

LECCIÓN 11 PROGRAMACIÓN EN C

Prof. Ing. Miguel Angel Aguilar Ulloa

© 2009-2010

TEC

Usted es libre de:

- ✓ Copiar, distribuir y comunicar públicamente la obra.
- ✓ Hacer obras derivadas.

Bajo las siguientes condiciones:

- ✓ Reconocimiento — Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador (pero no de una manera que sugiera que tiene su apoyo o apoyan el uso que hace de su obra).
- ✓ No comercial — No puede utilizar esta obra para fines comerciales.
- ✓ Compartir bajo la misma licencia — Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

Texto de la licencia: <http://creativecommons.org/licenses/by-nc-sa/3.0/legalcode>.



1. Historia de C.
2. Estructura de un programa en C.
3. Tipos de datos simples.
4. Tipos de datos complejos.
5. Modificadores de clase de almacenamiento.
6. Modificadores de tipos de datos.
7. Operadores.
8. Estructuras de control.
9. Funciones.
10. Preprocesador de C.
11. Biblioteca estándar de C.
12. C en sistemas empotrados.

1. HISTORIA DE C

- Es un lenguaje imperativo, procedural y estructurado.
- Dispone de las estructuras típicas de los lenguajes de alto nivel pero, a su vez, dispone de construcciones del lenguaje que permiten un control a muy bajo nivel.
- Los compiladores suelen ofrecer extensiones al lenguaje que posibilitan mezclar código en ensamblador con código C o acceder directamente a memoria o dispositivos periféricos.

- El lenguaje de programación **C** fue creado por Dennis Ritchie entre 1969 y 1973 cuando trabajaba en Bell Laboratories de AT&T junto con Ken Thompson en el diseño del sistema operativo UNIX. **C** fue creado para poder escribir dicho sistema operativo en un lenguaje de alto nivel, independiente del hardware donde se ejecutara.
- **C** se basa en el lenguaje de programación B, escrito por Ken Thompson en cuya mejora colaboró Dennis Ritchie.

Evolución de C

- En 1969, Ken Thompson escribe el Lenguaje B, en Bell Laboratories, con el objetivo de recodificar UNIX (escrito hasta ese momento en lenguaje ensamblador) usando un lenguaje de alto nivel más portable y flexible.
- En 1972, Dennis Ritchie modifica el lenguaje B, creando el lenguaje **C** y reescribiendo el sistema UNIX en dicho lenguaje; añade características nuevas: diseño de tipos y estructuras de datos.
- 1983 ANSI (American National Standards Institute) estandariza **C**.
- En 1983/84, Bjarne Stroustrup (en Bell Laboratories) crea el "C con Clases", conocido como C++, el cual queda disponible en 1985.

Estandarización de C

- La primera estandarización del lenguaje C fue en ANSI, con el estándar X3.159-1989. El lenguaje que define este estándar fue conocido popularmente como ANSI C.
- Posteriormente, en 1990, fue ratificado como estándar ISO (ISO/IEC 9899:1990). La adopción de este estándar es muy amplia por lo que, si los programas creados lo siguen, el código es portátil entre plataformas y/o arquitecturas.

2. ESTRUCTURA DE UN PROGRAMA EN C

C está estructurado básicamente en tres partes:

1. ***Directivas:*** son instrucciones manejadas directamente por el preprocesador.
2. ***Declaraciones de identificadores:*** son instrucciones para que el compilador grabe el tipo y las ubicaciones de memoria asociadas a cada símbolo.
3. ***Sentencias:*** son las instrucciones ejecutables por un programa.

Preprocesador

- El preprocesador maneja directivas para la inclusión de código fuente (**#include**), definición de macros (**#define**), e inclusión condicional de código (**#if**).
- El lenguaje de las directivas del preprocesador no es estrictamente específico de la gramática de C, de tal manera que el preprocesador de C puede ser empleado independiente para procesar otro tipo de archivos.

Declaración de identificadores

- Las declaraciones definen el nombre y el tipo de identificadores empleados en un programa. Uno de los beneficios de la programación en alto nivel es la habilidad de construir grupos de instrucciones, llamadas funciones, para ejecutar tareas cuyos pasos no sean dependientes de valores específicos.
- Un identificador puede representar tanto un valor llamado variable, o bien un grupo de instrucciones, llamado función.
- Los identificadores de C representan direcciones en la memoria. En una dirección de memoria dada se puede almacenar un valor o un grupo de instrucciones de un programa.

Identificadores en memoria

- El compilador reserva memoria para todos los identificadores.
- Conforme el compilador lee un programa, graba todos los nombres de los identificadores en una tabla de símbolos. El compilador usa la tabla de símbolos internamente para llevar el control de los identificadores: su nombre, tipo y la localización en memoria que representan.
- Cuando el compilador finaliza la traducción de un programa en lenguaje máquina, reemplazará todos los identificadores empleados en el programa con instrucciones que refieran a direcciones de memoria asociadas con éstos identificadores.

Nombre de identificadores

- El nombre de un identificador puede ser cualquier palabra que empiece con una letra o un carácter underscore (_), seguido de cero o más letras, números y caracteres underscore.
- Las siguientes son las palabras reservadas de C:

auto	default	if	return	typedef
break	do	inline	short	union
case	else	int	signed	unsigned
char	enum	long	static	void
const	extern	main	struct	volatile
continue	for	pointer	switch	while

Identificadores de datos variables y constantes

- ✓ **Datos variables:** Los identificadores que representan datos de valores variables, requieren porciones de memoria que puedan ser alteradas durante la ejecución de un programa. El compilador reservará un bloque de su espacio de memoria de datos, usualmente memoria de lectura/escritura, para cada identificador de variable.

- ✓ **Datos constantes:** Los identificadores que representan valores de datos constantes, no requieren memoria alterable, una vez que el valor es escrito en memoria nunca cambia. Así el compilador reserva un bloque del espacio de memoria del programa, usualmente memoria de solo lectura.

Identificadores de funciones

- Los identificadores de funciones no son alterados durante la ejecución de un programa. Una vez que el valor de una función ha sido escrito en la memoria nunca cambia.
- Cuando una función es definida, el compilador ubica las instrucciones del programa asociadas a la función en memoria de solo lectura. Además, el compilador escribe las variables locales de la función en memoria de lectura/escritura.

Sentencias

- Cuando un programa corre ejecuta las sentencias de programa.
- Las declaraciones describen ubicaciones de memoria, las cuales son empleadas por las sentencias para almacenar y manipular valores.
- La sentencia más común en todo lenguaje de programación es la asignación. C provee muchas maneras de construir una sentencia de asignación, sin embargo, el siguiente es el caso más simple:

```
a = 20;
```

Terminador de una sentencia: El punto y coma

- Todas las sentencias en C deben terminar con un punto y coma. C usa el punto y coma como el terminador de una sentencia.
- Olvidar el punto y coma es un error común en programación. Si se olvida el punto y coma el compilador de C no sabrá donde debería terminar la sentencia.

¿Que pasa si se olvida el punto y coma?

- Dados las siguientes dos sentencias:

```
a = 20  
b = 10;
```

- Al olvidar el punto y coma al final de la primera sentencia se fuerza al compilador a leer ambas líneas como si fueran una sola . Es decir, el compilador vería lo siguiente:

```
a = 20 b = 10;
```

- Nótese que el compilador de C no toma en cuenta los espacios en blanco entre los “tokens”, mientras lee el programa. Además de los espacios en blanco, se ignoran la tabulación y los caracteres de fin de línea.

Combinando sentencias en un bloque

- Cuando se escribe una función de C se debe incluir las sentencias de la función como parte de la definición de la misma. Las sentencias pertenecientes a la función, son indicadas encerrándolas con llaves las cuales están inmediatamente después de la cabecera de la función.
- También se pueden crear bloques de sentencias en otras partes del programa como por ejemplo con las sentencias **if** y **while**:

```
while (condición) {  
    sentencias  
}
```

3. TIPOS DE DATOS SIMPLES

- Los tipos de datos en C proveen reglas para leer y guardar información en la memoria.
- La característica primaria que distingue a un tipo de datos es su tamaño. El tamaño de un tipo de datos indica la cantidad de memoria que se debe reservar para un tipo de datos en específico.
- El tamaño de los tipos de datos varía de una arquitectura a otra, de tal manera que no es posible asumir por ejemplo que las variables tipo **int** tienen el mismo tamaño en cualquier hardware. Por éste motivo en C existe un operador unario llamado **sizeof** que le permite al programador conocer el tamaño de los tipos de datos.

Tipos de datos primitivos en C

- En el lenguaje C existen tres tipos de datos básicos que son el **int**, **float** y **char**.
- A partir de estos tipos se crean otros que funcionan igual pero que difieren en el tamaño como el **long**, el **double**, y otros.

Tipos de datos de variables

- Cuando se declara un identificador de una variable que será empleada en un programa, se debe especificar el tipo de dato como parte de la declaración. Así, el compilador reservará la cantidad apropiada de memoria para cada variable.
- Es posible declarar varias variables del mismo tipo en la misma declaración, mediante una lista separada por comas. También es posible inicializar las variables a valores conocidos.

```
int a,b,c;  
char d;  
int e=0;
```

Ámbito de las variables

- No todas las partes de un programa pueden reconocer variables declaradas. La visibilidad de una variable dentro de un programa se conoce como **ámbito** de la variable.
- Una variable que tiene significado dentro de su **ámbito**, sin embargo fuera de él la variable es un símbolo desconocido o indefinido.
- En C existen dos **ámbitos**: **global** y **local**.

Ámbito global de una variable

- Si una variable es declarada fuera de todos los bloques de sentencias o funciones, el ámbito de dicha variable abarca desde el punto donde está declarada al final del mismo código fuente donde están declaradas. A éste tipo de variables se les conoce como globales.
- Si una variable quiere ser empleada antes de ser declarada en el mismo archivo de código fuente o bien desde otro archivo de código fuente, debe ser declarada como un símbolo externo, es decir, emplear el modificador de almacenamiento llamado **extern**.

```
extern int a;
```

Ámbito local de una variable

- Una variable declarada dentro de un bloque de sentencias tiene un ámbito que abarca desde la declaración de la misma hasta el final del bloque de sentencias. A éste tipo de variables se les conoce como locales.
- El nombre de la variable y su valor será definido para una función en específico y otras funciones no podrán referir a dichas variables.

```
void main () {  
    int a=0,b;  
    for (;a<10;a++) {  
        b = a;  
    }  
}
```

Tipos de datos de funciones

- El tipo de datos de una función reserva memoria para el tipo de datos del valor que retorna la función.
- El tipo de datos de una función indica la cantidad de memoria que debe ser reservada por el compilador para el valor retornado por la función.
- Existe también la posibilidad de que las funciones no retornen un valor, así no se reservará memoria para el valor de retorno de dicha función. Para definir una función de éste tipo se utiliza la palabra **void**.

```
void imprimir ();
```

Tipos de datos de los parámetros de una función

El tipo de datos de los parámetros indican el tamaño de la memoria reservada para los valores de los parámetros de una función.

```
void esperar (int tiempo);
```

La palabra **void** puede ser empleada para indicar que la función no espera recibir parámetros cuando es llamada, pero su uso en éstos casos no es obligatorio.

```
void main(void);
```

Tipo de datos enteros

- Los enteros son el tipo de dato más primitivo en C, y pueden ser almacenados como **int**, **short** o **long**.
- Para el lenguaje C existen diferentes tamaños de números enteros que pueden tener desde 1 byte hasta 8 bytes.
- El lenguaje C hace la distinción de si el entero es con signo o sin signo (**signed** o **unsigned**). En caso de que no se declare si es con signo o sin signo, se toma con signo.
- El tipo **short** usualmente almacena los datos en un espacio menor o igual al tipo **int**, mientras que el tipo **long** almacena los datos en un espacio que usualmente es el doble del **int**.

Notaciones de números enteros

- Los tipos de datos enteros usualmente manejan valores expresados en una notación decimal, sin embargo, es posible expresar un valor entero en otras notaciones.

```
int decimalInt = 32;  
  
// Los valores octales empiezan con 0  
int octalInt = 040;  
  
// Los valores hexadecimales empiezan con 0x  
int hexadecimalInt = 0x20;  
  
// Los valores binario empiezan con 0b  
int binarioInt = 0b00100000;
```

Tipos de datos flotantes

- Los valores flotantes son más modernos y se usan mucho en aplicaciones que trabajan con gráficos o que necesitan de mucha precisión.
- El tipo de dato flotante en lenguaje C sólo tiene dos tamaños: el **float** y el **double**, que son 4 bytes y 8 bytes respectivamente. A diferencia de los enteros el tipo de dato flotante soporta números decimales y números con exponente.

```
float a;  
double a = 1e23;  
double a = 3.1416;  
float a = 4e-9;  
double a = -78;
```

Tipo de datos de caracteres

- En C el tipo de datos de caracteres, **char**, almacena valores de caracteres en 1 byte de memoria. El tipo de datos **char** representa los caracteres empleando el código ASCII.
- Cuando se asigna un carácter a un identificador se debe encerrar dicho carácter entre comillas simples, las cuales le dicen al compilador que se le está asignando un carácter y no el nombre de otro identificador.

```
char letra;  
letra = 'a';  
letra = a;
```

Secuencias de escape

Las secuencias de escape son útiles para escribir un carácter literal que el compilador de C podría interpretar de otra manera.

\ \c	Backslash literal
\ "	Comillas dobles
\ '	Comillas simples
\n	Nueva línea
\r	Retorno de carro
\b	Backspace
\t	Tabulación horizontal
\a	Alerta(campana)
\v	Tabulación vertical
\?	Signo de pregunta

Cadenas de caracteres (Strings)

- Es un tipo particular de arreglos, que surge de la necesidad de contar con cadenas de caracteres. Particular porque el tipo de dato conocido como **string** o "cadena" no existe en rigor en C, para lo cual se hace uso de un conjunto de caracteres (vale decir, tipo **char**) determinando su fin con el carácter nulo: **\0**.

```
char nombre []="Nombre";  
  
/* O también */  
char nombre []={'N','o','m','b','r','e','\0'};
```

4. TIPOS DE DATOS COMPLEJOS

- Los tipos de datos complejos incluyen punteros, arreglos, enumeraciones, uniones y estructuras.
- Un sólido entendimiento de los punteros y los arreglos es absolutamente vital para un uso efecto de C.

Punteros

- Los tipos de datos primitivos de C, **char**, **int** y **float**, almacenan valores que son empleados directamente. A diferencia de los tipos primitivos, el tipo de datos puntero representa valores que son usados indirectamente.
- Un puntero al igual que los tipos de datos básicos contienen un valor numérico, la diferencia radica en la forma como es interpretado el valor, es decir, como una dirección en memoria.
- En otras palabras un puntero es una variable que contiene la dirección, o ubicación en memoria, de algún valor.

Declaración de un puntero

- La declaración de un tipo de datos puntero debe especificar el tipo de datos al cual se apuntará. Por ejemplo, la siguiente sentencia declara un puntero que apunta a un tipo de datos entero:

```
int * miPtr;
```

- Cuando se declara un puntero, el compilador le asigna un valor **NULL**, lo cual especifica que apunta a una dirección no válida. Sin embargo, puede ser asignado a una dirección conocida.
- El tipo de datos del puntero es muy importante, dado que se emplea para determinar el tamaño del bloque reservado al cual se apuntará.

Operadores con punteros

- ✓ **Operador de referencia (&)** : Es un operador prefijo unario, el cual devuelve la dirección donde está apuntando el puntero. Típicamente, se emplea para asignar la dirección de una variable a un puntero o para pasar la dirección de una variable a una función.

- ✓ **Operador de desreferencia (*)**: Es un operador prefijo unario, que devuelve el valor almacenado en la dirección de memoria en la cual apunta el puntero.

Ejemplo de uso punteros

```
int * miIntPtr; // es un puntero a un entero
int ** miPtrPtr; //es un puntero a un puntero
int miInt;

// Asigna la dirección de miInt a miIntPtr
miIntPtr = &miInt;
// Asigna la dirección de miIntPtr a miPtrPtr
miPtrPtr = &miIntPtr;
// Desrefencia el puntero y le asigna un valor
*miIntPtr = 2
```

Espacio de memoria	Dirección	Tipo	Nombre	Valor
00000010	0x00	int	miInt	2
00000000	0x01	Ptr a int	miIntPtr	0x00
00000001	0x02	Ptr a Ptr	miPtrPtr	0x01

Errores con punteros

- Desreferenciar punteros definidos a **NULL** genera un error, debido que los punteros que apuntan a **NULL** no apuntan a una dirección válida de memoria y no pueden ser desreferenciados. El siguiente código ilustra el caso:

```
int * myIntPtr;
int myInt;

*myIntPtr = myInt // desreferenciando un puntero nulo!

myIntPtr = &myInt; // forma correcta
```

Aritmética de punteros

- Sean los punteros **ptr1** y **ptr2** son del mismo tipo y **n** es entero, las siguientes son las operaciones permitidas:

Operación	Resultado	Comentario
<code>ptr1++</code>	puntero	Desplazamiento ascendente de 1 elemento
<code>ptr1--</code>	puntero	Desplazamiento descendente de 1 elemento
<code>ptr1 + n</code>	puntero	Desplazamiento ascendente n elementos
<code>ptr1 - n</code>	puntero	Desplazamiento descendente n elementos
<code>ptr1 - ptr2</code>	entero	Distancia entre elementos
<code>ptr1 == NULL</code>	booleano	Siempre se puede comprobar la igualdad o desigualdad con <code>NULL</code>
<code>ptr1 != NULL</code>	booleano	Se verifica que no sea <code>NULL</code>
<code>ptr1 <R> pt2</code>	booleano	<code><R></code> es una expresión relacional <code>==, !=, <, >, <=, >=</code>
<code>ptr1 = ptr2</code>	puntero	Asignación
<code>ptr1 = void</code>	puntero genérico	Asignación

Conversión de punteros

- Un puntero de un tipo (tipoA) puede ser convertido a otro (tipoB) usando el mecanismo de conversión:

```
char *ptrChar;           // puntero a char
int *ptrInt;             // puntero a int
ptrChar = (char *)ptrInt; // conversión
```

Punteros a funciones

- Son punteros que cuando son desreferenciados invocan a una función, pasándole uno o más argumentos como una función normal.
- Un uso muy común que se les da a los punteros de funciones es definir funciones llamadas “listener” o “callback”, las cuales son invocadas cuando un determinado evento sucede.

```
int(*ptrFuncion) (int) ;
```

Sintaxis de punteros de funciones

- **ptrFuncion** es un puntero a función que recibe un **int** como parámetro y devuelve **void**:

```
int(*ptrFuncion) (int);
```

- **ptrFuncion** un puntero a función que recibe un **int** y devuelve un puntero a un **int**:

```
int * (*ptrFuncion) (int);
```

Ejemplo de punteros de funciones

```
int max (int a, int b) {  
    if (a > b)  
        return a;  
    else  
        return b;  
}  
  
int main () {  
    int num;  
    int (*ptrFuncion) (int,int);  
    ptrFuncion = &max  
  
    num = ptrFuncion(5,3); //Como una función normal  
  
    num =(*ptrFuncion) (5,3); // En notación de punteros  
    return 0;  
}
```

Arreglos

- Los arreglos permiten agrupar elementos consecutivos relacionados de datos en bloques o estructuras.
- Cuando se declara un arreglo, se debe declarar el tipo del arreglo y el número de elementos que contiene.
- En el siguiente ejemplo se declara un arreglo que contiene 8 elementos de tipo **int**:

```
int miIntArreglo[8];
```

- Es posible definir arreglos multidimensionales:

```
int miIntArreglo[8][8];
```

Accediendo a los elementos de un arreglo

- El operador postfijo **[]**, es empleado para referirse a un elemento específico del arreglo. Así para acceder al iésimo elemento del arreglo, la sintaxis sería **miArreglo[i]**, lo cual refiere a un valor almacenado en ese elemento del arreglo.
- Debido a la interrelación entre arreglos y punteros, las direcciones de cada elemento del arreglo puede ser expresado en aritmética de punteros.

Índice del elemento	0	1	2	n
Arreglo	miIntArreglo[0]	miIntArreglo[1]	miIntArreglo[2]	miIntArreglo[n]
Puntero	*miIntArreglo	* (miIntArreglo+1)	* (miIntArreglo+2)	* (miIntArreglo+n)

Arreglos de punteros

- Un arreglo puede contener punteros a otros tipos de datos. Como en el siguiente ejemplo:

```
void func1(char *p) {  
    p[0] = "Presione 1 para empezar";  
    p[1] = "Presione 2 para reiniciar";  
    p[2] = "Presione 3 para salir";  
}  
  
void main(void) {  
    int val;  
    char *mensaje[10];  
    if (val == TRUE) {  
        func1(mensaje);  
    }  
    else  
        mensaje[0] = "El estado es correcto";  
}
```

Hardware mapeado en memoria

- Algunas arquitecturas de hardware como los microcontroladores y los SoC's tienen dispositivos mapeados en la memoria, en donde generalmente se encuentran los registros de configuración y operación del dispositivo.
- Simplemente inicializando un puntero a la dirección del registro que deseamos acceder se puede leer y escribir datos en él.

```
Int * hw_dir = (int *)0x7FFF;
```

Tipos de datos definidos por el usuario

- El tipo de datos más flexible es el definido por el programador. C permite construir nuevos tipos de datos en términos de otros ya antes definidos. Esto se logra mediante la palabra **typedef**.
- Por ejemplo, se podría crear dos nuevos tipos de datos llamados **UBYTE** y **UWORD** de la siguiente manera:

```
typedef unsigned int UBYTE;  
typedef unsigned long UWORD;  
  
UBYTE Var1;  
UWORD Var2;
```

Tipo enumerado

- El tipo enumerado en C, es especificado con la palabra **enum**.
- Éste es un tipo designado para representar valores a lo largo de una lista numerada. Cada uno de los elementos enumerados es de tipo **int**. Por defecto al primer elemento de la lista se le asigna un valor de 0.

```
enum colors { ROJO, VERDE, AZUL} colores;
```

Especificando los valores para los elementos numerados

1. Especificando los valores para cada elemento numerado:

```
enum colors { ROJO=3, VERDE=7, AZUL=21} colores;
```

2. Especificando el valor inicial para los elementos numerados:

```
enum colors { ROJO=3, VERDE, AZUL} colores;
```

Estructuras

- Las estructuras en C están definidas como contenedores de datos, que consisten de una secuencia de miembros de varios tipos de datos.
- Las estructuras están definidas por la palabra **struct**.
- Los miembros de una estructura están almacenados en ubicaciones de memoria consecutivas.

```
struct s {  
    int x;  
    float y;  
    char *z;  
} t;
```

```
struct s {  
    int x;  
    float y;  
    char *z;  
};  
  
struct s t;
```

Accediendo a los miembros de una estructura

1. Con punto:

```
t.x = 3;
```

2. Indicando el campo con un puntero de estructura:

```
struct s *tPtr = &t;  
(*tPtr).x = 3;  
tPtr->x = 3;
```

3. Inicialización:

```
struct s pi = { 3, 3.1415, "Pi" };  
struct s pi = { .x = 3, .y = 3.1415, .z = "Pi" };
```

Campos de bit

- C también provee un tipo especial de miembros de una estructura llamados campo de bit, que es un entero que explícitamente especifica el número de bits del miembro de una estructura.
- Así un elemento que ocupa un solo bit puede tener un valor entero de 0 o 1, mientras que un elemento que ocupa dos bits puede tener un valor entre 0 y 3.

```
struct f {  
    unsigned int bandera: 1; /* on (1) o off (0) */  
    signed int num : 4; /* rango -7...7 o -8...7 */  
} g;
```

Uniones

- Las uniones en C están relacionadas con las estructuras y son definidos como objetos que pueden manejar miembros de diferentes tipos y tamaños. Se definen mediante la palabra **union**.
- A diferencia de las estructuras, los componentes de una unión refieren a la misma ubicación de memoria. En este sentido una unión puede ser usada varias veces para manejar diferentes tipos de miembros, sin la necesidad de reservar memoria separada para cada miembro de la unión. El tamaño de la unión es igual al tamaño de su miembro más grande.

Ejemplo de unión

```
// Se define la unión
union compartir {
    int como_Int;
    char como_Char;
};

union compartir instancia; // Se declara la instancia
union compartir * instanciaPtr; // Se declara el puntero

Instancia.como_Int = 3; // Se asigna con punto

instanciaPrt = &instancia;
instanciaPtr->como_int = 3; // Se asigna con puntero
```

5. MODIFICADORES DE CLASE DE ALMACENAMIENTO

- Los modificadores de clase de almacenamiento controlan el proceso de reservar memoria para los identificadores declarados.
- C soporta los siguientes modificadores de clase de almacenamiento: **extern, static, register, auto** e **inline**.
- Cuando el compilador lee un programa debe decidir como reservar el almacenamiento para cada identificador. A ésta tarea se le conoce como enlazado.
- C soporta tres tipos de enlazado: externo, interno y sin enlace.

Enlazado externo

- Todas las referencias a un identificador con enlazado externo a través de un *programa* (que incluye todos los archivos objeto o de código fuente) llaman al mismo objeto en memoria.
- Tiene que haber una sola definición para un identificador con enlazado externo, o el compilador dará un error para definiciones de símbolos duplicados.
- Por defecto cada función y variable global en un programa tienen enlazado externo.

Enlazado interno

- Todas las referencias a un identificador con enlazado interno a través de una *unidad de compilación* (archivo objeto o código fuente) refieren al mismo objeto en memoria.
- Esto quiere decir que se puede definir una sola definición para cada identificador con enlazado interno en cada unidad de compilación del programa.
- Cualquier identificador que incluya **static** en su definición tiene enlazado interno.
- Ningún identificador en C tiene enlazado interno por defecto. El enlazado interno es el menos frecuente.

Sin enlazado

- Todas las referencias a un identificador sin enlazado externo a través de un *bloque de sentencias* llaman al mismo objeto en memoria.
- Una variable definida en un bloque de sentencias (función o estructuras de control **while**, **if**, etc) no tiene enlazado por defecto a menos que incluya **static** o **extern** en su definición.
- Tanto los valores de retorno de una función como sus parámetros no poseen enlazado, lo cual permite llamadas recursivas a funciones. Cada copia de una llamada recursiva a una función puede reservar copias de parámetros y valores de retorno.

Modificador **extern**

- Un identificador con enlazado externo puede ser usado en cualquier punto de un programa. Así, si se define una función en un archivo de código fuente, puede ser empleada en cualquier otra unidad de compilación.
- El concepto es similar al prototipo de funciones. Para declarar una función como externa se debe usar la palabra **extern**:

```
extern void imprimir();
```

- La palabra **extern** le dice al compilador que la definición de la función está en otra unidad de compilación. El compilador le deja el trabajo de conexión de dicho código al *enlazador*, quién resuelve las referencias de símbolos entre unidades de compilación.

Funciones externas dentro de bloques de sentencias

- C permite declarar una función externa dentro de un bloque de sentencias, lo cual informa al compilador que la función está definida en otro lugar del programa y su ámbito se restringe al bloque de sentencias.
- Por ejemplo, supóngase que la función **inicializar()** se encuentra definida en otro archivo de código fuente y se quiere hacer visible solo a **main()** y no al resto de funciones en el mismo código fuente que **main()**.

```
void main(void) {  
    extern int inicializar(void);  
    inicializar();  
}
```

Variables globales y **extern**

- Al igual que las funciones, las variables globales tienen enlazado externo. Para emplear una variable global en más de un archivo de código fuente, debe ser declarada como **extern**:

```
extern int variableGlobal;
```

- El compilador interpreta lo anterior como una declaración externa no como la declaración de una variable. Así, la declaración real de la variable debe estar en otro archivo de código fuente.

```
int variableGlobal;
```

Ejemplo de variables externas

✓ Archivo 1:

```
int variableGlobal; // definición
void algunaFuncion(void); // declaración externa implícita

int main() {
    GlobalVariable = 1;
    algunaFuncion();
    return 0;
}
```

✓ Archivo 2:

```
extern int variableGlobal; // declaración externa

void algunaFuncion(void) {
    ++variableGlobal;
}
```

Modificador **static**: Ámbito global

- Por defecto, todas las funciones y variables declaradas en el ámbito global tienen enlazado externo y son visibles para el programa entero. Sin embargo, a veces se requiere que las variables o funciones tengan enlazado interno, es decir, que sólo sean visibles para una unidad de compilación. Para lograr esto se emplea la palabra **static**:

```
static int variableEstatica;  
static int funcionEstatica(void);
```

- Éstas declaraciones crean identificadores globales que no son accesibles por ninguna otra unidad de compilación.

Modificador **static**: Ámbito local

- La característica de las variables automáticas de perder su valor al salir del bloque en que han sido definidas, es un serio inconveniente en algunas ocasiones. Para resolver el problema se inventaron las variables estáticas locales, que un tipo especial de variable que no se destruye cuando la ejecución sale de su ámbito y conserva su valor.
- Una variable estática local actúa como una variable local en cuanto a visibilidad, pero como una externa en cuanto a duración. Es decir, solo son conocidas en la función o bloque de sentencias en el que se declaran, pero conservan su último valor entre llamadas sucesivas a la función.

Ejemplo de Uso de static: Ámbito local

- El siguiente es un ejemplo de un código recursivo en el cual la variable contador conserva su valor llamada tras llamada.

```
int funcionRecursiva(void) {  
    static int contador=1;  
    contador += 1;  
  
    if (contador > 10) {  
        funcionRecursiva();  
    }  
}
```

Modificador **register**

- Cuando se declara una variable con el modificador **register**, se le está informando al compilador que optimice el acceso a la variable empleando registros del procesador para almacenar dicha variable y no ubicaciones de memoria RAM.
- A diferencia de otros modificadores de clase de almacenamiento, **register** es simplemente una recomendación para el compilador.

```
void contador () {  
    register int contador=1;  
    while (contador<10) {  
        ...  
        contador += 1;  
    }  
}
```

Modificador auto

- La palabra **auto** denota una variable temporal. Se puede usar solo con variables debido a que C no soporta funciones de ámbito local.
- Debido a que todas las variables declaradas dentro de un bloque de sentencias no tiene enlazado por defecto, la única razón para usar **auto** es para aclarar.

```
void contador () {  
    auto int contador=1;  
}
```

Modificador **inline**

- La instrucción **inline** le sugiere al compilador que una función particular va a ser sujeta de una expansión en línea, es decir, se le sugiere al compilador insertar el cuerpo completo de una función en cada punto del programa donde dicha función es llamada.
- Una expansión en línea es típicamente usada para eliminar el tiempo inherente que ocurre cuando se llama una función. Es usado para funciones que son llamadas frecuentemente, por lo cual el retardo de las llamadas es más significativo.
- El uso de expansiones en línea de funciones presenta ventajas sobre la expansión de macros.

Ejemplo de uso de inline

```
inline int max (int a, int b) {  
    if (a > b)  
        return a;  
    else  
        return b;  
}  
  
int main () {  
    int num;  
  
    // Es equivalente a "a = (x > y ? x : y);"  
    num = max(5,3);  
    return 0;  
}
```

6. MODIFICADORES DE TIPOS DE DATOS

- Los modificadores de tipos de datos modifican la forma en que la información es almacenada y leída de memoria. Los modificadores de tipo extienden los tipos de datos básicos.
- Los modificadores de tipo de datos se aplican solo a variables y no a funciones. Se pueden emplear con variables, parámetros y valores retornados por funciones.

Modificadores de constancia de valores

- La habilidad de un compilador para optimizar un programa se basa en varios factores. Uno de esos factores es la constancia de los datos de un programa.
- Por defecto, las variables utilizadas en un programa cambian de valor cuando una instrucción para hacer tal cosa es dada por el desarrollador en el mismo programa.
- Los tipos de modificadores de constancia son **const** y **volatile**.

Modificador **const**

- A veces se requiere crear variables cuyos valores sean inalterables durante la ejecución de un programa. Para definir variables de éste tipo en C se emplea la palabra **const**. También es posible definir los parámetros de funciones como constantes.

```
const float PI = 3.1415926;
```

- Cuando el programa es compilado la variable PI y todas las variables constantes de un programa se ubican en memoria de solo lectura.
- Si se quisiera alterar el valor de dicha variable mediante una asignación el compilador producirá un error en tiempo de compilación.

Modificador **volatile**

- Las variables volátiles son variables cuyos valores pueden cambiar sin una instrucción directa que haga esto. Por ejemplo, una variable que contiene datos que son recibidos desde un puerto pueden cambiar al mismo tiempo que los datos del puerto, razón por la cual las variables volátiles son de gran importancia en sistemas empotrados.
- Empleando la palabra **volatile** se le informa al compilador que no debe realizar optimizaciones sobre dichas variables debido a que el valor puede cambiar en cualquier momento.

Ejemplo del uso de volatile

- Dado el siguiente ciclo:

```
static int contador;
void main (void) {
    contador = 0;
    while (contador != 255);
}
```

- Un compilador optimizador notará que no hay código con la posibilidad de cambiar a contador, de tal manera que el compilador lo sustituirá con un ciclo infinito.

```
static int contador;
void main (void) {
    contador = 0;
    while (true);
}
```

Ejemplo del uso de volatile

- Para prevenir éste problema se emplea **volatile**:

```
volatile static int contador;  
void main (void) {  
    contador = 0;  
    while (contador != 255);  
}
```

Modificadores de Signo (`unsigned` y `signed`)

- ✓ **`signed`**: La palabra **`signed`** fuerza al compilador a usar el bit más significativo de un entero como bit de signo. Por defecto, **`short`**, **`int`** y **`long`** tienen signo. El tipo **`char`** no tiene signo:

```
signed char a;
```

- ✓ **`unsigned`**: Para crear variables de tipo **`short`**, **`int`** o **`long`** sin signo se debe emplear la palabra **`unsigned`** en la declaración.

```
unsigned int a;
```

Modificadores de tamaño (`short`, `long`)

- Los modificadores `short` y `long` le indican al compilador cuento espacio se debe reservar para una variable tipo `int`.

- ✓ `short`: La palabra `short` declara a una variable tipo `int` del mismo tamaño que una `char`, usualmente un byte. `Short` será siempre del mismo tamaño o menor que el `int`.

```
short int a;
```

- ✓ `long`: La palabra `long` declara a un `int` usualmente como el doble una variable `int` normal; `long` será siempre mayor o igual a `int`.

```
long int a;
```

Límites típicos de los tipos de datos enteros

Especificadores Implícitos	Especificador Explícito	Bits	Bytes	Valor Mínimo	Valor Máximo
signed char	Igual	8	1	-128	+127
unsigned char	Igual	8	1	0	255
char	-	8	1	-128 o 0	+127 o 255
short	signed short int	16	2	-32,768	+32,767
unsigned short	unsigned short int	16	2	0	65,535
int	signed int	16 or 32	2 or 4	-32,768 o -2,147,483,648	+32,767 o +2,147,483,647
unsigned	unsigned int	16 or 32	2 or 4	0	65,535 o 4,294,967,295
long	signed long int	32	4	-2,147,483,648	+2,147,483,647
unsigned long	unsigned long int	32	4	0	4,294,967,295
long long	signed long long int	64	8	-9,223,372,036,854,775,808	+9,223,372,036,854,775,807
unsigned long long	unsigned long long int	64	8	0	18,446,744,073,709,551,615

Modificadores de tamaño de punteros (**near**, **far**)

- **far** y **near** permiten diferentes tamaños de punteros para direccionar diferentes áreas de memoria.
- ✓ **near**: Le indica al compilador que el puntero se encuentra en el mismo segmento de memoria que el programa en el que se encuentra definido. Se emplean usualmente para datos y variables.

```
int near * a;
```

- ✓ **far**: Le indica al compilador que el puntero se encuentra en otro segmento de memoria diferente al del programa en el que se encuentra definido. Se emplean usualmente para funciones.

```
int far * a;
```

7. OPERADORES

- Las variables y funciones contienen y pasan valores entre los diferentes módulos del programa. En éste sentido, los operadores son los que permiten realizar cálculos con éstos valores.
- Cuando se escribe un programa la mayor parte de dicho programa está dedicada a realizar manipulación de datos como por ejemplo incrementar y decrementar contadores.
- C posee uno de los conjuntos de operadores más grandes en lenguajes de programación, lo cual, algunos lo consideran como una desventaja debido a que hace el código más difícil de leer y depurar.

Expresiones

- Los cálculos y manipulación de datos se conforman empleando expresiones. En C una expresión está formada de la combinación de operadores, constantes y variables.
- Las constantes e identificadores se pueden ver como bloques de construcción y los operadores como un conjunto de formas predefinidas en que se pueden combinar los bloques.
- Una expresión se convierte en una sentencia al terminarla con un punto y coma.

Ligadura (Binding)

- ¿Cómo hace un compilador para determinar cuáles expresiones aplican a cada operador en las sentencias de un programa?
- Las reglas que determinan el comportamiento especifican el número de expresiones que requiere el operador.
- Ésta relación se indica diciendo que un operador liga a un número de determinado de expresiones.

Operadores unarios

- Los operadores unarios ligan a una sola expresión.
- Los operadores que ligan a la expresión en su inmediata derecha se les conoce como operadores prefijos unarios. A los operadores unarios que ligan a la expresión en su inmediata izquierda se les conoce como operadores unarios postfijos.

```
a[6]; //operador unario postfijo  
a++; //operador unario postfijo  
++a; //operador unario prefijo  
&a; //operador unario prefijo
```

Operadores binarios

- Los operadores binarios ligan a dos expresiones.
- Los operadores ligan expresiones localizadas a su inmediata izquierda y derecha.

```
a * b; //multiplicación de dos expresiones  
a / b; //división de dos expresiones  
a - b; //substracción de una expresión a otra  
a + b; //suma de dos expresiones  
a >> b; //desplazamiento de bits hacia la derecha
```

Operador ternario

- C soporta un solo operador ternario, el cual liga tres expresiones.
- El operador condicional **? :** liga tres expresiones.

```
a ? b : c;
```

- La expresión **a** es evaluada, si es verdadera (no es cero) entonces el valor de la expresión entera es el valor de la expresión **b**. Si la expresión **a** es falsa (cero), entonces el valor de la expresión entera es el valor de la expresión **c**.

Precedencia de operadores

- Las sentencias usualmente contienen más de un operador.
- Con el objetivo de evitar ambigüedades C provee un conjunto de reglas de precedencia. Éstas reglas definen el orden en el cual los operadores ligan expresiones.

```
a * b + c - d / f;
```

- Existen dos reglas básicas de precedencia:
 1. Los operadores de multiplicación y división ligan primero que los de suma y substracción.
 2. Los paréntesis definen explícitamente el orden en que ligan expresiones los operadores.

Tabla de precedencia de operadores

- Las operaciones en un nivel más alto en la tabla tienen precedencia sobre las que están debajo de ellas. que están en el mismo nivel se ejecutan en el orden en que aparecen. El optimizador del compilador usualmente reagrupa sub-expresiones que son tanto asociativas y conmutativas con el objetivo de mejorar la eficiencia del código.

Tipo de Expresión	Operadores		
Primaria	Identificador Constante String Expresión		
Postfijo	a [b] f () a . b a -- a ++		
Prefijo	++a --a sizeof (a) &a *a ~a !a +a -a		
Conversión	(tipo) a		
Multiplicativo	a * b a / b a % b		

Tabla de precedencia de operadores

Tipo de Expresión	Operadores	
Aditivo	$a+b$	$a-b$
Desplazamiento	$a<<b$	$a>>b$
Relacional	$a < b$ $a \leq b$	$a > b$ $a \geq b$
Igualdad	$a == b$	$a != b$
AND a nivel de bits	$a \& b$	
XOR a nivel de bits	$a ^ b$	
OR a nivel de bits	$a b$	
AND lógica	$a \& \& b$	
OR lógica	$a b$	
Condicional	$a ? b : c$	
Asignación	$a = b$ $a += b$ $a *= b$ $a \&= b$ $a <<= b$	
	$a -= b$ $a /= b$ $a ^= b$ $a >>= b$	
Coma	a, b	

Operadores aritméticos

- Los operadores aritméticos ($+$, $-$, $*$, $/$, $\%$) realizan aritmética simple sobre las expresiones. Los primeros tres simplemente suman, resta y multiplican expresiones.

$a + b$

$a - b$

$a * b$

- El operador de división ($/$) retorna el cociente completo, en donde cualquier porción fraccional es truncada y perdida para número enteros. Si se está manejando números en punto flotante el operador de división llevará a cabo una división en punto flotante. Por otra parte el operador de módulo ($\%$) retorna el residuo de la división.

Operadores incrementales y decrementales

- Los operadores incrementales y decrementales son unarios y poseen una precedencia más alta que otros operadores aritméticos.
- El operador incremental (**++**) suma uno a su identificador, mientras que el operador decremental (**--**) resta uno.
- Los operadores incrementales y decrementales pueden ser empleados en dos formas: prefijo y postfijo.
- Debido a que los operados incrementales y decrementales modifican el valor del identificador al cual están asociados no es posible emplearlos en expresiones complejas como por ejemplo: **++ (a+b) ;** .

Postfijo

- Las versiones postfijo de los operadores incrementales y decrementales primero retornan el valor de la expresión y luego realizan el incremento o decremento.

```
contador=0;    //Se establece en 0
a=contador++; //a se le asigna 0
b=contador;   //b se le asigna 1

contador=10;   //Se establece en 10
a=contador--; //a se le asigna 10
b=contador;   //b se le asigna 9
```

Prefijo

- Las versiones postfijo de los operadores incrementales y decrementales primero realizan el incremento o decremento y luego retornan el valor.

```
contador=0; //Se establece en 0
a=++contador; //a se le asigna 1
b=contador; //b se le asigna 1

contador=10; //Se establece en 10
a=--contador; //a se le asigna 9
b=contador; //b se le asigna 9
```

Tabla de operadores aritméticos

Nombre del Operador	Sintaxis
Más unario	$+a$
Suma	$a + b$
Incremento Prefijo	$++a$
Incremento Postfijo	$a++$
Asignación mediante suma	$a += b$
Menos unario (negación)	$-a$
Substracción	$a - b$
Decremento Prefijo	$--a$
Decremento Postfijo	$a--$

Nombre del Operador	Sintaxis
Asignación mediante resta	$a -= b$
Multiplicación	$a * b$
Asignación mediante multiplicación	$a *= b$
División (Cociente)	a / b
Asignación mediante división	$a /= b$
Módulo (Residuo)	$a \% b$
Asignación mediante módulo	$a \%= b$

Operadores de comparación

- La comparación de valores en programación es una habilidad muy empleada en los programas.
- Preguntas como: ¿Son éstos valores iguales?, ¿Es “a” mayor a “b”? son comunes en programación.
- C provee dos conjuntos de operadores que se emplean para probar y retornar el valor verdadero de una expresión: operadores relacionales y operadores lógicos.

Expresando verdadero y falso (True y False)

- Cualquier expresión que retorne un valor de 0 es considerada como falsa mientras que una expresión que retorna otro valor es considerada como verdadera.
- C no posee el tipo de datos booleano, así que resulta útil definir las constantes simbólicas **TRUE** y **FALSE** con el objetivo de mejorar la legibilidad del código.

```
#define TRUE 1  
#define FALSE 0
```

Operadores relacionales

- ✓ El operador `==` retorna 1 si las dos expresiones ligadas son idénticas en valor.
- ✓ El operador `!=` retorna 1 si las dos expresiones ligadas son diferentes en valor.
- ✓ El operador menor que `<`, retorna 1 si la expresión de la izquierda es menor en valor a la expresión de la derecha.
- ✓ El operador mayor que `>`, retorna 1 si la expresión de la izquierda es mayor en valor a la expresión de la derecha.
- ✓ El operador menor que y mayor que tienen una versión “o igual a”. Tanto menor o igual a `<=` y mayor o igual a `>=`, también retornan 1 si ambos valores son iguales.

Tabla de operadores relacionales

Nombre del Operador	Sintaxis
Menor que	$a < b$
Menor o igual que	$a \leq b$
Mayor que	$a > b$
Mayor o igual que	$a \geq b$
No es igual que	$a \neq b$
Igual que	$a == b$

Operadores lógicos

- ✓ El operador unario lógico **NOT** (!) retorna 1 si la expresión a la cual está ligada tiene un valor de 0, de lo contrario retorna 0.
- ✓ El operador binario lógico **AND** (&&), retorna 1 si sus dos expresiones ligadas poseen valores diferentes de cero, de otra manera retorna 0.
- ✓ El operador binario lógico **OR** (||), retorna 1 si cualquiera de sus expresiones ligadas posee un valor de 1. Sólo puede retornar 0 en caso de que sus dos expresiones ligadas sean 0.

Corto circuito de expresiones lógicas

- Es una característica del lenguaje C que se utiliza para optimizar programas en algunas situaciones.
- Los cortos circuitos entran en el momento en que tenemos condiciones muy grandes y la idea es que el programa puede verificar si una expresión es verdadera o falsa sin evaluar la condición completa.
- Así, cuando un programa corre, solo se evaluarán tantas expresiones lógicas como sean necesarias para determinar si la expresión completa retorna un 1 o un cero.

Ejemplo de corto circuito

```
if ( (teclaPresionada() == TRUE) &&
    ((tecla = getch()) == 0) ) {
    printf("La tecla es %s\n", tecla);
}
```

- En éste ejemplo si la función `teclaPresionada()` retorna cero se evita la necesidad de leer el valor de la tecla cuando no se ha presionado

Operadores a nivel de bits

- C soporta un operador unario lógico a nivel de bits y tres operadores binarios. Cada uno de éstos operadores puede actuar solo sobre valores almacenados en variables con el tipo de datos **char**, **short int**, **int** y **long int**.

- ✓ **AND a nivel de bits (&)** : produce una AND lógica a nivel de bits para cada pareja de operandos.

```
int a=5, b=2, c; // a es 101 y b 010
c = a & b;       // c obtiene un valor de 000
```

- ✓ **OR a nivel de bits (|)** : produce una OR lógica a nivel de bits para cada pareja de operandos.

```
int a=5, b=2, c; // a es 101 y b 010
c = a | b;       // c obtiene un valor de 111
```

Operadores a nivel de bits

- ✓ **XOR a nivel de bits (^)** : produce una XOR lógica a nivel de bits para cada pareja de operandos.

```
int a=5, b=2, c; // a es 101 y b 010
c = a ^ b;       // c obtiene un valor de 111
```

- ✓ **NOT a nivel de bits (~)** : Produce un complemento a uno de un valor binario. Cada bit que estaba en uno en el valor original se le asignará un cero y cada bit en cero en el valor original se le asignará un uno.

```
int a=5, b; // a es 101
b = ~a;     // b obtiene un valor de 010
```

Operadores de desplazamiento de bits

- Ambos operandos de un operador de desplazamiento de bits deben ser enteros.

- ✓ **Desplazamiento hacia la derecha (>>):** El operador de desplazamiento a la derecha desplaza los datos hacia la derecha el número de posiciones indicadas. Los bits desplazados fuera del límite derecho desaparecen.
- ✓ **Desplazamiento hacia la izquierda (<<):** El operador de desplazamiento a la izquierda desplaza los datos hacia la izquierda el número de posiciones indicadas. Los bits desplazados fuera del límite izquierdo desaparecen.

Operadores de desplazamiento de bits

```
a = 0b10000000;  
  
while (a.7 != 1) {  
    a >> 1;  
}  
  
while (a.0 != 1) {  
    a << 1;  
}
```

8. ESTRUCTURAS DE CONTROL

- Uno de las características más importantes de cualquier lenguaje de programación es la habilidad de controlar la forma en que las sentencias de un programa son ejecutadas.
- Normalmente, las sentencias son ejecutadas secuencialmente. Sin embargo, algunas veces se requiere desviar el flujo de ejecución secuencial.
- Las estructuras de control permiten realizar decisiones acerca de que instrucciones ejecutar. En general las estructuras de control en C se agrupan en estructuras de control de salto y de ciclo.

Expresiones condicionales

- Una estructura de control prueba el valor de una expresión particular en tiempo de ejecución y toma una decisión de cómo proceder.
- La representación de falso y verdadero en C es un concepto que hay que tener en mente al programar. Hay que recordar que falso se representa con 0 y verdadero se representa con un valor diferente de 0, típicamente 1 y eso es lo que interpretarán las estructuras de control para tomar decisiones.

Estructuras de decisión if

- La estructura **if** especifica la ruta de ejecución basado en el resultado expresión condicional que evalúa.

```
if (expresión) {  
    // Sentencias que se ejecutan si  
    // la expresión es verdadera  
}
```

- Nótese que la sentencia **if** no es precedida por un punto y coma, debido a que la sentencia **if** no es una sentencia completa por sí misma. Ésta requiere una sentencia o un bloque de sentencias para complementarla.

```
if (a) b=c;  
if (a) {  
    b=c;  
    d=e;  
}
```

Estructuras de decisión **else**

- Existe un componente adicional al **if** en caso de que la expresión evaluada sea falsa. Esta estructura es el **else**.
- La estructura **else** debe ser antecedida por un **if**. Cuando la condición evaluada por el **if** es falsa la ejecución pasa directamente a la estructura **else**.

```
if (expresión) {  
    // Sentencias que se ejecutan si  
    // la expresión es verdadera  
} else {  
    // Sentencias que se ejecutan si  
    // la expresión es falsa  
}
```

- Al igual que el **if**, el **else** necesita una sentencia o bloque de sentencias que le provean el punto y coma.

Estructuras de decisión: Coincidencia de if-else

- Un **else** siempre coincide con el **if** más cercano.
- ¿Si a es 1 cual será el valor final de b?

```
if (a)
    b=1;
    if (!a)
        b=2;
else          El valor de b
    b=3;      ←             será 3
```

- Una buena práctica de programación es utilizar siempre los corchetes para hacer más legible el programa:

```
if (a) {
    b=1;
    if (!a) {
        b=2;
    }else{
        b=3;
    }
}
```

Estructuras de decisión: switch-case

- La estructura **if-else** permite tomar la decisión entre dos rutas de código. Sin embargo, C provee una estructura llamada **switch-case** que permite manejar múltiples rutas de código para ser ejecutado.
- El **switch-case** tiene un valor o expresión (**switch**) que determina cuál ruta de código se va a ejecutar (**case**).

```
switch (expresión) {  
    case valor1:  
        // Sentencias  
        break;  
    case valor2:  
        // Sentencias  
        break;  
}
```

Estructuras de decisión: Ejecución de un switch-case

- Cuando el **switch** es ejecutado cada case es probado uno por uno. Si el valor de un caso no coincide con el valor de la expresión evaluada por el **switch**, todo el código de dicho case es ignorado.
- El proceso continúa hasta que algún case coincida o se alcance el final del **switch**.

Estructuras de decisión: default

- El **default** es una opción que siempre es considerada como coincidente con la expresión evaluada por el **switch**, es decir, no importa que valor sea evaluado por el **switch** el **default** siempre será ejecutado.
- Esto significa que el default siempre debe ubicarlo al final de la estructura **switch-case** o de lo contrario se omitirían otras posibles opciones del **switch-case**.

```
switch (expresión) {  
    case valor1:  
        // Sentencias  
        break;  
    default:  
        // Sentencias  
}
```

Estructuras de decisión: Efecto de ejecución en cascada

- Una vez que el valor de un **case** coincide con la expresión que está evaluando el **switch** cada línea subsecuente es ejecutada inclusive otros **case's** posteriores si no se emplea la instrucción **break** al final de cada **case**.
- Usualmente esto no es un comportamiento deseado por eso se emplea el **break**, sin embargo, en algunos casos se puede sacar ventaja de éste comportamiento en donde se desea que para dos opciones del case se ejecute el mismo código.

```
switch (expresión) {  
    case valor1:  
    case valor2:  
        // Sentencias  
        break;  
}
```

Estructuras de decisión: goto

- La sentencia **goto** sirve para indicar al programa que continúe ejecutándose desde la línea de código indicada. Sin embargo, ésta es una sentencia poco aceptada.
- C posee una variedad de estructuras de control útiles, de tal manera que no se debería utilizar **goto**. Sólo en ocasiones muy excepcionales será recomendado el uso del **goto** al crear iteraciones muy complejas.

ETIQUETA:

```
//Sentencias
```

```
goto ETIQUETA;
```

Estructuras de ciclo

- Como se vio anteriormente las estructuras de control de C permiten tomar decisiones sobre la ruta de ejecución del código. C también provee estructuras de ciclo para repetir conjuntos de sentencias.
- El componente clave de cualquier estructura de ciclo es la expresión de control. En algún punto de cada iteración la expresión de control es verificada. Si la expresión de control es 1 la ejecución continúa en bloque de sentencias del ciclo, sin embargo, si la expresión de control es 0 el ciclo se termina y la ejecución del programa continúa a partir de la sentencia que se encuentra inmediatamente después del ciclo.

Estructuras de ciclo: while

- La estructura de ciclo más simple en C es el **while**.
- En el **while** la expresión de control está en el tope de la estructura. El ciclo **while** evalúa la expresión de control antes de cada iteración, incluida la primera. Así, si la expresión de control es 0 la primera vez que se ejecuta el **while**, las sentencias que se encuentran dentro de la estructura nunca se ejecutarán.

```
while (expresiónControl) {  
    // Sentencias  
}
```

Estructuras de ciclo: do-while

- El ciclo **do-while** verifica la expresión de control después de cada iteración.
- Debido a ésto el bloque de sentencias se ejecutará siempre al menos una vez, incluso si la expresión de control es 0 cuando el ciclo se ejecuta por primera vez.

```
do {  
    // Sentencias  
} while (expresiónControl);
```

Estructuras de ciclo: **for**

- La estructura de ciclo más compleja y flexible de C es el **for**. El ciclo **for** incorpora sentencias las cuales alteran las variables empleadas en la expresión de control. En el siguiente ejemplo se compara el ciclo **while** con **for**:

```
contador=0;
while (contador<=10) {
    //Sentencias
    contador++;
}
```

```
for(contador=0;contador<=10;
    contador++) {
    //Sentencias
}
```

- Un ciclo **for** ejecuta las sentencias un número predeterminado de veces. La expresión de control para el ciclo es inicializada, verificada y manipulada completamente dentro del paréntesis del **for**.

Estructuras de ciclo: Funcionamiento del **for**

- El ciclo **for** tiene tres expresiones que determinan su operación. La siguiente es la sintaxis general de un ciclo **for**:

```
for (inicialización; control; incremento) {  
    // Sentencias  
}
```

- ✓ **Inicialización:** determina el valor inicial para las variables de control. Pueden ser una o más variables de control. Es ejecutada solo al inicio del ciclo.
- ✓ **Control:** determina las condiciones para finalizar el ciclo. Es ejecutada antes de cada ciclo.
- ✓ **Incremento:** modifica el valor de las variables de control. Es ejecutada después de cada ciclo.

Terminar un ciclo: **break**

- La sentencia **break** es empleada para terminar completamente la ejecución de un ciclo. El uso más común es en el **switch-case**, sin embargo, **break** puede ser utilizado para terminar cualquier ciclo.
- Cuando el **break** es encontrado en un ciclo dentro de una estructura de ciclo, ésta termina inmediatamente y la ejecución pasa a la sentencia que se encuentra inmediatamente después del ciclo.

```
while (expresión1) {  
    // Sentencias  
    while (expresión2) {  
        break;  
    }  
}
```

Terminar un ciclo: `continue`

- Algunas veces se desea saltar a la siguiente iteración de un ciclo sin terminar el mismo completamente. La sentencia `continue` permite esto.
- Cuando una sentencia `continue` es encontrada dentro de una estructura de control, la ejecución pasa al final del bloque del ciclo. Así la siguiente acción será evaluar la expresión de control del ciclo.

```
while (expresión1) {  
    // Sentencias  
    if (expresión2)  
        continue;  
}  
}
```

9. FUNCIONES

- Las funciones son los bloques de construcción básicos para todos los programas en C.
- Cada programa en C tiene al menos una función llamada **main()**. Cuando un procesador corre un programa, la ejecución generalmente empieza con la primera sentencia de la función **main()**.
- Cabe resaltar que la función inicial no necesariamente es **main()**. Se le puede indicar al enlazador que la función de entrada (entry point) del programa es otra diferente de **main()**.

Ejecución de una función

- Cualquier función en un programa C puede ejecutar o llamar cualquier otra función. Típicamente, la función `main()` llama a uno o más funciones.
- Existe una restricción en la llamada de funciones: una función que no está reconocida no puede ser llamada. Existen dos técnicas para permitirle a una función ser reconocida:
 1. Proveer la definición completa de la función antes de la parte del programa que la llama. Éste método tiene una complicación: Es posible que dos funciones se llamen entre sí ¿Cuál de ésta funciones se debe definir primero?
 2. Emplear un prototipo de función para indicarle al compilador acerca de la función antes de que ésta esté definida.

Llamada a una función

- La sintaxis para una llamada a una función en C es el nombre de la función y la lista de parámetros encerrados entre paréntesis. Cuando el compilador de C encuentra un identificador seguido para un paréntesis izquierdo sabe que dicho identificador representa una función.
- Si se ha definido una función llamada **sum()** para sumar dos enteros y retornar el resultado se puede asignar el valor de retorna a una variable de la siguiente manera:

```
c = sum (a,b) ;
```

- También es posible incluir expresiones en los lugares donde se pasan los parámetros cuando se llama la función:

```
c = sum (a*x, b*y) ;
```

Prototipo de funciones: Interfaz de una función

- Los prototipos de funciones permiten definir completamente la interfaz de una función sin preocuparse por su contenido. La interfaz de una función contiene:
 1. El tipo de datos returnedo por la función.
 2. El nombre de la función.
 3. Los tipos de datos de los parámetros de la función.
- Una vez que el prototipo de una función define una interface, el compilador puede verificar las llamadas a la función: ¿El número y tipo de parámetros es correcto?, ¿El tipo de datos del valor de retorno es tratado apropiadamente?

```
int funcion (int,int);
```

Prototipo de funciones: Interfaz de una función

- El emplear un prototipo de una función asegura que la función sea utilizada correctamente en cualquier parte del programa.
- El prototipo de una función le provee la información completa al compilador para que éste compile apropiadamente un archivo de código fuente, sin la necesidad de conocer la definición completa de la función.
- Es trabajo del enlazador es resolver los símbolos y conectar la definición completa de la función en los lugares que dejó previsto el compilador.

Tipo de la función, nombre y lista de parámetros

- Dado el siguiente prototipo de función:

```
<tipo> <nombre>(<parámetros>) ;
```

- ✓ **Tipo de datos:** Éste le dice al compilador el tipo de datos del valor returnedo por la función. Así el compilador sabrá cuanta memoria debe reservar para el valor returnedo por la función.
- ✓ **Nombre de la función:** Éste identificador es ingresado en la tabla de símbolos y asociado con una dirección que contiene el inicio del código ejecutable de función. Cuando el programa llama una función, el flujo de ejecución salta a la dirección asociada con el nombre de la función.

Tipo de la función, nombre y lista de parámetros

✓ **Parámetros:** Éstos son los que hacen que un identificador sea interpretado como una función y no como una variable. Se deben incluir los paréntesis inclusive si la función no acepta parámetros. Si la función acepta parámetros se debe incluir al menos el tipo de datos de cada parámetros, sin embargo, aunque no es necesario el nombre de ellos se suele incluir por claridad.

```
//Solo tipos de los parámetros
int sum(int,int);

//Solo tipos de los parámetros
int sum(int numA ,int numB);
```

Llamadas a funciones en otros archivos

- Los prototipos de funciones permiten emplear funciones que se encuentran definidas en otros archivos, inclusive si los archivos han sido ya compilados.
- Los archivos pre-compilados de funciones son llamados bibliotecas objeto y en la programación de C se hace uso extensivo de ellas. Los prototipos de funciones en una biblioteca pre-compilada están contenidos en un archivo de cabecera.
- Por ejemplo, en la programación en C se hace uso de la biblioteca estándar de funciones la cual provee funciones para la entrada y salida de información del usuario. El archivo cabecera de ésta biblioteca se llama **stdio.h** el cual contiene los prototipos de éstas funciones.

Funciones y void

- Algunas funciones no aceptan o retornan valores.

```
esperar();
```

- La mayoría de los compiladores cuando encuentran tal sentencia asumen que el tipo de retorno de datos es **int**, por lo cual se gasta memoria innecesariamente. Para evitar esto se emplea la palabra **void**:

```
void esperar();
```

- **void** le indica al compilador que la función no retorna valores así no se reservará memoria. **void** también puede ser empleado para indicar que la función no recibe parámetros.

```
void esperar(void);
```

Definición de una función

- Un prototipo de una función describe la interfaz de una función mientras que la definición de una función describe la interfaz y el contenido de la misma.
- Por el contenido de la función se entiende las sentencias que ejecuta la función cuando es llamada.
- Cuando el compilador encuentra la definición de una función, él reserva suficiente memoria para manejar las sentencias en una función y almacena la dirección de la primera sentencia con el nombre de la función.

Definición de una función: Bloque de sentencias

- La definición de una función incluye el bloque de sentencias el cual contiene todas las instrucciones que ejecuta una función.
- Un bloque de sentencias un grupo de una o más sentencias encerradas entre llaves `{ }`. Inclusive si la función tiene una sola sentencia se debe encerrar entre llaves.

```
int sum (int a, int b) {  
    return (a+b);  
}
```

Declaración de variables en la definición de una función

- Un bloque de sentencias de una función puede contener cualquier número de declaraciones de variables.
- Las variables pueden ser declaradas en cualquier parte del bloque de sentencias siempre y cuando sea antes de ser empleada por primera vez.
- Una buena práctica de programación es declarar las variables al inicio del bloque de funciones.

```
int sum (int a, int b){  
    int sum;  
    sum = a + b;  
    return sum;  
}
```

Parámetros de una función

- La mayoría de funciones requieren información del código que las llaman. La forma más común de pasar información a una función es a través de la lista de parámetros.
- También se puede pasar información mediante variables globales, sin embargo, es una buena práctica de programación evitar usar variables globales para pasar información a funciones. Esto depende de las circunstancias de cada programa.
- Existen dos formas de pasar valores mediante la lista de parámetros: por valor y por referencia.

Parámetros de una función: Paso de argumentos por valor

- Cuando se llama a una función, una copia de los valores de los argumentos son pasados a la función, es decir, lo importante en el paso por valor es el valor del argumento.
- El compilador reservará la cantidad de memoria necesaria para mantener éstos valores. Es por esto que es muy importante especificar los tipos de datos de los parámetros en el prototipo de una función.
- Como se realiza una copia del valor de los argumentos para ser empleados por la función, los cambios de valor que se realicen en los parámetros de la función no afectarán a las variables utilizadas originalmente como argumentos en la llamada a la función.

Ejemplo de paso de argumentos por valor

```
void cambiar(int num) {  
    num = 4;  
}  
  
void main(void) {  
    int val = 2;  
    cambiar(val); // Se envía el valor de val val += val;  
// val = 2 + 2 = 4  
}
```

Cuando `main()` llama a `cambiar`, el valor del argumento `val` es pasado a la función, no su dirección. La función almacena una copia del valor de `val` en una dirección de memoria reservada para el parámetro `num`. El valor en esta dirección de memoria es cambiado por `cambiar()` pero dicho cambio no tiene efecto en el argumento `val`.

Paso de argumentos por referencia

- A diferencia del paso por valor, en el paso por referencia los parámetros no copian el valor de los argumentos, sino que comparten su valor. Por lo que cuando se cambia el valor del parámetro también cambia el valor de la variable utilizada como argumento en la llamada a la función.
- Para realizar un paso de argumentos por referencia los parámetros de la función deben ser punteros (*) y lo que se pasa como argumento a la función en la llamada es la dirección en memoria donde se encuentra el valor del argumento (&).

Ejemplo de paso de argumentos por referencia

```
void cambiar(int *num) {  
    *num = 4;  
}  
  
void main(void) {  
    int val = 2;  
    cambiar(&val); // Se envía la dirección de val  
    val += val;    // val = 4 + 4 = 8  
}
```

Cuando `main()` llama a la función se le envía como argumento la dirección del `val` y no su valor. La asignación realizada por la función usa el operador de desreferencia `*`, entonces se le asigna 4 a la dirección de memoria donde se encuentra el valor de `val`.

10. PREPROCESADOR DE C

- El preprocessador de C (cpp) es el primer programa que se llama en el proceso de compilación. Éste procesa el código fuente antes del compilador.
- El preprocessador lee el código fuente línea por línea y realiza las acciones sobre las directivas del preprocessador que va encontrando.
- Las principales funciones del preprocessador son: inclusión de archivos (**#include**), definición de constantes simbólicas (**#define**, **#undef**), definición de macros, compilación condicional de código (**#ifdef**, **#endif**), generación de mensajes de error (**#error**) e inclusión de código de ensamblador (**#asm**, **#endasm**).

Sintaxis de las directivas del preprocesador

- Cuando cualquier línea de código fuente empieza con **#**, es un comando para el preprocesador y se le llama directiva del preprocesador, además, se asume que la línea entera es parte de la misma directiva.

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

- Para continuar una sola directiva en más de una sola línea se emplea \ al final de la línea. Cuando éste carácter aparece el preprocesador concatena el contenido de la siguiente línea al final de la directiva actual.

```
#define PI \
    3.1416;
```

Espacios en blanco en el preprocesador

- A diferencia del compilador de C, los espacios en blanco son muy importantes en el preprocesador. Por ejemplo un código C:

```
int sum1(int a, int b);  
int sum2(int a, int b);
```

- El siguiente es un ejemplo de dos macros:

```
#define sum1 (a,b) (a+b);  
#define sum2 (a,b) (a+b);
```

- **sum1** está definida como una constante simbólica y no como una macro como se esperaba al igual que **sum2**. Así **sum1** será expandida como:

```
(a,b) (a+b);
```

Inclusión de archivos

- La directiva **#include** le indica al preprocesador que reemplace dicha directiva con los contenidos del archivo especificado.
- En sistemas empotrados los archivos de cabeceras se emplean usualmente para establecer definiciones de los recursos de hardware (**#include <machine.h>**).
- Si el preprocesador no puede encontrar el archivo especificado generará un error y terminará el proceso. Entonces surge una pregunta ¿Dónde busca el preprocesador los archivos de cabeceras?.

Búsqueda de archivos de inclusión <archivo.h> y “archivo.h”

- ✓ <**archivo.h**> Si se encierra el nombre del archive entre <> el preprocesador buscará el archivo en una ubicación del sistema determinada por el compilador que se está empleando. En general éste tipo de paréntesis generará dos tipos de búsqueda: 1) Buscará en un directorio o lista de directorios que el usuario le indica al compilador para que busque el archivo de cabecera, 2) Buscará en un directorio o lista de directorios especificados por el sistema operativo.

- ✓ “**archivo.h**” Las comillas dobles le indican al preprocesador que busque el archivo en el mismo lugar que el código fuente el cual contiene la directiva.

Constantes simbólicas: Macro objetos

- La directiva **#define** le indica al preprocesador que genere una constante simbólica o macro objeto.

```
#define PI 3.14
```

- Hay dos razones por las cuales las constantes simbólicas son útiles: 1) aclaran código ambiguo, 2) facilitan la mantenibilidad de código.
- También existe la posibilidad que se quiere indefinir del todo alguna constante simbólica o bien redefinir su valor. Esto se logra con la directiva **#undef**. El preprocesador dará un error si intenta definir un símbolo ya definido anteriormente.

```
#undef PI  
#define PI 3.14
```

Constantes simbólicas: Símbolos en blanco

- Otra característica útil de las constantes simbólicas es cuando ellas se definen no tienen que tener un valor asociado a ellas. Por ejemplo:

```
#define DEPURACION
```

- Ésta directiva le indica al compilador que ubique **DEPURACION** en la lista de símbolos con ningún valor asociado. En el caso particular de éste ejemplo **DEPURACION** puede actuar como una bandera que puede indicar al preprocesador que compile cierto código que muestra información para depuración.

Definición de macro funciones

- Las macro funciones son una característica muy ponderosa del preprocesador de C. Los macros son definidos mediante la directiva **#define** y a diferencia de los macro objetos, éstos reciben parámetros.
- Por convención los nombre de las macros se escriben en mayúsculas.
- Cuando el preprocesador encuentra una referencia a una macro realiza una reemplazo de texto con la definición de la macro y retiene la lista argumentos.

```
#define MAX(a,b) ((a)<(b) ? (a) : (b))

max = MAX(a, b);
//Se expande a:
max = ((a)<(b) ? (a) : (b));
```

Expansión de macros

- Es posible pasar expresiones como argumentos a una macro función.
- Cuando se pasan expresiones a funciones éstas son primero evaluadas y luego los valores resultantes son recibidos por la función. Como el preprocesador simplemente realiza un reemplazo de texto, no evalúa las expresiones antes de pasarla a la macro.
- Es por eso que el uso de paréntesis en macros es necesario para preservar la precedencia correcta.

```
// Macro incorrecta
#define CUADRADO(x) x * x
r = CUADRADO(a+1);
r = a+1 * a+1;
```

```
// Macro correcta
#define CUADRADO(x) ((x)*(x))
r = CUADRADO(a+1);
r = (a+1)*(a+1);
```

La función `defined()`

- La función `defined()` puede ser empleada con `#if`. Ésta retorna 1 si el símbolo ésta definido no importa si tiene un valor numérico asociado al símbolo o no y si el símbolo no está definido retorna 0.

```
#define DEPURACION  
#if defined(DEPURACION)  
    #include <depuracion.h>  
#endif
```

- Es posible emplear también `!defined()` para verificar si un símbolo no está definido. Ésta expresión retornará 1 si el símbolo no está definido y 0 si está definido.

```
#if !defined(DEPURACION)  
    #include <machine.h>  
#endif
```

Compilación condicional de código `#if` y `#endif`

- El preprocesador soporta directivas las cuales permiten la compilación condicional de código fuente.
- Las directivas `#if` y `#endif` incluyen código cuando la expresión evaluada por `#if` es un valor entero diferente de 0.

```
#define DEPURACION 1
#if DEPURACION
    #include <depuracion.h>
#endif
```

- Debido a que `#if` acepta expresiones como argumentos es posible hacer los siguientes:

```
#define DEPURACION 1
#if DEPURACION == 1
    #include <depuracion.h>
#endif
```

La directiva #else

- El preprocesador de C posee la habilidad de elegir entre compilar dos bloques de código mediante el uso de la directiva **#else**.

```
#define DEPURACION 1
#if DEPURACION == 1
    #include <depuracion.h>
#else
    #include <machine.h>
#endif
```

- También es posible construir una estructura tipo **switch-case** de bloques de compilación mediante el uso de la directiva **#elif**.

```
#define ESTADO DEPURACION

#if ESTADO == DEPURACION
    #include <depuracion.h>
#elif ESTADO = PRUEBAS
    #include <pruebas.h>
#elif ESTADO == HARDWARE
    #include <machine.h>
#endif
```

Directivas `#ifdef` y `#ifndef`

- Si no se quiere usar `defined()` o `!defined()`, se puede emplear equivalentemente las directivas `#ifdef` y `#ifndef`.

```
#define DEPURACION
#ifndef DEPURACION
    #include <depuracion.h>
#endif
#define DEPURACION
    #include <machine.h>
#endif
```

Generación de mensajes de error

- La directiva `#error` le indica al preprocesador que se detenga y produzca un mensaje de error:

```
#if ESTADO == DEPURACION
    #include <debug.h>
#elif ESTADO == HARDWARE
    #include <machine.h>
#else
    #error Estado no válido
#endif
```

Directiva #pragma

- El C estándar permite la definición de extensiones específicas para un compilador con la directiva **#pragma**.
- La directiva **#pragma**, permite a los compiladores definir sus directivas particulares (que no corresponden a nada establecido en el estándar ANSI) sin interferir con otros compiladores soporten este tipo de directivas. Si una directiva **#pragma** no es reconocida es ignorada.
- Ésta directiva es empleada en sistemas empotrados para describir recursos del hardware objetivo tales como memoria disponible, puertos y conjuntos de instrucciones especializados.

Inclusión de código ensamblador

- Muchos de los compiladores para sistemas empotrados proveen medios para incorporar lenguaje ensamblador.
- Una de las formas comunes es mediante las directivas del preprocesador **#asm** y **#endasm**. Cada línea entre ambas directivas se asume como código ensamblador y será procesado por el macro ensamblador.

```
int funcion () {  
    #asm  
    // Código ensamblador  
    #endasm  
}
```

11. BIBLIOTECA ESTÁNDAR DE C

- La biblioteca estándar de C (también conocida como libc) es una recopilación de ficheros cabecera y bibliotecas con rutinas que implementan operaciones comunes, funciones matemáticas, de tratamiento de cadenas, conversiones de tipo y entrada/salida por consola o por ficheros.
- La biblioteca estándar de ANSI C consta de 24 ficheros cabecera que pueden ser incluidos en un proyecto de programación con una simple directiva. Cada cabecera contiene la declaración de una o más funciones, tipos de datos y macros.

Cabeceras

Cabecera	Descripción
<assert.h>	Contiene la macro assert (aserción), utilizada para detectar errores lógicos y otros tipos de fallos en la depuración de un programa.
<complex.h>	Conjunto de funciones para manipular números complejos (nuevo en C99).
<ctype.h>	Contiene funciones para clasificar caracteres según sus tipos o para convertir entre mayúsculas y minúsculas independientemente del conjunto de caracteres (típicamente ASCII o alguna de sus extensiones).
<errno.h>	Para testar los códigos de error devueltos por las funciones de biblioteca.
<fenv.h>	Para controlar entornos en coma flotante (nuevo en C99).
<float.h>	Contiene la definición de constantes que especifican ciertas propiedades de la biblioteca de coma flotante, como la diferencia mínima entre dos números en coma flotante (_EPSOLON), el número máximo de dígitos de precisión (_DIG), o el rango de valores que se pueden representar (_MIN, _MAX).
<inttypes.h>	Para operaciones de conversión con precisión entre tipos enteros (nuevo en C99).
<iso646.h>	Para utilizar los conjuntos de caracteres ISO 646 (nuevo en NA1).
<limits.h>	Contiene la definición de constantes que especifican ciertas propiedades de los tipos enteros, como rango de valores que se pueden representar (_MIN, _MAX).
<locale.h>	Para la función setlocale() y las constantes relacionadas. Se utiliza para seleccionar el entorno local apropiado (configuración regional).
<math.h>	Contiene las funciones matemáticas comunes.

Cabeceras

Cabecera	Descripción
<setjmp.h>	Declara las macros setjmp y longjmp para proporcionar saltos de flujo de control de programa no locales.
<signal.h>	Para controlar algunas situaciones excepcionales como la división por cero.
<stdarg.h>	posibilita el acceso a una cantidad variable de argumentos pasados a una función.
<stdbool.h>	Para el tipo booleano (nuevo en C99).
<stdint.h>	Para definir varios tipos enteros (nuevo en C99).
<stddef.h>	Para definir varios tipos de macros de utilidad.
<stdio.h>	Proporciona el núcleo de las capacidades de entrada/salida del lenguaje C (incluye la venerable función printf).
<stdlib.h>	Para realizar ciertas operaciones como conversión de tipos, generación de números pseudo-aleatorios, gestión de memoria dinámica, control de procesos, funciones de entorno, de señalización (??), de ordenación y búsqueda.
<string.h>	Para manipulación de cadenas de caracteres.
<tgmath.h>	Contiene funcionalidades matemáticas de tipo genérico (<i>type-generic</i>) (nuevo en C99).
<time.h>	Para tratamiento y conversión entre formatos de fecha y hora.
<wchar.h>	Para manipular flujos de datos anchos y varias clases de cadenas de caracteres anchos (2 o más bytes por carácter), necesario para soportar caracteres de diferentes idiomas.
<wctype.h>	Para clasificar caracteres anchos (nuevo en NA1).

12. C EN SISTEMAS EMPOTRADOS

- Los siguientes son algunos aspectos que diferencian a la programación de C en sistemas empotrados.

- ✓ **Conocimiento del dispositivo:** ésto es fundamental para estar consciente de aspectos como las limitaciones del dispositivo, como por ejemplo memoria y capacidad de procesamiento. Además, un sistema empotrado tiene diferentes dispositivos que corren a diferentes velocidades por lo cual es esencial conocer el timing del dispositivo.
- ✓ **Acceso Directo al Hardware:** Los desarrolladores de sistemas empotrados están más cerca del hardware que otros programadores debido a que requieren una manipulación directa del hardware como por ejemplo a registros del procesador y otros dispositivos periféricos.
- ✓ **Flujo de programas:** El uso de ciclo infinitos así como la sentencia goto, es muy particular de los sistemas empotrados en ciertas circunstancias y aplicaciones.

- ✓ ***Uso de lenguaje ensamblador:*** Muchos desarrolladores prefieren escribir ciertos segmentos de código en lenguaje ensamblador por razones de eficiencia de código.
- ✓ ***Conocimiento mecánico:*** Las técnicas empleadas en un programa de un sistema empotrado se basan en el conocimiento de o los dispositivos o periféricos que maneja el sistema. En general muchos sistemas empotrados manejan dispositivos periféricos con ciertas propiedades físicas como por ejemplo motores, sensores, teclados, entre otros, y a la hora de programar software para sistemas empotrados se deben tener en consideración las propiedades físicas de dichos dispositivos.

- ✓ Byte Craft Limited. *First Steps with Embedded Systems*. Byte Craft Limited. 2002.

- ✓ Moreno, Alejandro. *Programación en C*. Wikilibros. 2006.