido, o el puntero NULL en caso de error.

```
#include <stdio.h>

char fgets( char texto, max_caracteres, FILE *puntero_fichero );
```

Ejemplo:

```
/* Lectura de un fichero de texto */
/* fichero2.c */

#include <stdio.h>

main()
{
FILE *canal;
char texto[80];

if( (canal = fopen( "c:\\fichero.txt", "r")) != NULL )
{
do
      {
      fgets( texto, 80, canal );
      printf( "\n %s", texto);
      } while( feof( canal ) == 0);
fclose( canal );
}
else
printf( "No se ha podido abrir el fichero \n");
}
```

## rewind

Sitúa el cursor de lectura de ficheros al inicio del primer registro.

### 1.1.19 Manejo de ficheros en CVI

En los sistemas de adquisición de datos es habitual el empleo de ficheros en disco para almacenar o recuperar los datos que se obtienen el las mediciones realizadas en un proceso.

Las denominaciones entrada / salida E/S (I/O input / output) se suelen referir al ordenador como elemento de referencia en un sistema de adquisición de datos, y al fichero en una operación de escritura / lectura E/L, cuando se habla de operaciones con ficheros.

Los ficheros se identifican a través de los identificadores (file handlers), que son número que se asocian a los mismos para realizar las operaciones de entrada / salida, sobre un fichero en particular.

#### 1.1.19.1 Abrir ficheros

```
int File_Handle =
OpenFile (char File_Name [], int Read/Write_mode, int Action, int
File_Type);
```

#### 1.1.19.2 OpenFile

en una función que le indica al computador que se va a realizan una operación de E/S sobre un fichero cuyo nombre se define en `File_Name`, y asociar a dicho fichero un identificador cuyo valor se incorpora a la variable `File_Handle`.

En el caso de que el fichero ya existiera, se abre y se le asocia el correspondiente valor del identificador.

En el caso de que no existiera, se crea un nuevo fichero con el nombre indicado y se le asocia un valor al identificador. Si no es posible crear el fichero (disco lleno, nombre no válido de fichero, ...) se retorna un identificador de valor -1.

En ambos casos, el fichero se referenciará a partir de entonces por medio del identificador.

| Parámetro | =0 | =1 | =2 |
|---|---|---|---|
| Read/Write Mode | Abrir fichero L/E | Abrir fichero L | Abrir fichero E |
| Action | Borrar contenidos | Agregar datos | No borrar contenidos |
| File_Type | binario | ASCII | N/A |

N/A = no aplicable.

#### 1.1.19.3 Cerrar fichero

```
int status = CloseFile (int File_Handle);
```

#### 1.1.19.4 CloseFile

cierra el fichero referenciado por el identificador del fichero

Se retorna un valor (0) se si cierra, y (-1) si no es posible cerrarlo.
Todo fichero abierto durante la ejecución de un programa debe ser cerrado antes de que se termine la ejecución de dicho programa.

#### 1.1.19.5 Escribir en un fichero

```
int n = WriteFile (int File_Handle, char Buffer[], int Count[]);
```

#### 1.1.19.6 WriteFile

escribe los datos almacenados en la variable `Buffer` en el fichero especificado por `FileHandle`.
`Count` indica el número de bytes a escribir en este fichero.
Los conjuntos de caracteres (strings) que utiliza CVI se termina con el carácter `NULL`, por lo que se puede emplear la funcion StringLength para determinar el número de bytes a escribir.

Antes de escribir datos numéricos en un fichero, se deben convertir en texto (caracteres), usando la función `Fmt`, por lo que se requiere una primera llamada a la función `Fmt` y una segunda a la función `WriteFile`. Un método alternativo es usar `Fmt` para escribir sobre el fichero directamente:

```
int n =
FmtFile (int File_Handle, char Format_String[], source1, source2,
... sourceN);
```

#### 1.1.19.7 FmtFile

Esta función mueve los datos provenientes de `sourcex`, a través de formato definido en `Format_String`, y los coloca en el fichero referenciado por `File_Handle` (el valor se crea al ejecutar `OpenFile`).

Esta función se ejecuta de la misma manera que `Fmt`, salvo que se escribe sobre el fichero en disco, en vez de en memoria RAM.

```
int n = ReadFile (int File_Handle, char Buffer[], int Count);
```

#### 1.1.19.8 ReadFile

Se leen datos de un fichero, que debe haber sido abierto previamente con `OpenFile`.
Los datos leídos se colocan en la variable Buffer que debe haber sido declarada previamente.
`Count` es una variable entera que indica el número máximo de bytes a ser leídos.
La función `ReadFile` devuelve un valor entero que indica el número de bytes que se han leído realmente.
Tras la lectura, los datos se almacenan en forma de caracteres, por lo que para poder procesarlos como numéricos es necesario utilizar la función `Scan`.

Un método alternativo es utilizar la función `ScanFile` para convertir directamente los datos en disco en valores numéricos en memoria RAM:

```
int n =
ScanFile (int File_Handle, char Format_String[], target1, target2,
... targetN);
```

Los datos leídos de disco se colocan en las variables `targetx`.

#### 1.1.19.9 Lectura de un fichero binario y escritura en una variable entera

Se leen 100 valores enteros de un fichero binario, y se almacenan en el vector `readings`, siendo cada entero de 4 bytes.

1 byte = 1 carácter = 8 bits

```
int readings [100];
int file_handle;

dile_handle = OpenFile ("TEST.DAT", 1, 2, 0);
ScanFile (file_handle, "%100i>%100i", readings);
CloseFile (file_handle);
```

#### 1.1.19.10 Lectura de un fichero binario y escritura en una variable real (coma flotante)

Se trabaja sobre la base de que cada número real tiene una longitud de 8 bytes.

```
double waveform[100];
int file_handle;

dile_handle = OpenFile ("TEST.DAT", 1, 2, 0);
ScanFile (file_handle, "%100f>%100f", waveform);
CloseFile (file_handle);
```

#### 1.1.19.11 Lectura de un fichero ASCII con numeros separados por comas, y escritura en una variable real

El contenido del fichero es del tipo indicado en la siguiente muestra:

```
1024 12, 28, 63, ... 2, 14, 66 <EOF>
```

donde el primer valor identifica en número de elementos a leer del fichero, y <EOF> indica el carácter de final de fichero (End Of File).
`ScanFile` lee el número de elementos y coloca el valor leído en la variable `count`.
`ScanFile` lee los valores de cada uno de los elementos, asociando el valor de `count` al asterisco en la definición de formato: *f[x], con el modificador `x` para ignorar los separadores (las comas).

```
double values[1024];
int file_handle;
```

```
int count;

dile_handle = OpenFile ("TEST.DAT", 1, 2, 1);
ScanFile (file_handle, "%s>%i", &count); /* se lee el número de
elementos */
ScanFile (file_handle, "%s>%*f[x]",   count, values); /* se leen
los elementos */
CloseFile (file_handle);
```

### 1.1.20 Referencias:


[1]
El lenguaje de programación C
Kernighan, B.W.; Ritchie, D.M.
Prentice Hall, 1991

[2]
LabWindows/CVI Basics Course Manual
National Instruments, 1995.

[3]
Lenguaje C
Moldes Teo, F.J.
Anaya, 1994.

### 1.1.21 Biblioteca ANSI

Encabezados (headers): se incluyen mediante `#include <header>`

| | |
|---|---|
| <assert.h> | diagnósticos |
| <ctype.h> | clasificación de caracteres |
| <errno.h> | |
| <float.h> | límites de constantes reales |
| <limits.h> | límites de constantes reales |
| <locale.h> | |
| <math.h> | funciones matemáticas |
| <setjmp.h> | saltos no locales |
| <signal.h> | señales |
| <stdarg.h> | listas de argumentos de variables |
| <stddef.h> | |
| <stdio.h> | entrada y salida estandar (standard input/output) |
| <stdlib.h> | utilidades |
| <string.h> | funciones para cadenas |
| <time.h> | fecha y hora |

## *1.2 CVI básico*

LabWindows/CVI es un entorno de desarrollo de aplicaciones de instrumentación basado en el lenguaje C.

El entorno de desarrollo es el conjunto de utilidades, tales como editores de texto, compilador, enlazador, editor de interfaz gráfica; que permite la creación de las aplicaciones de instrumentación que puedan ser ejecutadas de forma aislada.

Dentro del entorno de desarrollo, la preparación de la aplicación de instrumentación virtual se desglosa en la definición de sus contenidos distribuidos en varios ficheros. El conjunto de los ficheros empleados dentro del entorno de desarrollo, se conoce como proyecto, y a su vez genera un nuevo fichero (con extensión .prj) que engloba direcciones hacia los ficheros individuales (.uir, .c, .h y otros).

### 1.2.1 Proyecto

Un proyecto es un conjunto de ficheros que componen la aplicación desarrollada cuando son compilados(*compiled*) y enlazados(*linked*). Los ficheros pueden ser :

- **Extensión .c :** Ficheros fuente de código C.
- **Extensión .uir :** Ficheros que contiene la interfaz gráfica.
- **Extensión .h :** Ficheros fuente de cabecera, escritos en código C. Aquí se almacenan los valores de los nombres de los controles(*constant name*) así como los nombres de las funciones *callback*.

- **Extensión .dll :** Ficheros que almacenan librerías de enlace dinámico de Windows.

- **Extensión .obj :** Ficheros de módulo de tipo objeto.

### 1.2.2 Ventana de proyecto

Cuando entramos en entorno de programación LabWindows/CVI, lo primero que aparece es lo que se llama *ventana de proyecto*. Desde esta ventana abrimos, editamos, construimos, ejecutamos y salvamos proyectos.

Un ejemplo de ventana de proyecto es el que aparece en la Figura 1:

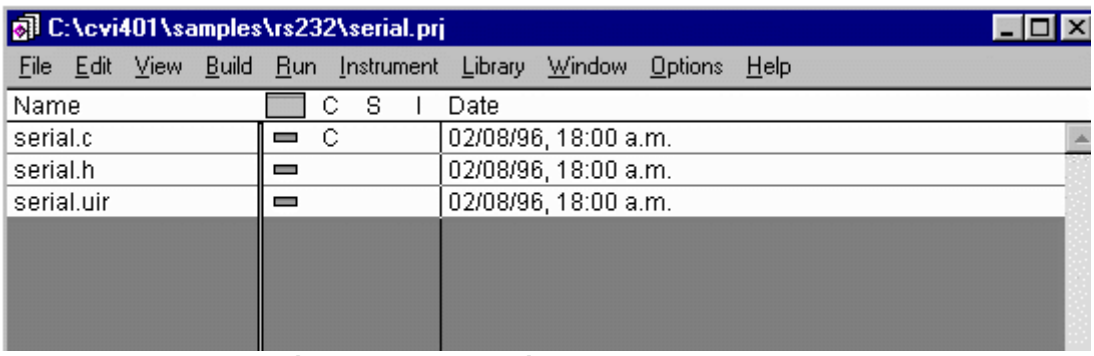


**Figura 1.- Ventana de proyecto CVI**

En esta ventana aparecen una serie de iconos destinados a facilitar algunas de las operaciones más típicas que se realizan sobre ficheros. Estos iconos están representados por letras :

- **C :** Indica que el archivo no ha sido compilado, ha sido modificado desde la última vez que se compiló o se marcó específicamente para su compilación.

- **S :** Indica que el archivo ha sido modificado desde la última vez que fue salvado.

- **I :** Indica que el archivo contiene código fuente de un controlador de un instrumento.

- **A:** Indica que el archivo está asociado a un controlador de un instrumento cargado.

Para examinar el contenido de cualquiera de los archivos que conforman un proyecto no tenemos más que hacer doble click en nombre del archivo.

### 1.2.3 Ventana standard Entrada/Salida (I/O)

Mientras un programa se está ejecutando, con frecuencia se sacan mensajes por pantalla. LabWindows/CVI puede desplegar mensajes de texto y en general datos por la ventana standard de entrada/salida.

LabWindows/CVI puede a su vez aceptar entrada de datos desde esta ventana. El número de líneas que se permite desplegar en esta ventana, así como otras características, puedes configurarlas desde el comando **Environment** en el menú **Options**.

Para borrar el contenido de la ventana se usa el comando **Clear Window** en el menú **Options**, o la función *Cls* de la librería *Utility*.

La forma de desplegar mensajes en esta ventana se consigue mediante la función *printf* , vista en el primer punto.

### 1.2.4 ¿Cómo abrir un proyecto?

El primer paso cuando entramos en el entorno LabWindows/CVI es abrir un proyecto. LabWindows/CVI guarda la dirección del último proyecto y lo presenta en la ventana de proyecto por defecto cuando se carga.

Desplegando el menú **File** se encuentra la opción **Open**. Desplegando este submenú se puede encontrar la opción deseada :

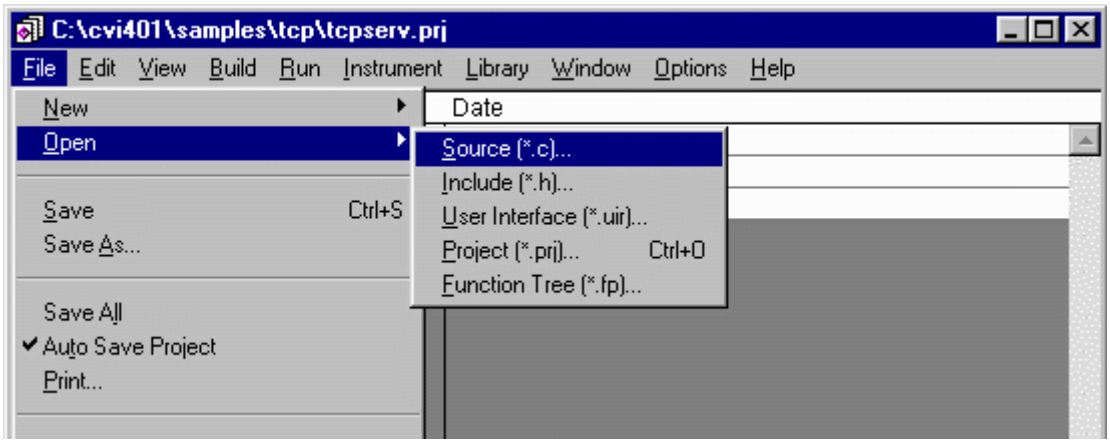


**Figura 2.- Abrir un proyecto en CVI**

Al igual que se puede abrir un proyecto, podemos abrir cualquier tipo de archivo sin más que seleccionar una extensión u otra.

Una vez que tenemos el proyecto cargado, los archivos aparecen en la ventana de proyecto.

### 1.2.5 ¿Cómo crear un nuevo proyecto?

La forma de crear un nuevo proyecto consiste en desplegar el menú **File** de la ventana de proyecto y seleccionar el submenú **New,** tal y como muestra la Figura 3:

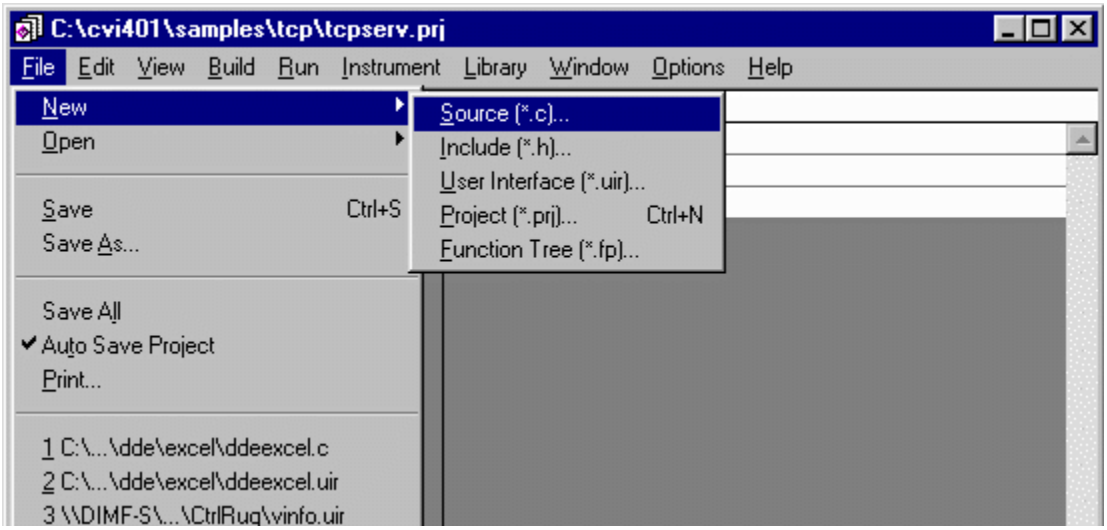


**Figura 3.- Crear un nuevo proyecto nuevo**

Al igual que se puede crear un nuevo proyecto, podemos crear cualquier tipo de archivo sin más que seleccionar una extensión u otra. Cuidado porque cuando generamos un nuevo archivo, no se incluye directamente en el proyecto, sino que hay que incluirlo mediante el comando **Add Files To Projec**, que se encuentra en el menú **Edit**:

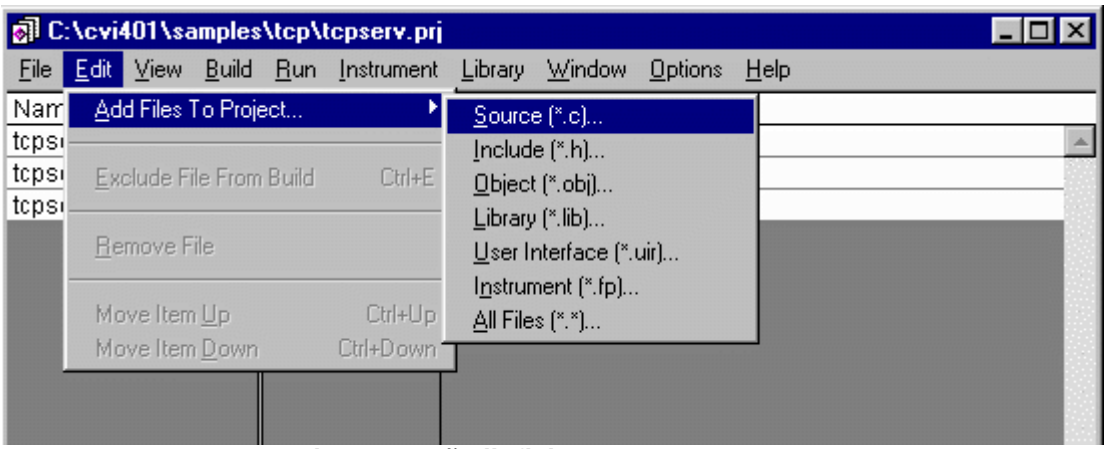


**Figura 4.- Añadir ficheros a un proyecto**

También es recomendable salvar el proyecto y los archivos que vayamos generando.

### 1.2.6 ¿Cómo crear una interfaz gráfica?

La interfaz gráfica es el conjunto de objetos tales como paneles, controles, gráficos que aparecen cuando ejecutamos nuestra aplicación.

La interfaz gráfica viene representada por los archivos de extensión .uir. Cuando accedemos a cualquiera de estos archivos se despliega el *editor de interfaz gráfica*. En este editor se da facilidad para crear paneles, controles y en general para definir el aspecto que tendrá nuestra interfaz.

En la Figura 5 se muestra un ejemplo de editor de interfaz gráfica :

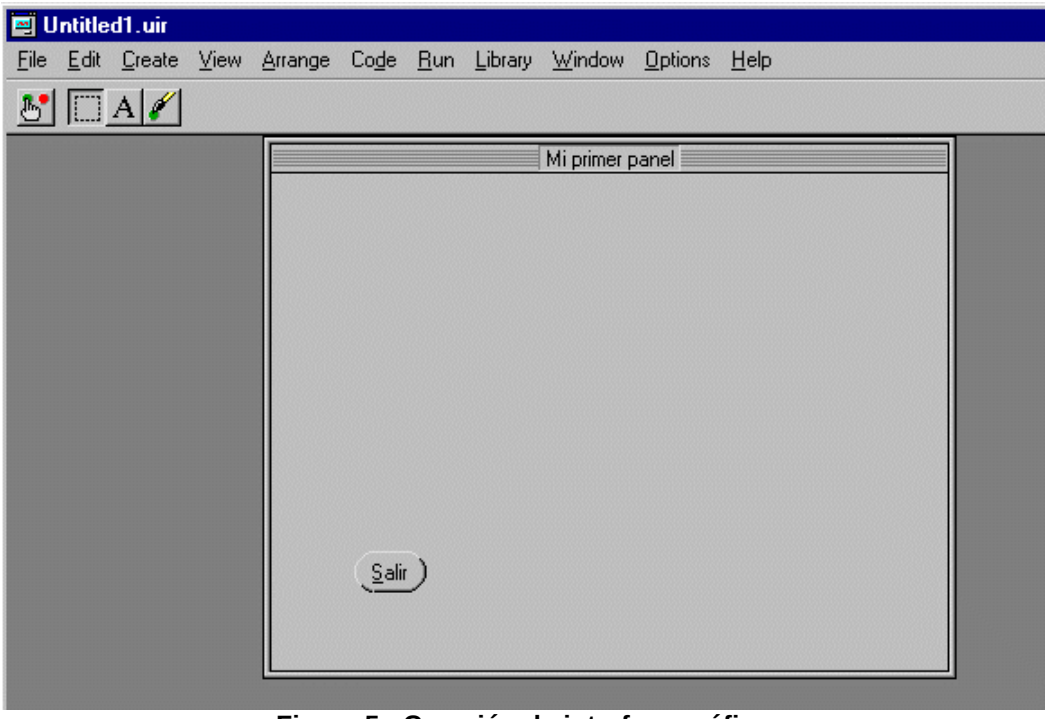


**Figura 5.- Creación de interfase gráfica**

Lo primero que tenemos que hacer es crear un panel en el que situar los controles, mediante el comando **Panel** del menú **Create** desde el editor de interfaz gráfica.

Una vez que tenemos creada la interfaz gráfica, lo siguiente que tenemos que hacer es definir las características de los elementos que hemos creado. Para acceder a las características, hay dos métodos :

- Haciendo **doble-click** en la superficie del control o panel.
- Pulsando intro situándonos encima del control.

De todas las características que podemos modificar en un control, hay dos que son comunes para todos los controles y paneles:

- **Constant Name :** También llamado *resource ID* o identificador. Es el identificador del panel o control.

  Será muy importante porque cuando tengamos en un panel de ID(**PNL**) un control con ID(**NUM**), el nombre que usaremos cuando nos refiramos al control desde el código fuente será **PNL_NUM**.

- **Callback function :** Simplemente decir que en este apartado se almacena el nombre de la función que se invocará cuando ocurra un evento determinado. Esto se clarificará más adelante cuando se vea el concepto de evento.

A modo de introducción, decir que LabWindows/CVI requiere una programación orientada a objetos. Esto significa que nosotros escribiremos una serie de sentencias para cada control que se ejecutarán independientemente del resto.

La forma de desencadenar este tipo de funiones es mediante los eventos, o acciones que realizamos sobre nuestra interfaz, tales como presionar un botón, pinchar con el ratón, modificar el estado de u control, introducir un valor,...

Para colocar todos los controles que deseemos, despliegamos el menú **Create,** o pulsamos el botón derecho del ratón:


**Figura 6.- Generación de controles para la interfase gráfica**

Un ejemplo de cuadro de dialogo que aparecería si tuviéramos un gráfico aparece recogido en la Figura 7:

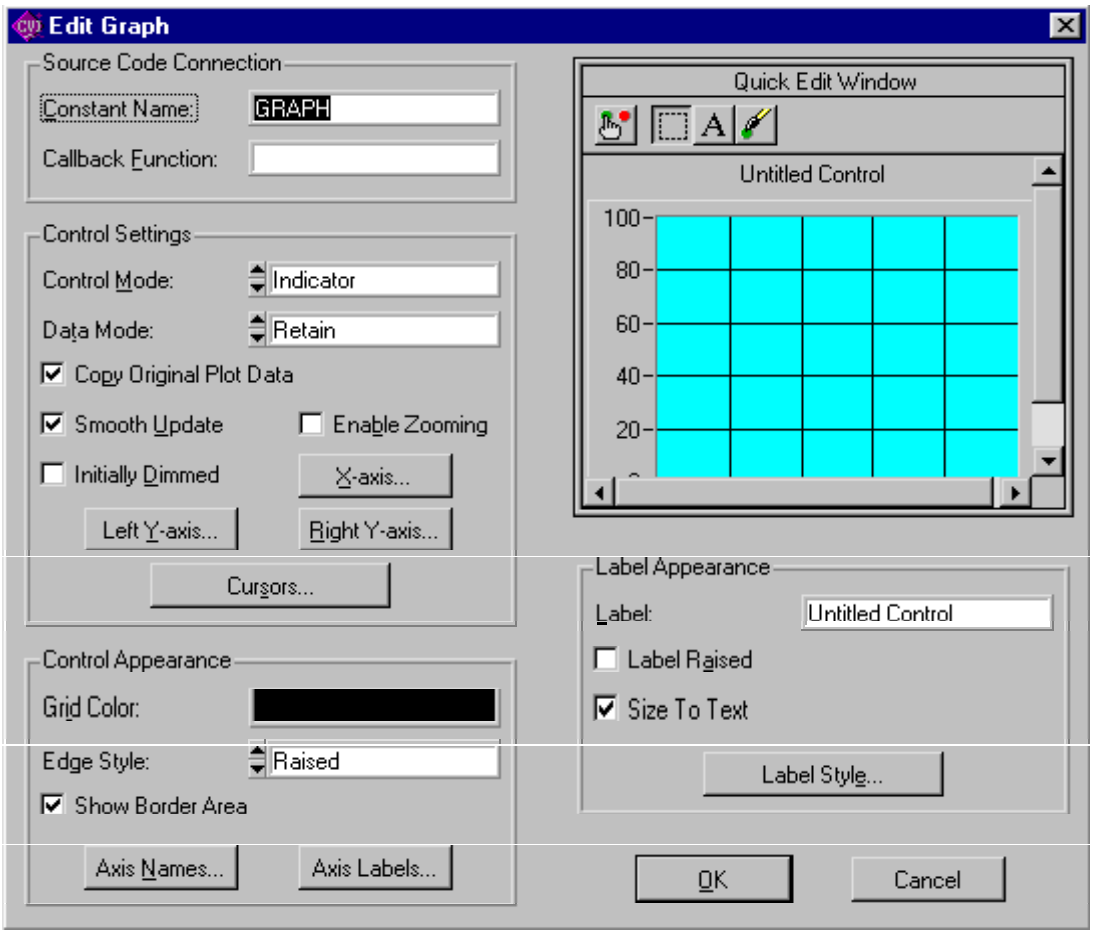


**Figura 7.- Cuadro de dialogo para control de gráfico**

Los ID de todos los elementos junto con los nombres de las callback se almacenan en un archivo de cabecera (.h) de forma automática cuando salvamos el archivo (.uir) correspondiente.

Lo siguiente que tenemos que hacer es escribir el código fuente en C.

### 1.2.7 Ventana de código

En esta ventana se almacena el código fuente del programa que vamos a desarrollar. Esta ventana se comporta como cualquier editor de texto.

LabWindows/CVI da la posibilidad de crear un esqueleto de programa en C desde el editor de interfaz gráfica, mediante el comando **Generate All Code** del menú **Code**.

## *1.3 CVI avanzado*

### 1.3.1 Opciones avanzadas de la ventana de proyecto

En este apartado veremos algunas características más detalladas acerca de las posibilidades del menú de la ventana de proyecto.

### 1.3.2 Objetivo de nuestro proyecto

Lo primero que nos tenemos que plantear cuando desarrollamos un proyecto es la finalidad del mismo. Es en esta ventana y mediante la opción **Build \ Target** el método de selección. Las posibilidades que tenemos las tenemos en la Figura 8:

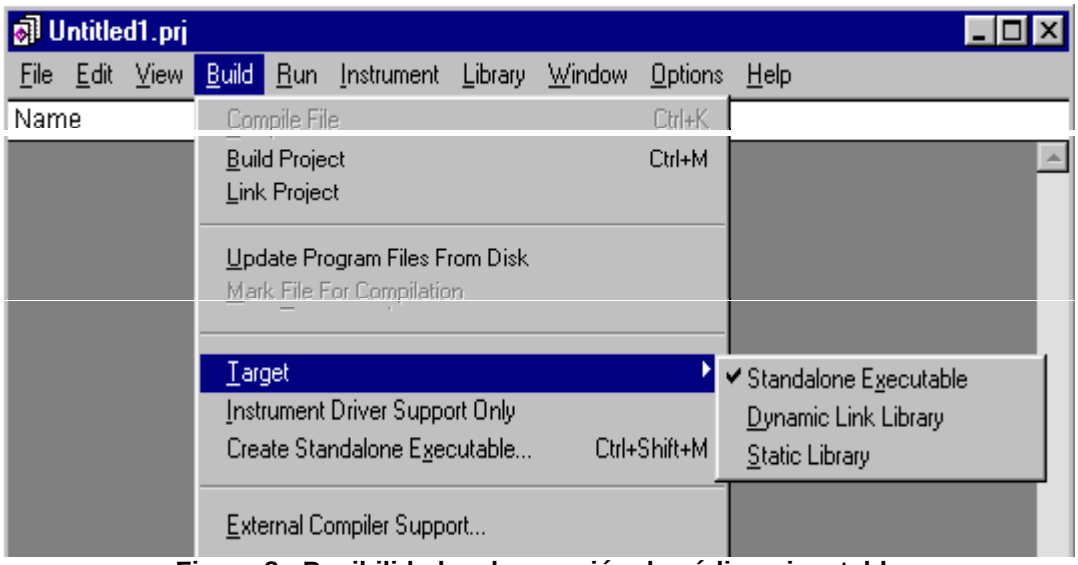


**Figura 8.- Posibilidades de creación de código ejecutable**

Standalone executable : Ejecutable típico .exe.

Dinamic link library : Librería dinámica de windows .dll.

Static Library : Librería típica .lib.

Además de esto, y una vez que tengamos completado nuestro proyecto, tenemos diferentes opciones a la hora de almacenar nuestro proyecto. Según queramos construir sólo el ejecutable (**create standalone executable**), un kit de discos de distribución(**create distribution kit**) o un compilador sin necesidad del entorno LabWindows/CVI(**external compiler support**).

### 1.3.3 Predeterminar las características del compilador

Aunque el entorno establece unas condiciones iniciales y unos valores por defecto para compilar, nosotros podremos modificarlas, junto con las características de la ejecución de los programas, accediendo al menú **Options.**

### 1.3.4 Opciones avanzadas de la ventana de código

En la ventana de código aparecen una serie de funciones que serán útiles cuando realicemos la depuración de nuestro programa. Así como un grupo de comandos que facilitan el análisis del código.

### 1.3.5 Depurar errores

Cuando compilamos y ejecutamos un programa es muy común que nos salgan errores. Este entorno contiene una serie de herramientas que ayudan a su detección :

*Toggle exclusion* : Esta opción se encuentra en el menú **Edit**, aunque también se puede seleccionar desde el teclado mediante Ctrl+E. Con esto conseguimos eliminar durante la compilación líneas de código. Cuando haya una línea de código excluida, su color cambiará.

*Tag* : Se encuentra en el menú **Edit** también. Con ella establecemos unas marcas o banderas por todo el código que ayudarán cuando haya muchas sentencias. Estos elementos aparecerán como franjas verde fosforescente en la parte izquierda de la pantalla.

*Breakpoints* : Estos elementos se usan cuando queremos probar si la ejecución del programa se realiza correctamente o no, y qué fallos ocurren. Aparecen como puntos rojos a la derecha del código, y se seleccionan pinchando con el ratón en la zona o mediante el menú **Run**.

Será imprescindible que haya una línea de código para poder situar un breakpoint.

*Ventana de variables* : Con esta ventana, a la que podemos acceder desde el menú **Window** observamos el valor de las variables en las distintas partes de un programa. Asociada a esta ventana está la **Watch expresion**, en la que nosotros podremos diseñarnos un conjunto de variables para analizar.

Esta ventana de variables se combina muy bien con los breakpoints, dado que si paramos la ejecución obtendremos los valores de las variables en ese momento.

### 1.3.6 Opciones avanzadas del editor de interfaz gráfica

Lo primero que hay que comentar sobre esta pantalla es que aparecen 4 iconos debajo de la barra de menú, que servirán de mucha utilidad. Si empezamos desde la parte izquierda :

Icono que permite conocer los eventos asociados a los controles. No se puede editar el panel en este modo.

Icono que permite editar el panel, colocar controles sobre él y nombrarlos.

Icono que da la posibilidad de cambiar las etiquetas de los controles.

Icono que representa una paleta de colores,  que con solo pinchar se transmite al resto del panel.

#### 1.3.6.1 Edit

En este menú se encuentran las funciones típicas de copiar y pegar elementos, junto con una serie de funciones relacionadas con la creación de barras de menú, el control del orden de tabulación, ...

#### 1.3.6.2 Create

En este menú tenemos todos los tipos de controles que nos podrán aparecer en programas de CVI. También se da la posibilidad de crear un panel o una barra de menú. Esta opción es igual que si pulsamos el botón derecho del ratón en cualquier parte del panel.

#### 1.3.6.3 Arrange

Con este menú se permite configurar la apariencia de los controles en nuestro panel. Tendremos comandos del tipo: alinear, distribuir, controlar el orden de permanencia en el panel...

#### 1.3.6.4 Code

Una vez que tengamos el panel completo seleccionamos **Generate / All Code** para que el entorno genere un esqueleto de código C. Si lo que necesitáramos es generar una parte del código, seleccionaríamos la opción correspondiente, como por ejemplo, **Main function**, **control callback**, Figura 9

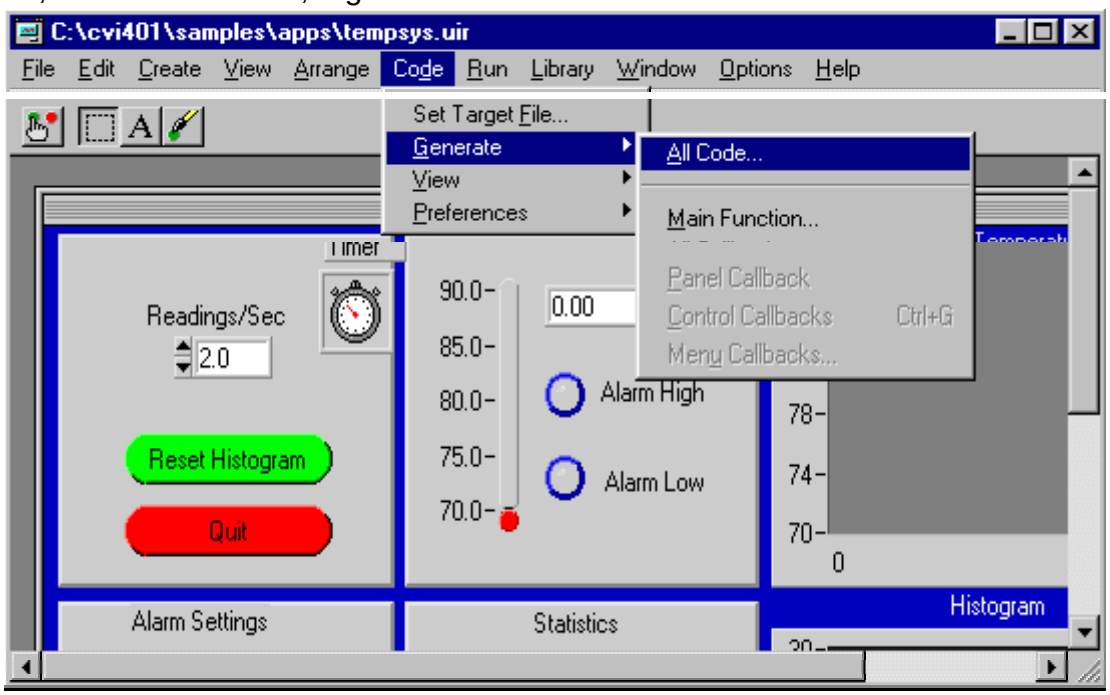


**Figura 9.- Generación automática de código fuente**

La otra opción es la del submenú **preferences** en la que decidimos el tipo de evento que queremos que salga en las callbacks, cuando se genere. Una vez que tengamos todos los elementos definidos, invocaremos a la función generate code.

Mediante la opción Set Target File elegimos el archivo a dónde queremos escribir el esqueleto de código C.

#### 1.3.6.5 Control Mode

Esta característica que aparece en la ventana de características de los controles especifica qué nivel de interacción tiene el usuario con el panel. Esto significa que según el modo de nuestro control nosotros podremos generar una serie de eventos u otros.

Los eventos que podemos seleccionar son :

*Indicator:* En este modo no se puede interactuar con el control, sólo muestra la información para la que esté diseñado. Es un modo muy típico para la presentación de resultados.

*Hot:* En este estado tenemos nivel total de interacción con el control. Genera un EVENT_COMMIT cuando modificamos su estado. Modo típico para los botones de comando, switch, toggle button, ...

*Validate*: Este modo también genera un EVENT_COMMIT, pero después de verificar los rangos de los controles numéricos.

*Normal* : Así podemos actuar con el control.

### 1.3.7 Eventos

Ya se ha visto el concepto de calllback, y estrechamente relacionado están los eventos. Un evento es una acción que el usuario realizará sobre el panel y que invocará a una callback.

Los tipos de eventos que existen son los que se muestran a continuación.

| | |
|---|---|
| ***1.*** *EVENT_COMMIT* | Generamos un evento así cuando pulsamos un botón con el ratón, cuando nos situamos encima de un control y pulsamos intro. Se podría decir que es el evento tipo o general. |
| ***2.*** *EVENT_VAL_CHANGED*<br>***3.*** | Generamos un evento así cuando variamos de alguna forma el valor del control, también su estado. |
| ***4.*** *EVENT_LEFT_CLICK* | Un evento así especifica que únicamente se generará un evento así cuando pulsemos el botón izquierdo del ratón. |
| ***5.*** *EVENT_LEFT_DOUBLE_CLICK*<br>***6.*** | Idem que el anterior pero haciendo doble click. |
| ***7.*** *EVENT_RIGHT_CLICK* | Idem que el EVENT_LEFT_CLICK, pero con el botón derecho. |

| | |
|---|---|
| ***8.*** | |
| ***9.*** *EVENT_RIGHT_DOUBLE_CLICK*<br>***10.*** | Idem que el anterior, pero haciendo doble click. |
| ***11.*** *EVENT_KEYPRESS*<br>***12.*** | Generamos el evento cuando presionamos una tecla. |
| ***13.*** *EVENT_GOT_FOCUS*<br>***14.*** | Generamos un evento así cuando nos situamos encima del control y lo hacemos potencialmente seleccionable. Esto se consigue, por ejemplo, pulsando el TAB hasta situarnos encima del control. |
| ***15.*** *EVENT_LOST_FOCUS*<br>***16.*** | Este evento se genera cuando perdemos el estatus anterior de focus. |

### 1.3.8 Timer controls

El tema de los controles de tiempo es muy importante a la hora de desarrollar aplicaciones de tipo dinámico.

La forma de incluir un *timer* es nuestro panel se consigue desde la ventana del editor de interfaz gráfica (en cualquier archivo .uir). Se incorporará al panel como el resto de controles.

Estos controles sirven para ejecutar una serie de sentencias de forma repetitiva cada cierto intervalo de tiempo, que tendremos que definir en su ventana de características correspondientes.

Cada vez que pasa el intervalo de tiempo especificado, se genera u nuevo evento : EVENT_TIMER_TICK. La forma de razonar la secuencia de ejecución es igual que para cualquier callback.

Existen dos funciones estrechamente relacionadas con los timer, y que sirven para controlarlos : *SuspendTimerCallbacks()* y *ResumeTimerCallbacks()*. La primera de ellas mantiene en estado de pausa permamente la ejecución de los timer y la segunda las activa de nuevo.

Las sentencias relacionadas con los timer se empiezan a ejecutar inmediatamente se comienza la aplicación, y tendremos que tener cuidado si no queremos que se empiezen ejecutando.

### 1.3.9 Ejecutar programas desde CVI

Existen dos funciones que permiten ejecutar programas desde nuestra aplicación en CVI :

*LaunchExecutable()* : Esta función se encuentra en la librería **Utility**. Con esta función ejecutamos un programa y regresamos al original sin esperar si la ejecución se realizó con éxito o no. En este caso podemos ejecutar archivos de extensión : *.exe, *.com, *.bat y *.pif.

System() : Esta función se encuentra en la librería **ANSI C**. Con esta función ejecutamos un programa, pero no volvemos al programa inicial sino que nons mantenemos en la aplicación. En este caso podemos ejecutar archivos de extensión : *.exe, *.com, *.bat y *.pif.

### 1.3.10 Librerías

Las librerías es un elemento fundamental de CVI, puesto que en ellas están definidas funciones que nos serán de mucha utilidad y que no tendremos que generar. Esto significa que cada vez que queramos insertar una función de una librería, nos iremos al menú **Library** y a la librería correspondiente, seleccionaremos la función que deseemos y después la insertaremos en nuestro código.

Por supuesto, tendremos que insertar cada librería que usemos en la cabecera de nuestro programa en C. El compilador no nos dejará ejecutar la aplicación sino hemos incluido todas las librerías que estemos usando.

No es el objetivo de esta documentación el detallar aquí todas las funciones de todas las librerías. Aquí simplemente daremos una descripción de la librería y nombraremos algunas funciones que pueden ser de utilidad.

#### 1.3.10.1 User Interface

En esta librería se almacenan funciones que ayudan en la programación de menús, paneles y controles contenidos en un archivo .uir. Esto significa que tendremos que acudir a esta librería siempre que queramos obtener un dato del panel, sacar un dato por pantalla, desplegar menús y barras de herramientas, controlar strip charts, ...

#### 1.3.10.2 Analysis

Esta librería contiene toda una serie de funciones establecidas para el tratamiento matemático de grandes series de números. En esta librería encontraremos funciones matemáticas, estadísticas, de filtrado de datos, de generación de señal, entre otras.

#### 1.3.10.3 Easy I/O for DAQ

**17.** En esta librería encontramos funciones que ayudan a controlar tarjetas de adquisición de datos específicas DAQ de National Instruments y hardware SCXI.

#### 1.3.10.4 Data acquisition

Aquí encontramos funciones prediseñadas con el mismo propósito que en la librería anterior, pero en un peldaño superior.

#### 1.3.10.5 Visa

La librería VISA (Virtual Instrument Software Architecture) ofrece toda una serie de funciones para el control de sistemas VXI, GPIB y otros instrumentos. Las funciones presentadas en esta librería son totalmente compatibles con las rutinas de entrada y salida usadas por CVI.

#### 1.3.10.6 GPIB y GPIB 488.2

Esta librería contiene funciones válidas para controlar sistemas GPIB ( General Purpose Interface Bus ), tanto a nivel de dispositivos como a nivel de puerto.

#### 1.3.10.7 VXI

Librería que contiene funciones para controlar sistemas VXI.

#### 1.3.10.8 RS – 232

Contiene un conjunto de funciones para controlar múltiples puertos RS–232. Todas las funciones de esta librería pueden ser usadas con el tarjeta de puerto serie RS-485 de Nartional Instrument.

Existen una serie de funciones que te permiten configurar, abrir, leer, escribir y cerrar el puerto serie de tu ordenador.

#### 1.3.10.9 TCP (Transfer Control Protocol)

Aquí se almacenan toda una serie de funciones relativas al intercambio de información entre computadores.

#### 1.3.10.10 DDE (Dinamic Data Exchange)

Librería que contiene toda una serie de funciones para proporcionar un intercambio dinámico de datos.

#### 1.3.10.11 Formatting y I/O

Librería que contiene funciones para el tratamiento general de ficheros, así para el control de los dispositivos de entrada y salida.

#### 1.3.10.12 Utility

Librería que contiene funciones de carácter general. Conteniendo funciones que proporcionan el día, hora, tiempo de espera, utilidades generales de archivos, acceso a módulos externos, y similares.

#### 1.3.10.13 ANSI C

En esta librería están contenidas las librerías más importantes relativas al ANSI : math.h, stdio.h, formatio.h, ...

### 1.3.11 Ejemplo de proyecto

Ejemplo : Proyecto senruido.prj.

El objetivo de este proyecto es el de construir un panel en el que generemos una curva senoidal afectada por una señal de ruido. El panel es muy simple (Figura 10), y contiene un gráfico, un indicador del número de ciclos de la función senoidal y un selector de la amplitud del ruido. Como aderezo se le ha proporcionado la posibilidad de cambiar el color del trazo.

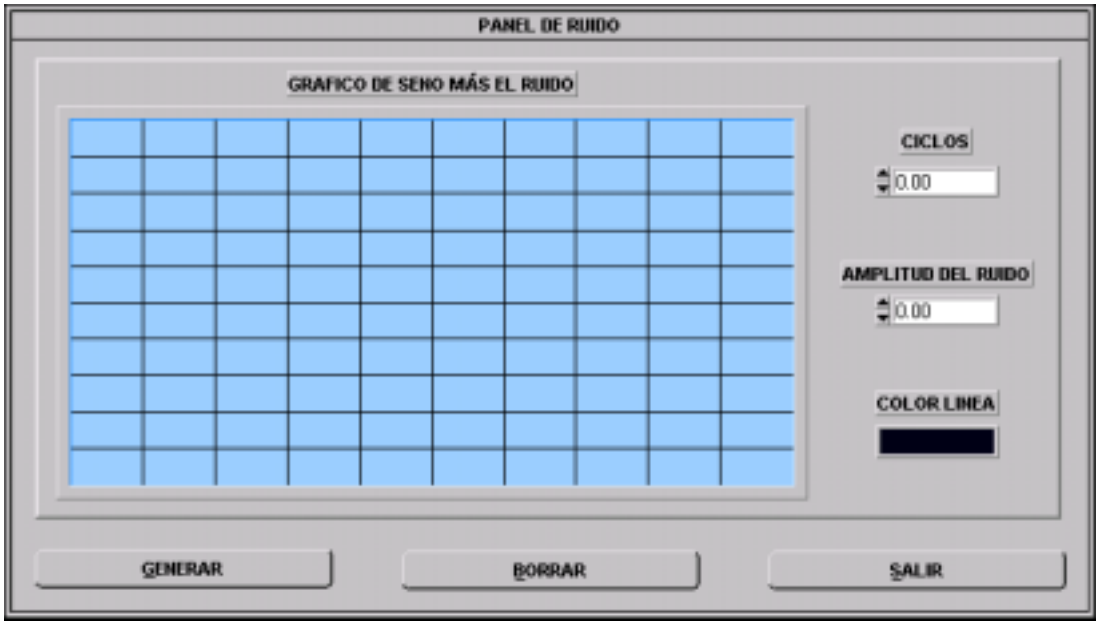


**Figura 10.- Panel de la aplicación**

Los ID y callbacks de los elementos que aparecen en el panel son :

| Elemento | ID | Callback |
|---|---|---|
| Panel | PNLSEN | |
| Generar | GENERAR | generar |
| Borrar | BORRAR | borrar |
| Salir | SALIR | salir |
| Número de ciclos | NCICLOS | |
| Amplitud del ruido | AMPRUIDO | |
| Color de línea | COLOR | |
| Gráfico | GRAFSEN | |

El código C relativo a este panel se especifica a continuación :

```
#include <analysis.h>
#include <cvirte.h>
#include <userint.h>
#include "senruido.h"

static int pnlsen;
static double seno[1000];
static double ruido[1000];
static double suma[1000];

int main (int argc, char *argv[])
{
	if (InitCVIRTE (0, argv, 0) == 0)
		return -1;
	if ((pnlsen = LoadPanel (0, "senruido.uir", PNLSEN)) < 0)
		return -1;
	DisplayPanel (pnlsen);
	RunUserInterface ();
	return 0;
}

int CVICALLBACK generar (int panel, int control, int event,
		void *callbackData, int eventData1, int eventData2)
{
	double vnciclos;
	double vampruido;
	int vcolor;
	int i;

	switch (event) {
		case EVENT_COMMIT:

			GetCtrlVal (PNLSEN, PNLSEN_NCICLOS, &vnciclos);
			GetCtrlVal (PNLSEN, PNLSEN_AMPRUIDO, &vamplitud);
			GetCtrlVal (PNLSEN, PNLSEN_COLOR, &vcolor);

			SinePattern (1000, 5.0, 0.0, vnciclos, seno);
			WhiteNoise (1000, vampruido, 1, ruido);

			for (i=0; i<999; i++)
			suma[i]=seno[i]+ruido[i];

PlotY (PNLSEN, PNLSEN_GRAFSEN, suma, 1000, VAL_DOUBLE,VAL_THIN_LINE,
VAL_EMPTY_SQUARE, VAL_SOLID, 1, vcolor);

break;
	}
	return 0;
}
```

```
int CVICALLBACK borrar (int panel, int control, int event,
		void *callbackData, int eventData1, int eventData2)
{
	switch (event) {
		case EVENT_COMMIT:

DeleteGraphPlot (PNLSEN, PNLSEN_GRAFSEN, -1, VAL_IMMEDIATE_DRAW);

			break;
	}
	return 0;
}

int CVICALLBACK salir (int panel, int control, int event,
		void *callbackData, int eventData1, int eventData2)
{
	switch (event) {
		case EVENT_COMMIT:
			QuitUserInterface(0);
			break;
	}
	return 0;
}
```