Universidad Nacional de Catamarca

Facultad de Tecnología y Ciencias Aplicadas

Ingeniería Electrónica

Lenguaje de Programación en C

Apuntes de cátedra: Informática

Docente: Lic. Ana María del Prado

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Tabla de contenido

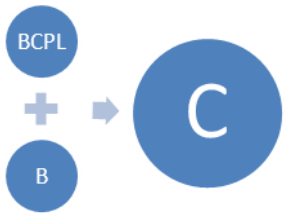
Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

UNIDAD 1 Lenguaje C.

1.1 Introducción al C.

C evolucionó de dos lenguajes de programación anteriores, BCPL y B. En 1967, Martin Richards desarrolló BCPL como un lenguaje para escribir software para sistemas operativos y compiladores. Ken Thompson, en su lenguaje B, modeló muchas de las características de C, luego del desarrollo de su contraparte en BCPL y, en 1970, utilizó B para crear las primeras versiones del sistema operativo UNIX en los laboratorios Bell, sobre una computadora DEC PDP-7. Tanto BCPL como B eran lenguajes "sin tipo" (cada dato ocupaba una "palabra" en memoria y, por ejemplo, el trabajo de procesar un elemento como un número completo o un número real, era responsabilidad del programador).



El lenguaje C evolucionó a partir de B; dicha evolución estuvo a cargo de Dennis Ritehie en los laboratorios BeIl y, en 1972, se implementó en una computadora DEC PDP- II C utiliza muchos conceptos importantes de BCPL y E cuando agrega tipos de datos y otras características. Inicialmente, C se hizo popular como lenguaje de desarrollo para el sistema operativo UNIX. En la actualidad, la mayoría de los sistemas operativos están escritos en C y/o C++. C se encuentra disponible para la mayoría de las computadoras, y es independiente del hardware. Con un diseño cuidadoso, es posible escribir programas en C que sean portables para la mayoría de las computadoras.

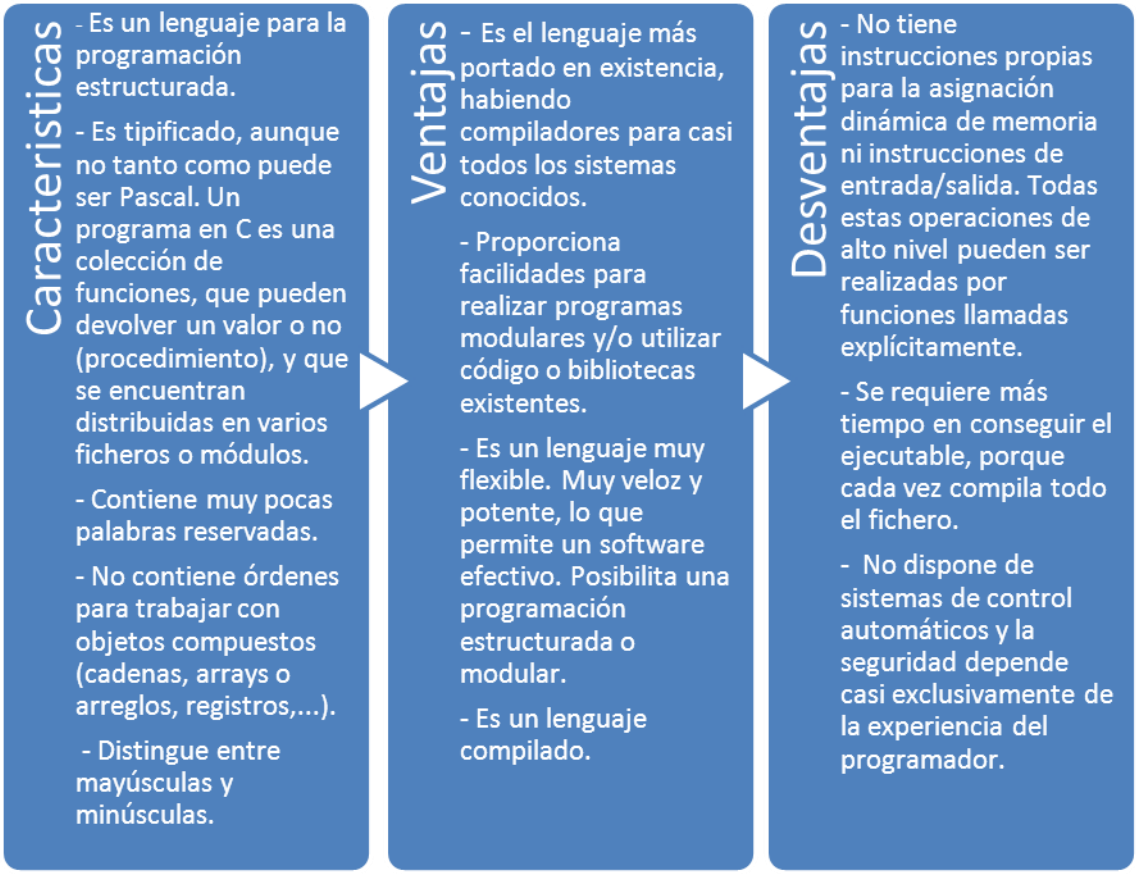
Para fines de la década de los setenta, C evolucionó a lo que ahora se conoce como "C tradicional", "C clásico", o "C de Kernigham y Ritchie".

La amplia utilización de C para distintos tipos de computadoras (en ocasiones llamadas plataformas de hardware) ocasionó, muchas variantes. Éstas eran similares, pero a menudo incompatibles, lo que se volvió un problema serio para los desarrolladores que necesitaban escribir programas que se pudieran ejecutar en distintas plataformas. Entonces se hizo evidente la necesidad de una versión estándar de C. En 1983, se creó el comité técnico X3J11 bajo la supervisión del American National Standards Comittee on Computers and Information Processing (X3), para "proporcionar una definición clara del lenguaje e independiente de la computadora". En 1989, el estándar fue aprobado; éste estándar se actualizó en 1999. Al documento del estándar se le conoce como INCITS/ISO/IEC 9899-1999.

El lenguaje de programación C está caracterizado por ser de uso general, con una sintaxis sumamente compacta y de alta portabilidad. C maneja los elementos básicos presentes en todas las computadoras: caracteres, números y direcciones. Esta particularidad, junto con el hecho de no poseer operaciones de entrada-salida, manejo de arreglo de caracteres, de asignación de memoria, etc, puede al principio parecer un grave defecto; sin embargo el hecho de que estas operaciones se realicen por medio de llamadas a Funciones contenidas en Librerías externas al lenguaje en sí, es el que

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

confiere al mismo su alto grado de portabilidad, independizándolo del "Hardware" sobre el cual corren los programas.
La descripción del lenguaje se realiza siguiendo las normas del ANSI C, por lo tanto, todo lo expresado será utilizable con cualquier compilador que se adopte.
El lenguaje C es uno de los lenguajes de programación más populares que existen hoy en día.



1.2 Variables y aritmética.

En un programa intervienen objetos sobre los que actúan las instrucciones que lo componen. Algunos de estos objetos tomarán valores a lo largo del programa.
Dependiendo de si pueden cambiar de valor o no, podemos distinguir dos tipos de objetos:

- Constante: Objeto –referenciado mediante un identificador- al que se le asignará un valor que no se podrá modificar a lo largo del programa.
- Variable: Objeto –referenciado por un identificador– que *puede tomar distintos valores* a lo largo del programa.

Será misión del programador asignar el *identificador* que desee a cada constante y variable. El identificador o nombre de cada objeto sirve para identificar sin ningún tipo de ambigüedad a cada objeto, diferenciándolo de los demás objetos que intervienen en el programa. En C++ hay que indicar el nombre de las constantes y variables que vamos a usar, para que el compilador pueda asociar internamente a dichos nombres las posiciones de memoria correspondientes.

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Para esto último es preciso también que con dichos nombres, se indique explícitamente para cada constante o variable el tipo de los datos que pueden contener.
En C++ existen una serie de tipos básicos predefinidos, así como herramientas para la construcción de tipos más complejos.

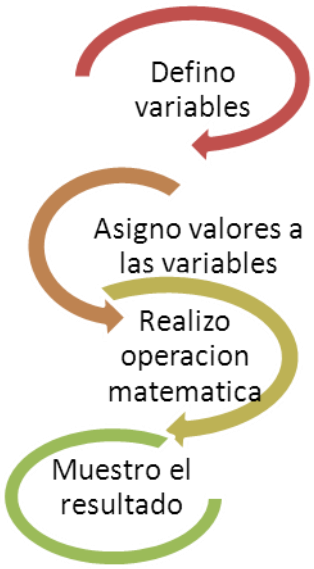
Si se quiere imprimir los resultados de multiplicar un número fijo por otro que adopta valores entre 0 y 9, la forma normal de programar esto sería crear una CONSTANTE para el primer número y un par de variables para el segundo y para el resultado del producto. Una variable, en realidad, no es más que un nombre para identificar una (o varias) posiciones de memoria donde el programa guarda los distintos valores de una misma entidad. Un programa debe definir a todas las variables que utilizará, a fin de indicarle al compilador de que tipo serán, y por lo tanto cuanta memoria debe destinar para albergar a cada una de ellas. Por ejemplo:

```
#include <stdio.h>
main()
{
        int multiplicador;
        int multiplicando;
        int resultado;

        multiplicador = 1000 ;
        multiplicando = 2 ;

        resultado = multiplicando * multiplicador;


        printf("Resultado = %d\n", resultado);
        return 0;
}
```



En la función main() defino mis variables como números enteros, es decir del tipo "int" seguido de un identificador (nombre) de la misma . Este identificador puede tener la cantidad de caracteres que se desee, sin embargo de acuerdo al Compilador que se use, este tomará como significantes sólo los primeros n de ellos; siendo por lo general n igual a 32.
Es conveniente darle a los identificadores de las variables, nombres que tengan un significado que luego permita una fácil lectura del programa. Los identificadores deben comenzar con una letra ó con el símbolo de subrayado "_", pudiendo continuar con cualquier otro carácter alfanumérico ó el símbolo "_" .El único símbolo no alfanumérico aceptado en un nombre es el "_". El lenguaje C es sensible a mayúsculas y minúsculas; así tomará como variables distintas a una llamada "variable" , de otra escrita como "VARIABLE". Es una convención entre los programadores de C escribir los nombres de las variables y las funciones con minúsculas, reservando las mayúsculas para las constantes.

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

El compilador dará como error de "Definición incorrecta" a la definición de variables con nombres del tipo de: 4pesos $variable primer-variable !variable
En las dos líneas subsiguientes a la definición de las variables, que puedo ya asignarles valores (1000 y 2) y luego efectuar el cálculo de la variable "resultado".
La función printf(),nos mostrará la forma de visualizar el valor de una variable. Insertada en el texto a mostrar, aparece una secuencia de control de impresión "%d" que indica, que en el lugar que ella ocupa, deberá ponerse el contenido de la variable (que aparece luego de cerradas las comillas que marcan la finalización del texto, y separada del mismo por una coma) expresado como un número entero decimal. Así, si compilamos y corremos el programa, obtendremos una salida:

Resultado = 2000

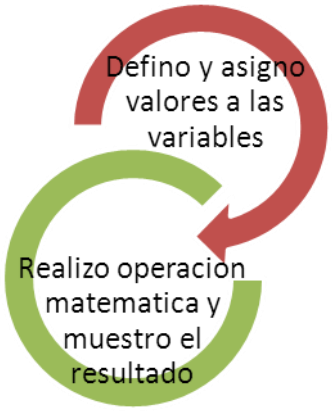
Inicialización de variables:

Las variables del mismo tipo pueden definirse mediante una definición múltiple separándolas mediante "," a saber:
int multiplicador, multiplicando, resultado;
Esta sentencia es equivalente a las tres definiciones separadas en el ejemplo anterior. Las variables pueden también ser inicializadas en el momento de definirse.
int multiplicador = 1000, multiplicando = 2, resultado;

De esta manera el ejemplo podría escribirse:

```
#include <stdio.h>
main()
{
        int multiplicador=1000;
        int multiplicando=2;

        printf("Resultado = %d\n", multiplicando *
        multiplicadorresultado);
        return 0;
}
```



En la primera sentencia se definen e inicializan simultáneamente ambas variables. La variable "resultado" la hemos hecho desaparecer ya que es innecesaria. Si analizamos la función printf() vemos que se ha remplazado "resultado" por la operación entre las otras dos variables. Esta es una de las particularidades del lenguaje C: en los parámetros pasados a las funciones pueden ponerse operaciones.

1.3 Constantes.

Aquellos valores que, una vez compilado el programa no pueden ser cambiados, como por ejemplo los valores literales que hemos usado hasta ahora en las inicializaciones de las variables (1000, 2, 'a', '\n', etc), suelen denominarse CONSTANTES. Como dichas

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

constantes son guardadas en memoria de la manera que al compilador le resulta más eficiente suelen aparecer ciertos efectos secundarios.

Constantes de carácter.
Ej. ‘a’, ‘0’, ‘\0x5’, ‘\0’, ‘\n’, ‘\t’, ‘$’, ’\\’, NULL

Constantes enteras.
Ej. 5, +5, -5, \05, \0x5, 5L, 5U, 5lu, etc.

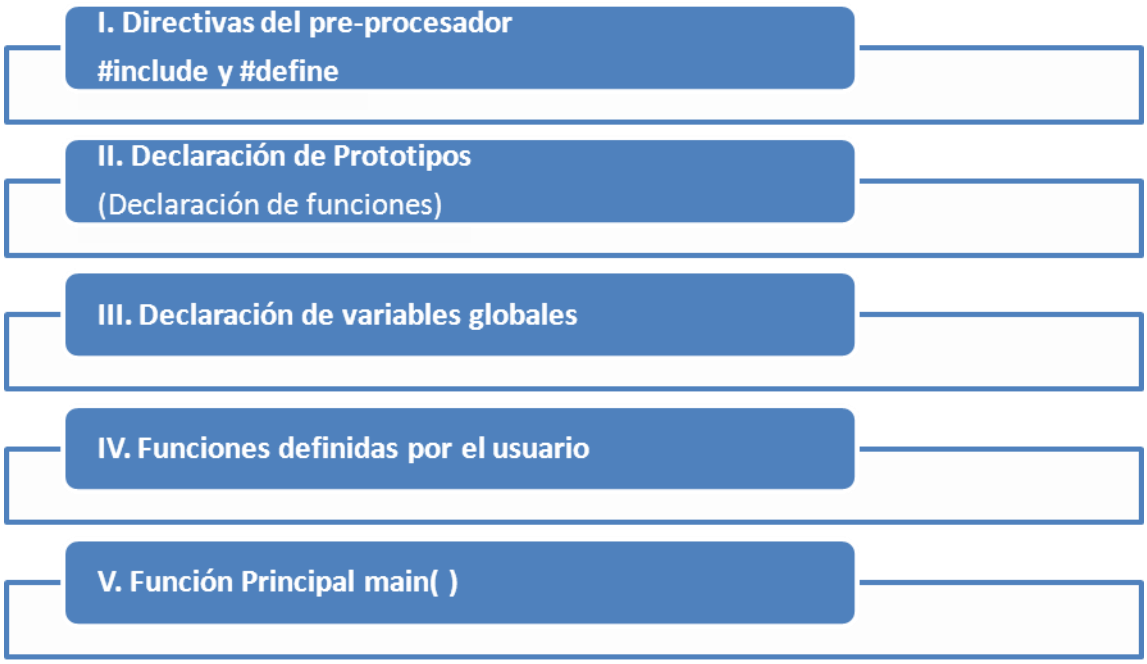
Constantes reales.
Ej. 0.5f, 0.5, 5e-01f, 5.0e-01, (float)5, etc.

Constantes de texto (Cadenas o “Strings”)
“Esto es una cadena…”

1.4 Constantes simbólicas.

Por lo general es una mala práctica de programación colocar en un programa constantes en forma literal (sobre todo si se usan varias veces en el mismo) ya que el texto se hace difícil de comprender y aún más de corregir, si se debe cambiar el valor de dichas constantes. Se puede en cambio asignar un símbolo a cada constante, y remplazarla a lo largo del programa por el mismo, de forma que este sea más legible y además, en caso de querer modificar el valor, bastará con cambiarlo en la asignación. El compilador, en el momento de crear el ejecutable, remplazará el símbolo por el valor asignado. Para dar un símbolo a una constante bastará, en cualquier lugar del programa (previo a su uso) poner la directiva: #define, por ejemplo:
#define VALOR_CONSTANTE 342
#define PI 3.1416

1.5 Estructura de un programa C.



Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Ejemplo de un programa escrito en C++ que muestra el famoso “Hola Mundo”, en pantalla antes de finalizar.

```
// Descripción: Este es el programa “Hola Mundo”.
#include <stdio.h> // Para usar printf
void main ()
{
        Printf( “Hola Mundo\n” );
}
```

El programa comienza con un comentario y en el cual se da información sobre el propósito del programa. Esto no es sintácticamente necesario, pero es importante al momento del mantenimiento del código para que otros desarrolladores entiendan el código y se encuentre documentado. A continuación nos encontramos con una operación de inclusión. Sin entrar en detalles sobre lo que internamente significan dichas inclusiones, lo que hacemos mediante las mismas es indicarle al compilador que pretendemos hacer uso de las funcionalidades que nos proporciona esa biblioteca. En este caso concreto queremos emplear la biblioteca (stdio) para incluir la funcion printf.
El algoritmo comienza en la línea void main(). Lo que sí debemos saber es que todo programa que realicemos tendrá un algoritmo principal (el que empezará a realizarse cuando ejecutemos el programa).
A continuación encontramos el cuerpo del algoritmo principal, que abarca desde la llave de apertura ({) hasta la llave de cierre (}). Siempre aparecerán estas llaves delimitando el cuerpo de un algoritmo. En este caso, el algoritmo consta de una sola orden, mostrar en pantalla el mensaje “Hola Mundo” y saltar a la línea siguiente. Como puede verse, esto se expresa haciendo uso del objeto predefinido (en stdio precisamente) printf.
Todo lo que coloquemos entre comillas hacia dicho objeto será mostrado en pantalla. En el ejemplo se redirige en primer lugar la cadena “Hola Mundo”, y a continuación la indicación de salto de linea (\n). La instrucción termina mediante un punto y coma (;), que la separa de instrucciones posteriores.

1.6 Clases de datos standard.

En lenguaje C, los datos que utilizan los programas son básicos (simples predefinidos o estándares) o derivados.

Los tipos básicos en lenguaje C se clasifican en:
Numéricos: Entero (int) Real (float y double)
Carácter (char)

Los tipos complejos se clasifican en:
Arrays (vectores)
Estructuras
Uniones

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

VARIABLES DEL TIPO ENTERO

De acuerdo a la cantidad de bytes que reserve el compilador para este tipo de variable, queda determinado el "alcance" ó máximo valor que puede adoptar la misma. Debido a que el tipo int ocupa dos bytes su alcance queda restringido al rango entre -32.768 y +32.767 (incluyendo 0). En caso de necesitar un rango más amplio, puede definirse la variable como "long int nombre_de_variable" ó en forma más abreviada "long nombre_de_variable" Declarada de esta manera, nombre_de_variable puede alcanzar valores entre - 2.347.483.648 y +2.347.483.647. A la inversa, si se quisiera un alcance menor al de int, podría definirse "short int " ó simplemente "short", aunque por lo general, los compiladores modernos asignan a este tipo el mismo alcance que "int". Debido a que la norma ANSI C no establece taxativamente la cantidad de bytes que ocupa cada tipo de variable, sino tan sólo que un "long" no ocupe menos memoria que un "int" y este no ocupe menos que un "short" los alcances de los mismos pueden variar de compilador en compilador. Para variables de muy pequeño valor puede usarse el tipo "char" cuyo alcance está restringido a -128, +127 y por lo general ocupa un único byte. Todos los tipos citados hasta ahora pueden alojar valores positivos ó negativos y, aunque es redundante, esto puede explicitarse agregando el calificador "signed" delante; por ejemplo:
signed int
signed long
signed long int
signed short
signed short int
signed char
Si en cambio, tenemos una variable que sólo puede adoptar valores positivos (como por ejemplo la edad de una persona) podemos aumentar el alcance de cualquiera de los tipos, restringiéndolos a que sólo representen valores sin signo por medio del calificador "unsigned".

En la TABLA 1 se resume los alcances de distintos tipos de variables enteras

VARIABLES DEL TIPO NUMERO ENTERO	BYTES	VALOR MINIMO	VALOR MAXIMO
signed char	1	-128	127
unsigned char	1	0	255
signed short	2	-32.768	+32.767
unsigned short	2	0	+65.535
signed int	2	-32.768	+32.767
unsigned int	2	0	+65.535
signed long	4	-2.147.483.648	+2.147.483.647
unsigned long	4	0	+4.294.967.295

VARIABLES DE NUMERO REAL O PUNTO FLOTANTE

Un número real ó de punto flotante es aquel que además de una parte entera, posee fracciones de la unidad. En nuestra convención numérica solemos escribirlos de la siguiente manera: 2,3456, lamentablemente los compiladores usan la convención del

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

PUNTO decimal (en vez de la coma). Así el numero Pi se escribirá: 3.14159 Otro formato de escritura, normalmente aceptado, es la notación científica.
Por ejemplo podrá escribirse 2.345E+02, equivalente a 2.345 * 100 ó 234.5 De acuerdo a su alcance hay tres tipos de variables de punto flotante, las mismas están descriptas en la TABLA 2

VARIABLES DE PUNTO FLOTANTE TIPO	BYTES	VALOR MINIMO	VALOR MAXIMO
float	4	3.4E-38	3.4E+38
double	8	1.7E-308	1.7E+308
long double	10	3.4E-4932	3.4E+4932

VARIABLES DE TIPO CARACTER

El lenguaje C guarda los caracteres como números de 8 bits de acuerdo a la norma ASCII extendida, que asigna a cada carácter un número comprendido entre 0 y 255 (un byte de 8 bits) Es común entonces que las variables que vayan a alojar caracteres sean definidas como:
char c ;
Sin embargo, también funciona de manera correcta definirla como
int c ;
Esta última opción desperdicia un poco más de memoria que la anterior, pero en algunos casos particulares presenta ciertas ventajas . Pongamos por caso una función que lee un archivo de texto ubicado en un disco. Dicho archivo puede tener cualquier carácter ASCII de valor comprendido entre 0 y 255. Para que la función pueda avisarme que el archivo ha finalizado deberá enviar un número NO comprendido entre 0 y 255 (por lo general se usa el -1 , denominado EOF, fin de archivo ó End Of File), en este caso dicho número no puede ser mantenido en una variable del tipo char, ya que esta sólo puede guardar entre 0 y 255 si se la define unsigned ó no podría mantener los caracteres comprendidos entre 128 y 255 si se la define signed (ver TABLA 1). El problema se obvia fácilmente definiéndola como int. Las variables del tipo carácter también pueden ser inicializadas en su definición, por ejemplo es válido escribir:
char c = 97;
Para que c contenga el valor ASCII de la letra "a", sin embargo esto resulta algo engorroso , ya que obliga a recordar dichos códigos . Existe una manera más directa de asignar un carácter a una variable; la siguiente inicialización es idéntica a la anterior:
char c = 'a' ;
Es decir que si delimitamos un carácter con comilla simple , el compilador entenderá que debe suplantarlo por su correspondiente código numérico . Lamentablemente existen una serie de caracteres que no son imprimibles , en otras palabras que cuando editemos nuestro programa fuente (archivo de texto) nos resultará difícil de asignarlas a una variable ya que el editor las toma como un COMANDO y no como un carácter . Un caso típico sería el de "nueva línea" ó ENTER .
Con el fin de tener acceso a los mismos es que aparecen ciertas secuencias de escape convencionales. Las mismas están listadas en la TABLA 3 y su uso es idéntico al de los caracteres normales, así para resolver el caso de una asignación de "nueva línea " se escribirá:

```
char c = '\n' ; /* secuencia de escape */
```


Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

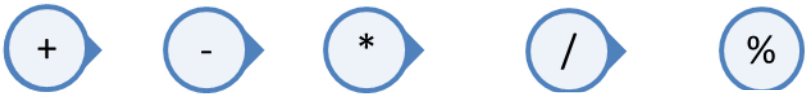
SECUENCIAS DE ESCAPE	SIGNIFICADO	VALOR ASCII (decimal)	VALOR ASCII (hexadecimal)
'\n'	nueva línea	10	0x0A
'\r'	retorno de carro	13	0x0D
'\f'	nueva página	2	x0C
'\t'	tabulador horizontal	9	0x09
'\b'	retroceso (backspace)	8	0x08
'\"	comilla simple	39	0x27
'\"'	comillas	4	0x22
'\\ '	barra	92	0x5C
'\? '	interrogación	63	0x3F
'\nnn'	cualquier carácter (donde nnn es el código ASCII expresado en octal)		
'\xnn'	cualquier carácter (donde nn es el código ASCII expresado en hexadecimal)		

1.7 Operadores y expresiones.

Operadores:

C++ proporciona operadores para realizar las operaciones más usuales, como pueden ser las siguientes:

- *Operaciones aritméticas:* se proporcionan los operadores binarios



suma resta multiplicación división resto o módulo

Asimismo, existe el operador unario – para la inversión de signo.

Ejemplos:

```
int num1, num2,
num3;
…
num1 = -10*num2;
num2 = num1%7;
num3 = num1/num2;
```

- *Operaciones relacionales:* se proporcionan los operadores binarios

igual distinto Mayor que Menor que Menor o igual que Mayor o igual que

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

El resultado de la operación relacional será un valor lógico.

Ejemplos:

```
int num1, num2;
bool b1, b2;
…
b1 = num2 == num1;
b2 = (num1-num2) >= (num2*2);
```

• *Operaciones lógicas*: se proporcionan los operadores binarios

conjunción

disyunción

También se dispone del operador unario ! con el significado de negación.

Ejemplos:

```
int num1, num2;
bool b1, b2;
…
b1 = !(num2 == num1); // equivale a num2!=num1
b2 = (num1>num2) && (num2<0);
```

e frecuente en el que el valor de una variable use como operando en una expresión aritmética ó lógica cuyo resultado se asigna a la propia variable. Dicha sintaxis permite simplificar asignaciones del tipo v = v <op> E, permitiendo expresarlas como v <op>= E.

Por ejemplo:

```
int x, y, z;
…
x *= x+y/z; // equivale a x = x * (x+y/z);
z += 1; // equivale a z = z + 1;
```

• *Operaciones de incremento y decremento*:

Precisamente en relación con este último ejemplo, C++ proporciona una forma sintáctica aún más simple mediante el uso de los denominados operadores de incremento (++) y decremento (--). Éstos son operadores unarios que sólo pueden aplicarse sobre variables (no sobre expresiones) y que modifican el valor de la misma, incrementándola o decrementándola según corresponda.

Por ejemplo:

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

```
int x, y, z;
…
z++; // equivale a z = z + 1 o a z +=1
--x; // equivale a x = x - 1 o a x -=1
```

Tanto el ++ como el -- pueden ponerse antes o después de la variable que deseamos incrementar o decrementar. La diferencia entre ambas formas sintácticas estriba en su uso dentro de expresiones más complejas: si el operador se coloca antes de la variable, ésta se incrementa (o decrementa) y es dicho valor modificado el que se emplea en la expresión; si el operador se sitúa después, se usa el valor actual de la variable en la expresión y luego se modifica.

Por ejemplo:

```
int x, y, z;
…
x = --y * z++; // equivale a y = y - 1
// x = y * z;
// z = z + 1;
```

• Operaciones de asignación

Anteriormente definimos a una asignación como la copia del resultado de una expresión sobre otra, esto implica que dicho valor debe tener LUGAR (es decir poseer una posición de memoria) para alojar dicho valor. Es por lo tanto válido escribir
a = 17;
pero no es aceptado , en cambio
17 = a ; /* incorrecto */
ya que la constante numérica 17 no posee una ubicación de memoria donde alojar al valor de a . Aunque parezca un poco extraño al principio las asignaciones, al igual que las otras operaciones, dan un resultado que puede asignarse a su vez a otra expresión. De la misma forma que (a + b) es evaluada y su resultado puedo copiarlo en otra variable: c = (a + b) ; una asignación (a = b) da como resultado el valor de b , por lo que es lícito escribir c = (a = b) ;
Debido a que las asignaciones se evalúan de derecha a izquierda, los paréntesis son superfluos, y podrá escribirse entonces:
c = a = b = 17;
Con lo que las tres variables resultarán iguales al valor de la contante. El hecho de que estas operaciones se realicen de derecha a izquierda también permite realizar instrucciones del tipo:
a = a + 17;
Significando esto que al valor que TENIA anteriormente a, se le suma la constante y LUEGO se copia el resultado en la variable.

Expresiones

Ya han aparecido algunos ejemplos de expresiones del lenguaje C en las secciones precedentes.
Una expresión es una combinación de variables y/o constantes, y operadores. La expresión es equivalente al resultado que proporciona al aplicar sus operadores a sus

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

operandos. Por ejemplo, 1+5 es una expresión formada por dos *operandos* (1 y 5) y un *operador* (+); esta expresión es equivalente al valor 6, lo cual quiere decir que allí donde esta expresión aparece en el programa, en el momento de la ejecución es evaluada y sustituida por su resultado. Una expresión puede estar formada por otras expresiones más sencillas, y puede contener paréntesis de varios niveles agrupando distintos términos. En C existen distintos tipos de expresiones.

Expresiones aritméticas

Están formadas por variables y/o constantes, y distintos operadores aritméticos e incrementales (+,-, *, /, %, ++, --). Como se ha dicho, también se pueden emplear paréntesis de tantos niveles como se desee, y su interpretación sigue las normas aritméticas convencionales. Por ejemplo, la solución de la ecuación de segundo grado:

$$X=\frac{-b\pm\sqrt[2]{b^2-4ac}}{2a}$$

Se escribe, en C en la forma:

```
x=(-b+sqrt((b*b)-(4*a*c)))/(2*a);
```

Donde, estrictamente hablando, sólo lo que está a la derecha del operador de asignación (=) es una expresión aritmética. El conjunto de la variable que está a la izquierda del signo (=), el operador de asignación, la expresión aritmética y el carácter (;) constituyen una ***sentencia***. En la expresión anterior aparece la llamada a la ***función de librería sqrt()***, que tiene como ***valor de retorno*** la raíz cuadrada de su único ***argumento***. En las expresiones se pueden introducir espacios en blanco entre operandos y operadores; por ejemplo, la expresión anterior se puede escribir también de la forma:

```
x = (-b + sqrt((b * b) - (4 * a * c)))/(2 * a);
```

Expresiones lógicas

Los elementos con los que se forman estas expresiones son ***valores lógicos***; *verdaderos* (***true***, o distintos de 0) y *falsos* (***false***, o iguales a 0), y los ***operadores lógicos*** `||`, `&&` y `!`. También se pueden emplear los ***operadores relacionales*** (`<`, `>`, `<=`, `>=`, `==`, `!=`) para producir estos valores lógicos a partir de valores numéricos. Estas expresiones equivalen siempre a un valor 1 (***true***) o a un valor 0 (***false***). Por ejemplo:

```
a = ((b>c)&&(c>d))||((c==e)||(e==b));
```

donde de nuevo la ***expresión lógica*** es lo que está entre el operador de asignación (=) y el (;). La variable **a** valdrá 1 si **b** es mayor que **c** *y* **c** mayor que **d**, ó si **c** es igual a **e** ó **e** es igual a **b**.

Expresiones generales

Una de las características más importantes (y en ocasiones más difíciles de manejar) del C es su flexibilidad para combinar expresiones y operadores de distintos tipos en una expresión que se podría llamar *general*, aunque es una expresión absolutamente ordinaria de C.

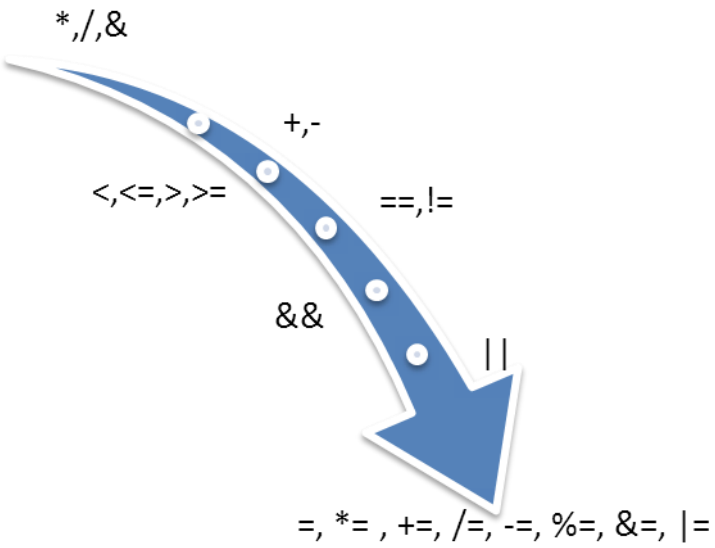
Recuérdese que el resultado de una expresión lógica es siempre un valor numérico (un 1 ó un 0); esto permite que cualquier expresión lógica pueda aparecer como sub-expresión en una expresión aritmética. Recíprocamente, cualquier valor numérico puede ser considerado como un valor lógico: ***true*** si es distinto de 0 y ***false*** si es igual a 0. Esto

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

permite introducir cualquier expresión aritmética como sub-expresión de una expresión lógica. Por ejemplo:
(a - b*2.0) && (c != d)
A su vez, *el operador de asignación* (=), además de introducir un nuevo valor en la variable que figura a su izda, *deja también este valor disponible para ser utilizado* en una expresión más general. Por ejemplo, supóngase el siguiente código que inicializa a 1 las tres variables **a**, **b** y **c**:
a = b = c = 1;
que equivale a:
a = (b = (c = 1));
En realidad, lo que se ha hecho ha sido lo siguiente. En primer lugar se ha asignado un valor unidad a **c**; el resultado de esta asignación es también un valor unidad, que está disponible para ser asignado a **b**; a su vez el resultado de esta segunda asignación vuelve a quedar disponible y se puede asignar a la variable **a**.

Reglas de precedencia y asociatividad

En el caso de que tengamos una expresión compleja con varios operadores, C++ proporciona reglas de precedencia que permiten determinar cuál es el orden en el que se aplican los operadores. Dichas reglas son las siguientes:
1. Los operadores unarios se aplican antes que los binarios. Si hay más de un operador unario aplicado sobre la misma variable o subexpresión, el orden de aplicación es de derecha a izquierda.
2. Los operadores binarios se aplican según el siguiente orden de precedencia, de mayor a menor nivel:



Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

UNIDAD 2 Entrada y Salida.

2.1 Acceso a la biblioteca estándar.

A diferencia de otros lenguajes, *C no dispone de sentencias de entrada/salida*. En su lugar se utilizan funciones contenidas en la librería estándar y que forman parte integrante del lenguaje.
Las funciones de entrada/salida (Input/Output) son un conjunto de funciones, incluidas con el compilador, que permiten a un programa recibir y enviar datos al exterior. Para su utilización es necesario incluir, al comienzo del programa, el archivo **stdio.h** en el que están definidos sus prototipos:
#include <stdio.h>
donde ***stdio*** proviene de ***standard-input-output***.

2.2. Entrada y Salida estándar: getchar, putchar, gets, puts.

Las macros **getchar()** y **putchar()** permiten respectivamente leer e imprimir *un sólo carácter* cada vez, en la entrada o en la salida estándar. La macro **getchar()** recoge un carácter introducido por teclado y lo deja disponible como valor de retorno. La macro **putchar()** escribe en la pantalla el carácter que se le pasa como argumento. Por ejemplo:
putchar('a');
escribe el carácter **a**. Esta sentencia equivale a printf("a");
mientras que
c = getchar();
equivale a
scanf("%c", &c);
Como se ha dicho anteriormente, **getchar()** y **putchar()** son *macros* y no *funciones*, aunque para casi todos los efectos se comportan como si fueran funciones. El concepto de ***macro*** se verá con más detalle en la siguiente sección. Estas macros están definidas en el fichero **stdio.h**, y su código es sustituido en el programa por el *preprocesador* antes de la compilación.
Por ejemplo, *se puede leer una línea de texto completa* utilizando **getchar()**:

```
int i=0, c;
char name[100];
while((c = getchar()) != '\n') // se leen caracteres hasta el '\n'
name[i++] = c; // se almacena el carácter en Name[]
name[i]='\0'; // se añade el carácter fin de cadena
```

gets(): Lee una cadena de caracteres introducida por el teclado y la sitúa en una dirección apuntada por su argumento de tipo puntero a carácter.

puts(): Escribe su argumento de tipo cadena en la pantalla seguida de un carácter de salto de línea.

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

2.3. Salida con formato printf: distintos formatos de salida.

La función **printf()** imprime en la unidad de salida (el monitor, por defecto), el texto, y las constantes y variables que se indiquen. La forma general de esta función se puede estudiar viendo su *prototipo*:

int printf("cadena_de_control", tipo arg1, tipo arg2, ...)

Explicación: La función **printf()** imprime el texto contenido en **cadena_de_control** junto con el valor de los otros argumentos, de acuerdo con los *formatos* incluidos en **cadena_de_control**. Los puntos suspensivos (...) indican que puede haber un número variable de argumentos. Cada formato comienza con el carácter (%) y termina con un *carácter de conversión*.

Considérese el ejemplo siguiente,

```
int i;
double tiempo;
float masa;
printf("Resultado nº: %d. En el instante %lf la masa vale %f\n", i, tiempo, masa);
```

En el que se imprimen 3 variables (**i**, **tiempo** y **masa**) con los formatos (**%d**, **%lf** y **%f**), correspondientes a los tipos (***int***, ***double*** y ***float***), respectivamente. La cadena de control se imprime con el valor de cada variable intercalado en el lugar del formato correspondiente.

Un número entero positivo, que indica la *anchura* mínima del campo en caracteres.
- Un signo (-), que indica *alineamiento* por la izda (el defecto es por la dcha).
- Un punto (.), que separa la anchura de la *precisión*.
- Un número entero positivo, la *precisión*, que es el nº máximo de caracteres a imprimir en un *string*, el nº de decimales de un *float* o *double*, o las cifras mínimas de un *int* o *long*.
- Un *cualificador*: una (h) para *short* o una (l) para *long* y *double*

Ejemplos de uso de la función **printf()**.

```
printf("Con cien cañones por banda,\nviento en popa a toda vela,\n");
```

El resultado serán dos líneas con las dos primeras estrofas de la famosa poesía. *No es posible partir* ***cadena_de_control*** *en varias líneas con caracteres* ***intro***, por lo que en este ejemplo podría haber problemas para añadir más estrofas. Una forma alternativa, muy sencilla, clara y ordenada, de escribir la poesía sería la siguiente:

```
printf("%s\n%s\n%s\n%s\n", "Con cien cañones por banda,", "viento en popa a toda
vela,", "no cruza el mar sino vuela,", "un velero bergantín.");
```

En este caso se están escribiendo 4 cadenas constantes de caracteres que se introducen como argumentos, con formato %s y con los correspondientes saltos de línea. Un ejemplo que contiene una constante y una variable como argumentos es el siguiente:

```
printf("En el año %s ganó %ld ptas.\n", "1993", beneficios);
```

donde el texto **1993** se imprime como cadena de caracteres (%s), mientras que **beneficios** se imprime con formato de variable ***long*** (%ld). Es importante hacer corresponder bien los formatos con el tipo de los argumentos, pues si no los resultados pueden ser muy diferentes de lo esperado.

La función **printf()** *tiene un valor de retorno de tipo int*, que representa el número de caracteres escritos en esa llamada.

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

2.4. Entrada con formato scanf.

La función scanf() es análoga en muchos aspectos a printf(), y se utiliza para leer datos de la entrada estándar (que por defecto es el teclado). La forma general de esta función es la siguiente:

int scanf("%x1%x2...", &arg1, &arg2, ...);

donde x1, x2, ... son los caracteres de conversión, mostrados en la Tabla 8.2, que representan los formatos con los que se espera encontrar los datos. La función scanf() devuelve como valor de retorno el número de conversiones de formato realizadas con éxito. La cadena de control de scanf()

puede contener caracteres además de formatos. Dichos caracteres se utilizan para tratar de detectar la presencia de caracteres idénticos en la entrada por teclado. Si lo que se desea es leer variables numéricas, esta posibilidad tiene escaso interés. A veces hay que comenzar la cadena de control con un espacio en blanco para que la conversión de formatos se realice correctamente.

En la función scanf() los argumentos que siguen a la cadena_de_control deben ser pasados por referencia, ya que la función los lee y tiene que trasmitirlos al programa que la ha llamado. Para ello, dichos argumentos deben estar constituidos por las direcciones de las variables en las que hay que depositar los datos, y no por las propias variables. Una excepción son las cadenas de caracteres, cuyo nombre es ya de por sí una dirección (un puntero), y por tanto no debe ir precedido por el operador (&) en la llamada.

Por ejemplo, para leer los valores de dos variables int y double y de una cadena de caracteres, se utilizarían la sentencia:

```
int n;
double distancia;
char nombre[20];
scanf("%d%lf%s", &n, &distancia, nombre);
```

en la que se establece una correspondencia entre n y %d, entre distancia y %lf, y entre nombre y %s. Obsérvese que nombre no va precedido por el operador (&). La lectura de cadenas de caracteres se detiene en cuanto se encuentra un espacio en blanco, por lo que para leer una línea completa con varias palabras hay que utilizar otras técnicas diferentes.

En los formatos de la cadena de control de scanf() pueden introducirse corchetes [...], que se utilizan como sigue. La sentencia,

scanf("%[AB \n\t]", s); // se leen solo los caracteres indicados lee caracteres hasta que encuentra uno diferente de ('A','B',' ','\n','\t'). En otras palabras, se leen sólo los caracteres que aparecen en el corchete. Cuando se encuentra un carácter distinto de éstos se detiene la lectura y se devuelve el control al programa que llamó a scanf(). Si los corchetes contienen un carácter (^), se leen todos los caracteres distintos de los caracteres que se encuentran dentro de los corchetes a continuación del (^). Por ejemplo, la sentencia, scanf(" %[^\n]", s); lee todos los caracteres que encuentra hasta que llega al carácter nueva línea '\n'. Esta sentencia puede utilizarse por tanto para leer líneas completas, con blancos incluidos. Recuérdese que con el formato %s la lectura se detiene al llegar al primer delimitador (carácter blanco, tabulador o nueva línea).

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Usos de printf y scanf	

printf("Mensaje");
printf("Mensaje %d",Variable);
printf("Mensaje %d",Variable:2:3);
cprintf("Mensaje");
	Escribe Mensaje en la pantalla
	Escribe Mensaje y el valor de la Variable en pantalla
	Escribe Mensaje y el valor de la Variable con 2 enteros y 3 decimales
	Escribe Mensaje en color especificado

scanf("%d",&Variable); scanf("%d %f",&Variable1,&Variable2);	Asigna valor entero a Variable Asigna valor entero a Variable1 y valor real a Variable2

2.5. Librerías.

A continuación se incluyen en forma de tabla algunas de las funciones de librería más utilizadas en el lenguaje C.

Función	Tipo	Propósito	lib
abs(i)	int	Devuelve el valor absoluto de **i**	stdlib.h
acos(d)	double	Devuelve el arco coseno de **d**	math.h
asin(d)	double	Devuelve el arco seno de **d**	math.h
atan(d)	double	Devuelve el arco tangente de **d**	math.h
atof(s)	double	Convierte la cadena **s** en un número de doble precisión	stdlib.h
atoi(s)	long	Convierte la cadena **s** en un número entero	stdlib.h
clock()	long	Devuelve la hora del reloj del ordenador. Para pasar a segundos, dividir por la constante CLOCKS_PER_SEC	time.h
cos(d)	double	Devuelve el coseno de **d**	math.h
exit(u)	void	Cerrar todos los archivos y buffers, terminando el programa.	stdlib.h
exp(d)	double	Elevar **e** a la potencia **d** (e=2.77182...)	math.h
fabs(d)	double	Devuelve el valor absoluto de **d**	math.h
fclose(f)	int	Cierra el archivo **f**.	stdio.h
feof(f)	int	Determina si se ha encontrado un fin de archivo.	stdio.h
fgetc(f)	int	Leer un carácter del archivo **f**.	stdio.h
fgets(s,i,f)	char *	Leer una cadena **s**, con **i** caracteres, del archivo **f**	stdio.h
floor(d)	double	Devuelve un valor redondeado por defecto al entero más cercano a **d**.	math.h
fmod(d1,d2)	double	Devuelve el resto de **d1/d2** (con el mismo signo de **d1**)	math.h
fopen(s1,s2)	FILE *	Abre un archivo llamado **s1**, para una operación del tipo **s2**. Devuelve el puntero al archivo abierto.	stdio.h
fprintf(f,...)	int	Escribe datos en el archivo **f**.	stdio.h
fputc(c,f)	int	Escribe un carácter en el archivo **f**.	stdio.h
free(p)	void	Libera un bloque de memoria al que apunta **p**.	malloc.h

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

fscanf(f,...)	int	Lee datos del archivo **f**.	stdio.h
getc(f)	int	Ler un carácter del archivo **f**.	stdio.h
getchar()	int	Lee un carácter desde el dispositivo de entrada estándar.	stdio.h
log(d)	double	Devuelve el logaritmo natural de **d**.	
malloc(n)	void *	Reserva **n** bytes de memoria. Devuelve un puntero al principio del espacio reservado.	malloc.h o stdlib.h
pow(d1,d2)	double	Devuelve **d1** elevado a la potencia **d2**.	
printf(...)	int	Escribe datos en el dispositivo de salida estándar.	stdio.h
rand(void)	int	Devuelve un valor aleatorio positivo.	stdlib.h
sin(d)	double	Devuelve el seno de **d**.	math.h
sqrt(d)	double	Devuelve la raíz cuadrada de **d**.	math.h
strcmp(s1,s2)	int	Compara dos cadenas lexicográficamente.	string.h
strcomp(s1,s2)	int	Compara dos cadenas lexicográficamente, sin considerar mayúsculas o minúsculas.	string.h
strcpy(s1,s2)	char *	Copia la cadena **s2** en la cadena **s1**	string.h
strlen(s1)	int	Devuelve el número de caracteres en la cadena s.	string.h
system(s)	int	Pasa la orden s al sistema operativo.	stdlib.h
tan(d)	double	Devuelve la tangente de **d**.	math.h
time(p)	long int	Devuelve el número de segundos transcurridos desde de un tiempo base designado (1 de enero de 1970).	time.h
toupper(c)	int	convierte una letra a mayúscula.	stdlib.h o ctype.h

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

UNIDAD 3 Estructuras de selección, control y repetición.

El C++, como todo lenguaje de programación basado en la algorítmica, posee una serie de estructuras de control para gobernar el flujo de los programas.
Dentro de las estructuras de selección encontramos dos modelos en el C++, las de condición simple (sentencias if else) y las de condición múltiple (switch).
A continuación estudiaremos ambos tipos de sentencias.

3.1. Selección: if – else y else – If.

Se emplea para elegir en función de una condición. Su sintaxis es:

```
if (expresión)
        sentencia 1;
else
        sentencia 2;
```

Los paréntesis de la expresión a evaluar son obligatorios, la sentencia 1 puede ser una sola instrucción (que no necesita ir entre llaves) o un bloque de instrucciones (entre llaves). El else es opcional, cuando aparece determina las acciones a tomar si la expresión es falsa.
El único problema que puede surgir con estas sentencias es el anidamiento de if y else: Cada else se empareja con el if más cercano:

```
if (expr1)
if (expr2)
        acción 1
else // este else corresponde al if de expr 2
        acción 2
else // este corresponde al if de expr1
        acción 3
```

Para diferenciar bien unas expresiones de otras (el anidamiento), es recomendable tabular correctamente y hacer buen uso de las llaves:

```
if (expr1)
{
        if (expr2)
                acción 1
} // Notar que la llave no lleva punto y coma después, si lo pusiéramos
// habríamos terminado la sentencia y el else se quedaría suelto
else // Este else corresponde al if de expr1
        acción 3
```


Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Por último indicaremos que cuando anidamos else - if se suele escribir:

```
if (e1)
        a1
else if (e2)
        a2
else if (e3)
        a3
        …
else
        an
```

de esta manera evitamos el exceso de tabulación.

3.2. Selecciones anidadas.

```
CODIGO C
if  (expresion)

        if  (expresion)
          sentencia_v;
         else
       sentencia_f;
else
          if (expresion)
            sentencia_v;
           else
         sentencia_f ;
```

3.3. Sentencias de control: switch.

Si queremos ver varios posibles valores, seria muy pesado tener que hacerlo con muchos "if" seguidos o encadenados. La alternativa es la orden "switch", cuya sintaxis es:

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

```
switch (expresión)
{
        case valor_1: sentencia 11;
                sentencia 12;
                …
                sentencia 1n;
                break;
        case valor_2: sentencia 21;
                sentencia 22;
                …
                sentencia 2m;
                break;
        …
        default: sentencia d1;
                sentencia d2;
                …
                sentencia dp
}
```

Es decir, se escribe tras "switch" la expresión a analizar, entre paréntesis. Después, tras varias ordenes "case" se indica cada uno de los valores posibles. Los pasos (porque pueden ser varios) que se deben dar si se trata de ese valor se indican a continuación, terminando con "break". Si hay que hacer algo en caso de que no se cumpla ninguna de las condiciones, se detalla tras "default".

Ejemplo:

```
#include <stdio.h>
main()
{
        char tecla;
        printf("Pulse una tecla y luego Intro: ");
        scanf("%c", &tecla);
        switch (tecla)
        {
                case ' ': printf("Espacio.\n");
                        break;
                case '1':
                case '2':
                case '3':
                case '4':
                case '5':
                case '6':
                case '7':
                case '8':
                case '9':
                case '0': printf("Dígito.\n");
                        break;
                default: printf("Ni espacio ni dígito.\n");
        }
}
```

3.4. Estructuras de repetición: while, for y do while.

Es habitual que ciertas partes de un algoritmo deban repetirse varias veces con objeto de resolver un problema. La *repetición* es un concepto importante a la hora de describir algoritmos. Los lenguajes de programación disponen de una serie de sentencias que permiten repetir varias veces algunos segmentos del programa. A este tipo de sentencias se les denomina *sentencias iterativas.*

Estructura for

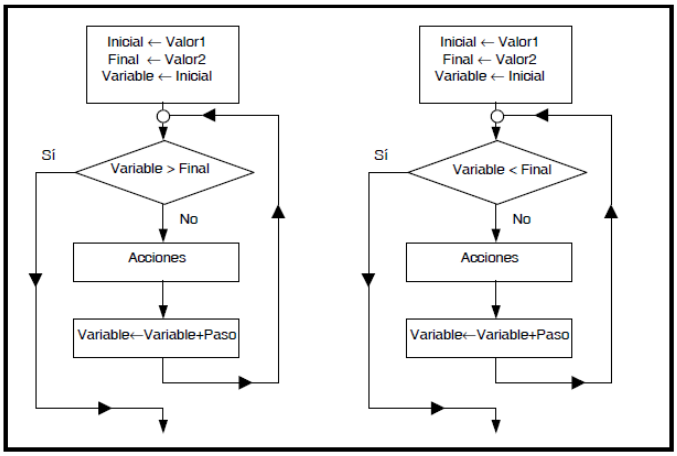
La sentencia FOR permite repetir, un número de veces conocido a priori, una serie de instrucciones. La sentencia FOR es el modo más adecuado de expresar bucles definidos.
Se puede especificar que la variable de control se incremente en un valor distinto a uno tras cada repetición de las acciones dentro del bucle.
Aunque en la mayoría de los lenguajes la sentencia `for` es una estructura de repetición con
contador, en C++ es muy similar a un `while` pero con características similares. Su sintaxis es:

```
for (expr1; expr2; expr3)
sentencia
```

Que es equivalente a:

```
expr1
while (expr2) {
sentencia
expr3
```

Es decir, la primera expresión se ejecuta una vez antes de entrar en el bucle, después se comprueba la verdad o falsedad de la segunda expresión y si es cierta ejecutamos la sentencia y luego la expresión 3.



Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Ejemplo:

```
#include <stdio.h>
main()
{
        int contador;
        for (contador=1;
        contador<=10;
        contador++)
        printf("%d ", contador);
}
```

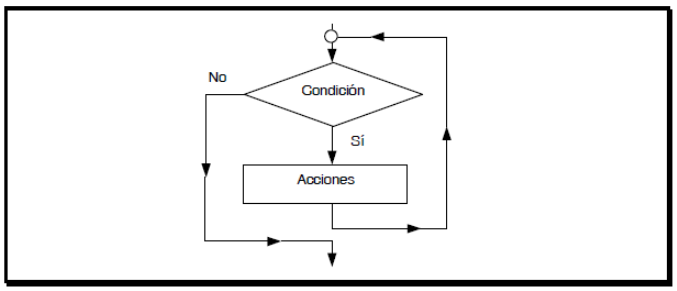
Recordemos que “contador++” es una forma abreviada de escribir “contador=contador+1”, de
modo que en este ejemplo aumentamos la variable de uno en uno.

Estructura While

Si queremos hacer que una sección de nuestro programa se repita mientras se cumpla una cierta condición, usaremos la orden “while”. Esta orden tiene dos formatos distintos, según comprobemos la condición al principio o al final.

La sentencia WHILE debe utilizarse exclusivamente para expresar bucles indefinidos. Es decir, la sentencia se repetira mientras la condicion sea cierta. Si la condicion es falsa ya desde un principio, la sentencia no se ejecuta nunca. Si queremos que se repita mas de una sentencia, basta agruparlas entre { y }.

```
while (expresión)
sentencia
```



Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Ejemplo:

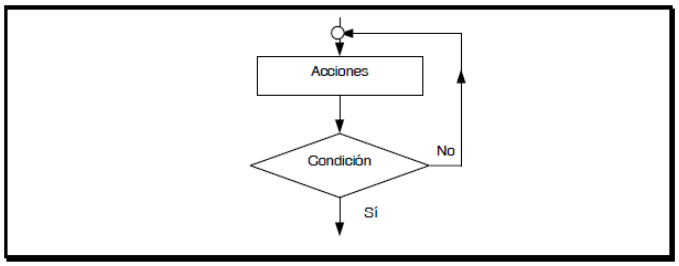
```
#include <stdio.h>
main()
{
        int numero;
        printf("Teclea un número (0 para salir): ");
        scanf("%d", &numero);
        while (numero!=0)
        {
                if (numero > 0) printf("Es positivo\n");
                else printf("Es negativo\n");
                        printf("Teclea otro número (0 para salir):
                ");
                        scanf("%d", &numero);
        }
}
```

En este ejemplo, si se introduce 0 la primera vez, la condición es falsa y ni siquiera se entra al bloque del "while", terminando el programa inmediatamente.
Nota: si recordamos que una condición falsa se evalúa como el valor 0 y una condición verdadera como una valor distinto de cero, veremos que ese "while (numero != 0)" se podría abreviar como "while (numero)".

Estructura do-while

Este es el otro formato que puede tener la orden "while": la condicion se comprueba al final.
El punto en que comienza a repetirse se indica con la orden "do":
La sentencia do-while permite repetir una serie de instrucciones hasta que cierta condición sea cierta. La sentencia do-while debe utilizarse exclusivamente para expresar bucles indefinidos.
El funcionamiento es simple, entramos en el do y ejecutamos la sentencia, evaluamos la expresión y si es cierta volvemos al do, si es falsa salimos.

```
do
        sentencia
while (expresión);
```



Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Ejemplo:

```
#include <stdio.h>
main()
{
        int valida = 711;
        int clave;
do
{
        printf("Introduzca su clave numérica: ");
        scanf("%d", &clave);
        if (clave != valida) printf("No válida!\n");
}
```

En este caso, se comprueba la condicion al final, de modo que se nos preguntara la clave al menos una vez. Mientras que la respuesta que demos no sea la correcta, se nos vuelve a preguntar. Finalmente, cuando tecleamos la clave correcta, el ordenador escribe "Aceptada" y termina el programa.

3.5. Break y continue.

El C++ pretende ser un lenguaje eficiente, por lo que nos da la posibilidad de romper la secuencia de los algoritmos de una forma rápida, sin necesidad de testear infinitas variables para salir de un bucle. Disponemos de varias sentencias de ruptura de secuencia que se listan a continuación.

La sentencia break

Es una sentencia muy útil, se emplea para salir de los bucles (do-while, while y for) o de un switch. De cualquier forma esta sentencia sólo sale del bucle o switch más interior, si tenemos varios bucles anidados sólo salimos de aquel que ejecuta el break.

La sentencia continue

Esta sentencia se emplea para saltar directamente a evaluar la condición de un bucle desde cualquier punto de su interior. Esto es útil cuando sabemos que después de una sentencia no vamos a hacer nada más en esa iteración.

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

3.6. Comparación de estructuras.

	Valores iníciales, operaciones previas	Cond. para seguir repitiendo	Cuando se comprueba	Cambios que hay que hacer	¿pueden quedar campos vacíos?
for	1º campo	2º campo	Antes de cada vuelta	3º campo	si
while	Hay que ponerlo fuera, antes del ciclo	En el while entre paréntesis	Antes de cada vuelta	Dentro del ciclo después de las instrucciones a repetir	no
do while	Hay que ponerlo fuera, antes del ciclo	En el while entre paréntesis	Después de cada vuelta	Dentro del ciclo después de las instrucciones a repetir	no

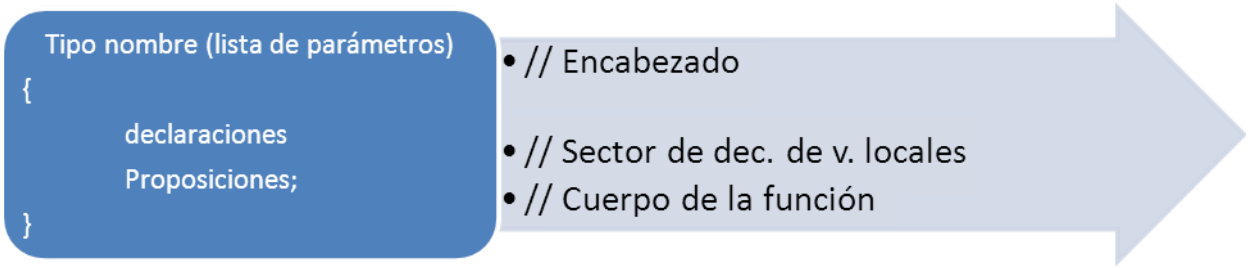
Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

UNIDAD 4 Funciones

4.1. Conceptos Básicos.

Cuando el programador diseñó la solución del problema, escribió módulos de propósitos específicos. Al codificar esos módulos en código C, está escribiendo sus propias funciones. En código C, los módulos se convierten en funciones.
Las funciones constituyen una parte aislada y autónoma del programa, que pueden estar dentro o fuera del código fuente del programa que las invoca. Las proposiciones o sentencias que definen la función se escriben una sola vez y se guardan de manera apropiada. Con funciones bien diseñadas es posible ignorar "cómo" lleva a cabo su trabajo, es suficiente saber "qué hace". Frecuentemente se ven funciones cortas, definidas y empleadas una sola vez, esto es porque el uso de funciones esclarece alguna parte del código.
Las funciones podrán tener o no una lista de parámetros Las funciones se invocan (o llaman o usan) mediante una **llamada de función.** Una llamada de función puede estar en cualquier parte de un programa.

4.2. Declaración de funciones: Formato general.

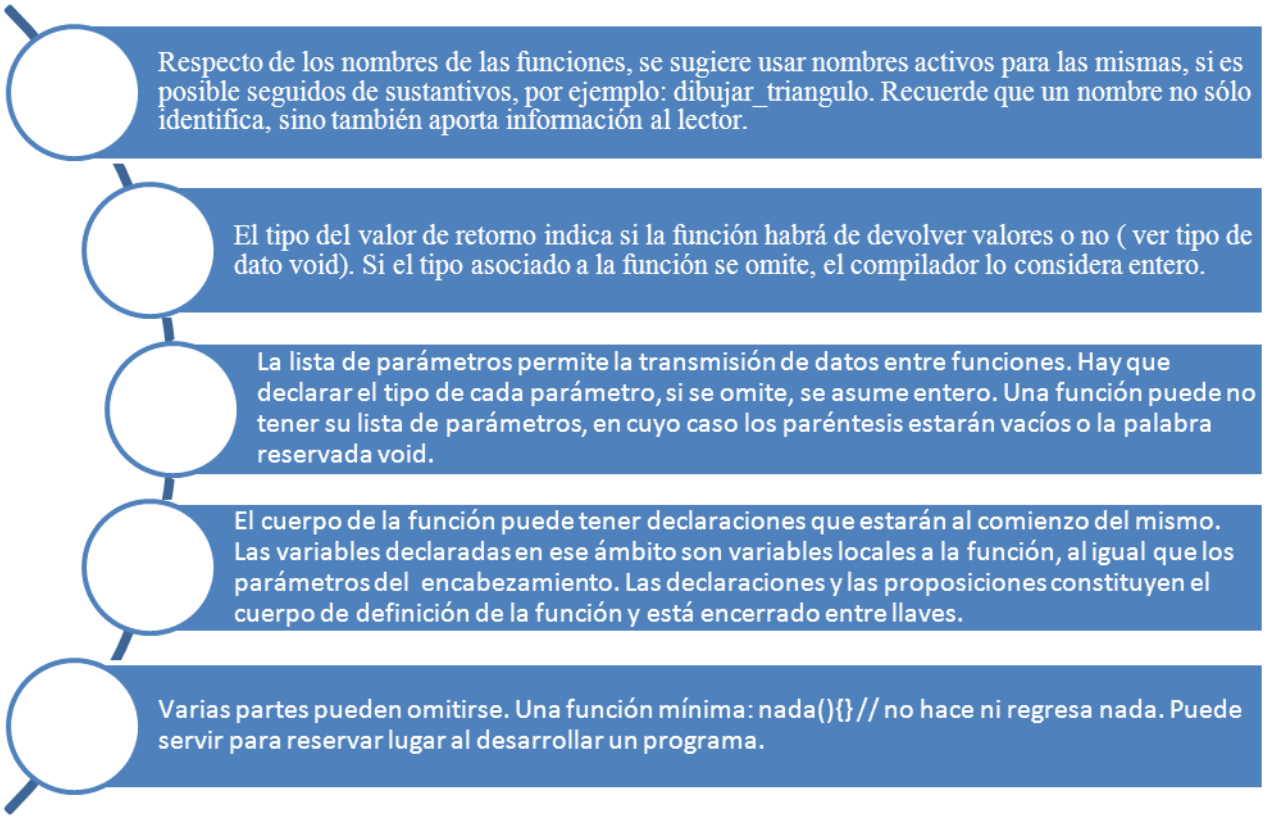


// Encabezado
Tipo: es el tipo asociado a la función y está ligado al valor de retorno de la misma.
nombre: es el nombre que identifica a la función. Es un identificador válido.
lista de parámetros: listado de los parámetros formales de la función, de la forma:
tipo1 param1, tipo2 param2… Donde ***tipoi*** es el tipo de dato asociado al parámetro.

// Sector de declaración de variables locales
De la forma: ***tipo1 var1; tipo2 var2; … tipoi vari;*** Donde ***tipoi*** es el tipo de dato asociado a la variable.
Es un listado de la variables internas o locales a la función.

// Cuerpo de la función
Formado por proposiciones o sentencias válidas.

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5



Las funciones:

- se declaran
- se definen
- se asocian a un tipo de datos
- se invocan
- se ejecutan
- se pueden usar como factores de una expresión
- pueden devolver un valor/valores
- pueden ejecutar tareas sin retornar ningún valor
- pueden llamarse desde distintas partes de un mismo programa

Un ejemplo simple: Imagine la siguiente situación: un usuario escribe un programa que realiza ciertas tareas y hay un cálculo que se repite a lo largo del mismo, es el cuadrado de un número. Para simplificar y esclarecer sectores del código, el programador escribe una apropiada. A continuación, se transcriben sectores del código en cuestión.

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

```
:
:
int calcular_cuadrado(int dato);
void main ()
{
        int dat1, dat2,resul1, resul2, i;
        :
        scanf("%d %d",&dat1,&dat2);
        resul1 = 25 +
        calcular_cuadrado(dat1); /* la
        función es usada como factor de una
        expresión */
        :
        :
        for (i =1; i<= 10; i++)
                printf("El cuadrado del
        número %d es : %d
        \n",i,calcular_cuadrado(i));
        :
}
int calcular_cuadrado(int dato)
{
        int aux;
        aux = dato*dato;
        return(aux);
 }
```

Una versión alternativa de la función:

```
int calcular_cuadrado(int dato)
{
        return(dato*dato);
 }
```


Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

4.3. Parámetros de una función.

Las palabras parámetro y argumento, aunque de significado similar, tiene distintas connotaciones semánticas: Se denominan parámetros los tipos declarados en el prototipo (que deben corresponder con los declarados en la definición). Cuando se realiza una llamada a la función, los "valores" pasados se denominan argumentos. A veces se utilizan también las expresiones argumentos formales, para los parámetros y argumentos actuales para los valores pasados.

Parámetros (en prototipo o definición)	• argumentos formales
Valores pasados (en tiempo de ejecución)	• argumentos actuales

La sintaxis utilizada para la declaración de la lista de parámetros formales es similar a la utilizada en la declaración de cualquier identificador.

Ejemplos:

int func(void) {...}	•sin parámetros
inf func() {...}	•idem
int func(T1 t1, T2 t2, T3 t3=1) {...}	•tres parámetros simples, uno con argumento por defecto
int func(T1* ptr1, T2& tref) {...}	•los argumentos son un puntero y una referencia.
int func(register int i) {...}	•Petición de uso de registro para argumento (entero).
int func(char* str,...) {...}	•Una cadena y cierto número de otros argumentos, o un número fijo de argumentos de tipos variables.

El tipo void está permitido como único parámetro formal. Significa que la función no recibe ningún argumento.

4.4 Funciones que devuelven valores.

Paso por valor:

Cuando se pasa un parámetro por valor a una función, (ver ejemplo de la función que suma), la función hace copias de las variables y utiliza las copias para hacer las operaciones. **No se alteran los valores originales**, ya que cualquier cambio ocurre sobre las copias que desaparecen al terminar la función.

Paso por referencia:

Cuando el objetivo de la función es **modificar el contenido de la variable pasada como parámetro**, debe conocer la dirección de memoria de la misma. Es por eso que, por ejemplo, la función scanf() necesita que se le anteponga a la variable el operador &, puesto que se le está pasando la dirección de memoria de la variable, ya que el objetivo de scanf() es guardar allí un valor ingresado por teclado.

El siguiente programa tiene una función que intercambia los valores de dos variables de tipo char.

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

```
#include <stdio.h>
#include <conio.h>
void cambia(char* x, char* y); //prototipo
void main(void)
{
        char a, b;
        a='@';
        b='#';
        clrscr();
        printf("\n**** Antes de la función ****\n");
        printf("Contenido de a = %c\n", a);
        printf("Contenido de b = %c\n", b);
        cambia(&a,&b); (*)
        printf("\n**** Después de la función ****\n");
        printf("Contenido de a = %c\n", a);
        printf("Contenido de b = %c\n", b);
        getch();
}

void cambia(char* x, char*y) (**)
{
        char aux;
        aux=*x;
        *x=*y;
        *y=aux;
}
```

En la línea (*) se llama a la función cambia() pasándole las direcciones de memoria de las variables, puesto que precisamente el objetivo es modificar los contenidos.
La función en **(**), recibe los parámetros como punteros**, puesto que son los únicos capaces de entender direcciones de memoria como tales. Dentro de la función tenemos entonces x e y que son punteros que apuntan a la variable a y a la variable b; utilizando luego el operador * sobre los punteros hacemos el intercambio.

Características

- En este tipo de llamadas los argumentos contienen direcciones de variables.
- Dentro de la función la dirección se utiliza para acceder al argumento real.
- En las llamadas por referencia cualquier cambio en la función tiene efecto sobre la variable cuya dirección se pasó en el argumento. No hay un proceso de creación/destrucción de esa dirección.

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

- Aunque en C todas las llamadas a funciones se hacen por valor, pueden simularse llamadas por referencia utilizando los operadores **&** (dirección) y *** (en la dirección).**
- Mediante **& podemos pasar direcciones de variables en lugar de** valores, y trabajar internamente en la función con los contenidos, mediante el operador ***.**

4.5 Funciones void.

Existe un tipo particular de funciones: el tipo *void*. Se usa en los casos en que la función no devuelve ningún valor. Se dice entonces que la función produce un efecto, por ejemplo, dibujar una figura, saltar líneas de impresión, imprimir una constancia, etc., que lleva a cabo un procedimiento.
La forma de definición de una función de este tipo responde a la forma general indicada previamente. Pueden ser invocadas en cualquier parte del programa, para ello, basta escribir el nombre de la misma, seguida de los argumentos indicados. Invocar una función *void* es similar a escribir una proposición en código C. Por último, no se puede poner la función en una expresión donde se requiera un valor.
Para obtener una figura como la siguiente, un código posible es:

```
1
12
123
1234
12345
1
12
123
1234
12345
1
12
123
1234
12345
1
1
```

```
#include <stdio.h>
#include <stdlib.h>
void dibujar(int filas);
main()
{
        int n,fil,i;
        printf("Ingrese cuantos niveles del árbol quiere\n");
        scanf("%d",&n);
        printf("Ingrese cuantos filas quiere en cada nivel\n");
        scanf("%d",&fil);
        for(i=1;i<=n;i++) /* aquí dibuja el árbol */
                dibujar(fil);
        for(i=1;i<=n-n/2;i++) /* aquí dibuja la base */
                dibujar(1);
        return 0;
}
void dibujar(int filas)
{
        int exte, num;
        for(exte=1;exte<=filas;exte++)
        {
                for(num=1;num<=exte;num++)
                printf("%d",num);
                printf("\n");
        }
}
```


Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

4.6. Ámbito de variables y funciones.

Alcance de un identificador

El alcance de un nombre es la parte del programa dentro de la cual se puede usar el nombre. Para una variable declarada al principio de una función, el alcance es la función dentro de la cual está declarado el nombre.

Variables Locales o Automáticas

Las variables declaradas en el ámbito de una función son privadas o locales a esa función. Ninguna otra función puede tener acceso directo a ellas, se puede pensar que los otros módulos o funciones "no las pueden ver" y por lo tanto no las pueden modificar.

El ciclo de vida de las variables locales es muy breve: comienzan a "existir" cuando la función se activa como efecto de una llamada o invocación y "desaparecen "cuando la ejecución de la función termina. Por esta razón estas variables son conocidas como variables automáticas. Las variables locales con el mismo nombre que estén en funciones diferentes no tienen relación.

Debido a que estas variables locales "aparecen y desaparecen" con la invocación de funciones, no retienen sus valores entre dos llamadas sucesivas, por lo cual deben ser inicializadas explícitamente en cada entrada, sino, contendrán "basura informática".

Variables Globales o Automáticas

En oposición a las variables locales, existen las variables externas o públicas o globales. Estas son variables que "no pertenecen" a ninguna función, ni siquiera a *main* y que pueden ser accedidas por todas las funciones.

Algunos autores se refieren a ellas como a variables públicas. Se declaran en forma externa al *main*.

Puesto que ellas son de carácter público (desde el punto de vista del acceso) pueden ser usadas en lugar de la lista de parámetros, para comunicar datos entre funciones, lo cual no garantiza que sus valores no se modifiquen. Para evitar esta situación, se puede trabajar con parámetros, los cuales son copiados por la función. A diferencia de las variables locales, ellas tienen un ciclo de vida importante, existen mientras dura la ejecución del programa completo.

4.7. Funciones de biblioteca.

Todas las versiones del lenguaje C ofrecen con una biblioteca estándar de funciones en tiempo de ejecución que proporciona soporte para operaciones utilizadas con más frecuencia. Estas funciones permiten realizar una operación con sólo una llamada a la función (sin necesidad de escribir su código fuente).

Las *funciones estándar o predefinida, como así se denominan las funciones pertenecientes a la* biblioteca estándar, se dividen en grupos; todas las funciones que pertenecen al mismo grupo se declaran en el mismo *archivo* de cabecera.

En los módulos de programa se pueden incluir líneas #include con los archivos de cabecera correspondientes en cualquier orden, y estas líneas pueden aparecer más de una vez.

Para utilizar una función o un macro, se debe conocer su número de argumentos, sus tipos y el tipo de sus valores de retorno. Esta información se proporcionará en los prototipos de la función. La sentencia #include mezcla el archivo de cabecera en su programa.

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Algunos de los grupos de funciones de biblioteca más usuales son:

- *E/S estándar (para operaciones de EntraddSalida);*
- *matemáticas (para operaciones matemáticas);*
- *rutinas estándar (para operaciones estándar de programas);*
- *visualizar ventana de texto;*
- *de conversión (rutinas de conversión de caracteres y cadenas);*
- *de diagnóstico (proporcionan rutinas de depuración incorporada);*
- *de manipulación de memoria;*
- *control del proceso;*
- *clasificación (ordenación);*
- *directorios;*
- *fecha y hora;*
- *de interfaz;*
- *diversas;*
- *búsqueda;*
- *manipulación de cadenas;*
- *gráficos.*

Se pueden incluir tantos archivos de cabecera como sean necesarios en sus archivos de programa.

Funciones de carácter

El archivo de cabecera <ctype.h > define un grupo de funciones/macros de manipulación de caracteres.

Todas las funciones devuelven un resultado de valor verdadero (distinto de cero) o falso (cero).

Función	Prueba (test) de
`int isalpha(int c)`	Letra mayúscula o minúscula.
`int isdigit(int c)`	Dígito decimal.
`int isupper(int c)`	Letra mayúscula (A-Z).
`int islower(int c)`	Letra minúscula (a-z).
`int isalnum(int c)`	`letra o dígito; isalpha(c)||isdigit(c)`
`int iscntrl(int c)`	Carácter de control.
`int isxdigit(int c)`	Dígito hexadecimal.
`int isprint(int c)`	Carácter imprimible incluyendo ESPACIO.
`int isgraph(int c)`	Carácter imprimible excepto ESPACIO.
`int isspace(int c)`	ESPACIO, AVANCE DE PÁGINA, NUEVA LÍNEA, RETORNO DE CARRO, TABULACIÓN, TABULACIÓN VERTICAL.
`int ispunct(int c)`	Carácter imprimible no espacio, dígito o letra.
`int toupper(int c)`	Convierte a letras mayúsculas.
`int tolower(int c)`	Convierte a letras minúsculas.

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Comprobación alfabética y de dígitos	Funciones de prueba de caracteres especiales	Funciones de conversión de caracteres
isalpha(c)Devuelve verdadero (distinto de cero) si c es una letra mayúscula o minúscula. Se devuelve un valor falso si se pasa un carácter distinto de letra a esta función. ***islower(c)***Devuelve verdadero (distinto de cero) si c es una letra minúscula. Se devuelve un valor falso (0),si se pasa un carácter distinto de una minúscula. ***isupper (c)*** Devuelve verdadero (distinto de cero) si c es una letra mayúscula, falso con cualquier otro carácter. **isdigit(c)**Comprueba si c es un dígito de O a 9, devolviendo verdadero (distinto de cero) en ese caso, y falso en caso contrario. Devuelve verdadero si c es cualquier dígito hexadecimal (0 a 9, A a F, o bien a a f) y falso en cualquier otro caso. ***isxdigit (c)***Las siguientes funciones comprueban argumentos numéricos o alfabéticos: ***isalnum(c)***Devuelve un valor verdadero, si c es un dígito de O a 9 o un carácter alfabético (bien mayúscula o minúscula) y falso en cualquier otro caso.	**iscntrl(c)**Devuelve verdadero si c es un *carácter de control (códigos ASCII O a 31) y falso en caso* contrario. **isgraph(c)**Devuelve verdadero si c es un carácter imprimible (no de control) excepto espacio; en caso contrario, se devuelve falso. **isprint(c)**Devuelve verdadero si c es un carácter imprimible (código ASCII 32 a 127) incluyendo un espacio; en caso contrario, se devuelve falso. **ispunct(c)**Devuelve verdadero si c es cualquier carácter de puntuación (un carácter imprimible distinto de espacio, letra o dígito); falso, en caso contrario. **isspace(c)**Devuelve verdadero si c es carácter un espacio, nueva línea (\n) , retorno de carro (\r), tabulación (\ t) o tabulación vertical (\ v).	**tolower (c)** Convierte el carácter c a minúscula, si ya no lo es. **toupper(c)** Convierte el carácter c a mayúscula, si ya no lo es.

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Ejemplos

Leer un carácter del teclado y comprobar si es una letra.

```
#include <stdio.h>
#include <ctype.h>
int main0
{
        // Solicita iniciales y comprueba que es alfabética
        char inicial;
        printf("¿Cuál es su primer carácter inicial?: " ) ;
        scanf ( "%c" , &inicial) ;
        while (!isalpha(inicial))
        {
                puts ("Carácter no alfabético 'I) ;
                printf ("¿Cuál es su siguiente inicial?: " )
;
                scanf ("%c"&,i nicial) ;
        }
        puts (";Terminado!";)
        return 0;
}
```

*Comprueba **si la entrada es una v o una H.***

```
#include <stdio.h>
#include <ctype.h>
int main()
{
        char resp; / * respuesta del usuario * /
        char c;
        printf ("¿Es un varón o una hembra (V/H)?: " ) ;
        scanf ("%c",&r esp) ;
        resp=toupper(resp);
        switch (resp)
        {
                case ‘V’:
                        puts ("Es un enfermero") ;
                        break;
                case 'H’:
                        puts ("Es una maestra") ;
                        break;
                default:
                        puts("No es ni enfermero ni maestra") ;
                        break;
        }
        Return 0;
}
```


Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Funciones numéricas

Virtualmente cualquier operación aritmética es posible en un programa C. Las funciones matemáticas disponibles son las siguientes:

- matemáticas
- *trigonométricas;*
- *Logarítmicas y exponenciales;*
- *aleatorias.*

La mayoría de las funciones numéricas están en el archivo de cabecera MATH. H; las funciones **abs** y **labs están definidas en MATH. H y STDLIB. H, y las rutinas div y ldiv en STDLIB. H.**

Funciones matemáticas	Funciones trigonométricas	Funciones logarítmicas y exponenciales	Funciones aleatorias
ceil(x) Redondea al entero más cercano. **fabs(x)** Devuelve el valor absoluto de x (un valor positivo). **floor(x)** Redondea por defecto al entero más próximo. **fmod(x, y)** Calcula el resto f en coma flotante para la división x/v. **pow(x,)**Calcula x elevado a la potencia y (x'). **pow10(x)**Calcula 10 elevado a la potencia x (IO); x debe ser de tipo entero. **Sqrt(x)** Devuelve la raíz cuadrada de x; x debe ser mayor *o igual a cero.*	**acos(x)** Calcula el arco coseno del argumento x. El argumento x debe estar entre -1 y 1. ***asin(x)*** Calcula el arco seno del argumento x. El argumento x debe estar entre -1 y 1 ***atan(x)*** Calcula el arco tangente del argumento x. **atan2(x,y)** Calcula el arco tangente de x dividido por y. ***cos(x)*** Calcula el coseno del ángulo x ; x se expresa en radianes. ***sin(x)*** Calcula el seno del ángulo x; x se expresa en radianes. **tan(x)**Devuelve la tangente del ángulo x ; x se expresa en radianes.	**exp(x1, expl(x)** Calcula el exponencial e, donde e es la base de logaritmos naturales de valor 2.7 18282. expl, calcula e utilizando un valor long double **log(x), logl(x)** La función log calcula el logaritmo natural del argumento x y log1 (x) calcula el citado logaritmo natural del argumento x de valor lonq double. **loglO(x), loglOl(x)** Calcula el logaritmo decimal del argumento x, de valor real double en 1 ogl 0 (X I y de valor real long double en log] O1 (x) ; x ha de ser positivo.	**rand(void)** La función rand genera un número aleatorio. El número calculado por rand varía en el rango entero de O a RAND-MAX. La constante RAND-MAX se define en el archivo STDLIB.H **randomize(void)** La macro randomize inicializa el generador de números aleatorios con una semilla aleatoria obtenida a partir de una llamada a la función time

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Ejemplo

```
/ *  Programa para generar 10 números aleatorios * /
#include <stdio.h>
#include <stdlib.h>
#include <time. h>
#include <conio.h>
fnt main (void)
{
        int i;
        clrscro; / * limpia la pantalla* /
        randomize ( ) ;
        for (i=l; i<=10; i++)
                printf ("8d 'I, rand ( ) ) ;
        return 0;
}
```

```
/* Ejemplo de generación de números aleatorios, se fija la semilla en 50 y se genera un número aleatorio. */
#include <stdio.h>
#include <stdlib.hi
#include <conio.h>
int main (void)
{
        clrscr ( ) ;
        srand(50);
        printf("Este es un número aleatorio: %d",rand 0 ) ;
        return 0;
}
```

Funciones de fecha y hora

La estructura de tiempo utilizada incluye los miembros siguientes:

```
struct tm
{
        int tm-sec; / * segundos * /
        int tm-min; / * minutos * /
        int tm-hour; / * horas * /
        int tm-mday; / * día del mes 1 a 31 * /
        int tmmon; / * mes, 0 para Ene, 1 para Feb, . . . * /
        int tmjear; / * año desde 1900 * /
        int tm-wday; / * días de la semana desde domingo (0-6) */
        int tmjday; / * día del año desde el 1 de Ene(0-365) * /
        int tm-isdt; / * siempre 0 para gmtime * /
} ;
```

clock (void) La función clock determina el tiempo de procesador, en unidades de click, transcurrido desde el principio de la ejecución del programa. Si no se puede devolver el tiempo de procesador se devuelve -I.

*time (hora)*La función time obtiene la hora actual

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

localtime(hora) Convierte la fecha y hora en una estructura de tipo tm .
mktime(t) Convierte la fecha en formato de calendario.

Una aplicación de clock ( ) para determinar el tiempo de proceso de un programa que calcula el factorial de un número.
El factorial de n! = n*(n-I)*(n-2).. . 2*1. La variable que vaya a calcular el factorhi, se define de tipo long para poder contener un valor elevado. El número, arbitrariamente, va a estar comprendido entre 3 y 15. El tiempo de proceso va a incluir el tiempo de entrada de datos. La función clock ( ) devuelve el tiempo en unidades de click, cada CLK-TCK es un segundo. El programa escribe el tiempo en ambas unidades.

```
#include <time.h>
#include <stdio.h>
int main (void)
{
        printf ( " Ffloat inicio, fin;
        int n, x;
        long int fact;
        inicio = clock() ;
        do {
                printf ( " Factorial de (3 <x< 15) : " ) ;
                scanf ("%d"h,x );
        }while (x<=3 / I x>=15);
        for (n=x,fact=l; x; x--1
                fact *=x;
        fin = clock() ;
        printf ("\nF actorial de %d! = %ld",n,f act) ;
        printf("\n Unidades de tiempo de proceso: %f,\t En segundos: %f‘,(fin-inicio),
        (fin-inicio) /CLK-CK);
        return 0;
}
```

Funciones de utilidad

C incluyen una serie de funciones de utilidad que se encuentran en el archivo de cabecera STDLIB. H
y que se listan a continuación.
abs(n), labs(n)
int abs (int n)
long labs(1ong n)
devuelven el valor absoluto de *n.*
div(num, denom)
div-t div(int num, i n t denom)
Calcula el cociente y el resto de num, dividido por denom y almacena el resultado en quot y rem,
miembros int de la estructura div-t.
typedef struct
{
        int quot; / * cociente * /
        int rem; / * resto * /

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

} div-t;
El siguiente ejemplo calcula y visualiza el cociente y el resto de la división de dos enteros.

```
#include <stdlib.h>
#include <stdio.h>
int main (void)
{
        div-t resultado;
        resultado = div(l6, 4);
        printf ( "Cociente %d" , resultado. quot ) ;
        printf ("Resto ad", resultado. rem) ;
        return  0;
}
```

4.8 Funciones recursivas

Las funciones escritas en código C pueden ser recursivas, esto es, pueden llamarse a sí mismas. El código recursivo es compacto y más fácil de escribir y de entender que su equivalente no recursivo. La recursividad es un recurso del lenguaje muy apto para trabajar con estructuras de datos complejas definidas en forma recursiva.
Un ejemplo típico es el cálculo del factorial de un número, definido en la forma:
N! = N * (N-1)! = N * (N-1) (N-2)! = N * (N-1)*(N-2)*...*2*1

La función factorial, escrita de forma recursiva, sería como sigue:

```
unsigned long factorial(unsigned long
numero)
{
        if ( numero == 1 || numero == 0 )
                return 1;
        else
        return numero*factorial(numero-1);
}
```

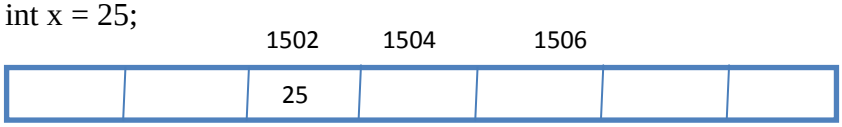
Por lo general la recursividad no ahorra memoria. Tampoco es más rápida, sino más bien todo lo contrario, pero el código recursivo es más compacto y a menudo más sencillo de escribir y comprender.

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

UNIDAD 5 Punteros y Arreglos.

5.1. Punteros: concepto.

¿Qué es un PUNTERO?:
Un puntero es un objeto que apunta a otro objeto. Es decir, una variable cuyo valor es la dirección de memoria de otra variable.
No hay que confundir una dirección de memoria con el contenido de esa dirección de memoria.
int x = 25;



La dirección de la variable x (&x) es 1502. El contenido de la variable x es 25.
Las direcciones de memoria dependen de la arquitectura del ordenador y de la gestión que el sistema operativo haga de ella.
En lenguaje ensamblador se debe indicar numéricamente la posición física de memoria en que queremos almacenar un dato. De ahí que este lenguaje dependa tanto de la máquina en la que se aplique.
En C no debemos, ni podemos, indicar numéricamente la dirección de memoria, si no que utilizamos una etiqueta que conocemos como variable (en su día definimos las variables como direcciones de memoria). Lo que nos interesa es almacenar un dato, y no la localización exacta de ese dato en memoria.

5.2. Declaración de punteros: inicialización.

Una variable puntero se declara como todas las variables.
Debe ser del mismo tipo que la variable apuntada. Su identificador va precedido de un asterisco (*):
int *punt;
Es una variable puntero que apunta a variable que contiene un dato de tipo entero llamada punt.
char *car:
Es un puntero a variable de tipo carácter.
long float *num;
float *mat[5]; // . . .

Es decir: hay tantos tipos de punteros como tipos de datos, aunque también pueden declararse punteros a estructuras más complejas (funciones, struct, ficheros...) e incluso punteros vacíos (void) y punteros nulos (NULL).

Declaración de variables puntero: Sea un fragmento de programa en C:

```
char dato; //variable que almacenará un carácter.
char *punt; //declaración de puntero a carácter.
```


Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

punt = &dato; //en la variable punt guardamos la dirección de memoria de la variable dato; punt apunta a dato. Ambas son del mismo tipo, char.

int *punt = NULL, var = 14;
punt = &var;
printf("%#X, %#X", punt, &var) //la misma salida: dirección
printf("\n%d, %d", *punt, var); //salida: 14, 14

Hay que tener cuidado con las direcciones apuntadas:
printf("%d, %d", *(punt+1), var+1);
*(punt + 1) repesenta el valor contenida en la dirección de memoria aumentada en una posición (int=2bytes), que será un valor no deseado. Sin embargo var+1 representa el valor 15.
punt + 1 representa lo mismo que &var + 1 (avance en la dirección de memoria de var).
Al trabajar con punteros se emplean dos operadores específicos:

Operador de dirección: & Representa la dirección de memoria de la variable que le sigue:
&fnum representa la dirección de fnum.

Operador de contenido o indirección: *

El operador * aplicado al nombre de un puntero indica el valor de la variable apuntada:
float altura = 26.92, *apunta;
apunta = &altura; //inicialización del puntero
float altura = 26.92, *apunta;
apunta = &altura; //inicialización del puntero
.printf("\n%f", altura); //salida 26.92
.printf("\n%f", *apunta);

No se debe confundir el operador * en la declaración del puntero:
int *p;
Con el operador * en las instrucciones: .
*p = 27;
printf("\nContenido = %d", *p);

Veamos con un ejemplo en C la diferencia entre todos estos conceptos

Es decir: int x = 25, *pint;
pint = &x;

La variable pint contiene la dirección de memoria de la variable x. La expresión: *pint representa el valor de la variable (x) apuntada, es decir 25. La variable pint también tiene su propia dirección: &pint

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Otro ejemplo

```
void main(void)
{
        int a, b, c, *p1, *p2;
        void *p;
        p1 = &a; // Paso 1. La dirección de a es asignada a p1
        *p1 = 1; // Paso 2. p1 (a) es igual a 1. Equivale a a = 1;
        p2 = &b; // Paso 3. La dirección de b es asignada a p2
        *p2 = 2; // Paso 4. p2 (b) es igual a 2. Equivale a b = 2;
        p1 = p2; // Paso 5. El valor del p1 = p2
        *p1 = 0; // Paso 6. b = 0
        p2 = &c; // Paso 7. La dirección de c es asignada a p2
        *p2 = 3; // Paso 8. c = 3
        printf("%d %d %d\n", a, b, c); // Paso 9. ¿Qué se imprime?
        p = &p1; // Paso 10. p contiene la dirección de p1
        *p = p2; // Paso 11. p1= p2;
        *p1 = 1; // Paso 12. c = 1
        printf("%d %d %d\n", a, b, c); // Paso 13. ¿Qué se imprime?
}
```

Vamos a hacer un seguimiento de las direcciones de memoria y de los valores de las variables en cada paso. Suponemos que la variable a es colocada en la dirección 0000, b en la siguiente, es decir 0002, con un offset de 2 bytes, por ser valores integer.
Se trata de un sistema de posiciones relativas de memoria. Se verá en aritmética de punteros.
Se obtiene el siguiente cuadro. En él reflejamos las direcciones relativas de memoria y los cambios en cada uno de los pasos marcados:

Paso	a 0000	b 0002	c 0004	p1 0006	p2 0008	p 0010
1				*0000*		
2	*1*			0000		
3	1			0000	*0002*	
4	1	*2*		0000	0002	
5	1	2		*0002*	0000	
6	1	*0*		0002	0002	
7	1	0		0002	*0004*	
8	1	0	*3*	0002	0004	
9	1	0	3	0002	0004	
10	1	0	3	0002	0004	*0006*
11	1	0	3	*0004*	0004	0006
12	1	0	*1*	0004	0004	0006
13	1	0	1	0004	0004	0006

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Inicialización de punteros:
< Almacenamiento > < Tipo > * < Nombre > = < Expresión >

Si <Almacenamiento> es extern o static, <Expresion> deberá ser una expresión constante del tipo <Tipo> expresado.

Si <Almacenamiento> es auto, entonces <Expresion> puede ser cualquier expresión del <Tipo> especificado.

Ejemplos:

La constante entera 0, NULL (cero) proporciona un puntero nulo a cualquier tipo de dato:

```
int *p;
p = NULL; //actualización
```

5.3. Aritmética de punteros.

Incremento y decremento:

Los operadores ++ y -- actúan de modo diferente según el tipo apuntado por el puntero.
Si p es un puntero a caracteres (char *p) la operación p++ incrementa el valor de p en 1.
Si embargo, si p es un puntero a enteros (int *p), la misma operación p++ incrementa el valor de p en 2 para que apunte al siguiente elemento, pues el tipo int ocupa dos bytes.
Del mismo modo, para el tipo float la operación p++ incrementa el valor de p en 4.
Lo dicho para el operador ++ se cumple exactamente igual, pero decrementando, para el operador --.

Comparación:

Pueden compararse punteros del mismo modo que cualquier otra variable, teniendo siempre presente que se comparan direcciones y NO contenidos.

```
int *p, *q;
if (p == q) puts ("p y q apuntan a la misma posición de memoria");
```

Suma y resta

Ocurre exactamente lo mismo que con las operaciones de incremento y decremento. Si p es un puntero, la operación

```
p = p + 5;
```

Hace que p apunte 5 elementos más allá del actual.

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Si p estaba definido como un puntero a caracteres, se incrementará su valor en 5, pero si estaba definido como un puntero a enteros, se incrementará en 10.

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

5.4. Indirección de punteros: los punteros void y NULL.

Cuando se asigna 0 a un puntero, este no apunta a ningún objeto o función.
La constante simbólica NULL definida en stdio.h tiene el valor 0 y representa el puntero nulo.
Es una buena técnica de programación asegurarse de que todos los punteros toman el valor NULL cuando no apuntan a ningún objeto o función.

int *p = NULL;

Para ver si un puntero no apunta a ningún objeto o función:

```
if (p == NULL)
        printf("El puntero es nulo\n");
else
        printf("El contenido de *p es\n", *p);
```

5.5. Punteros y verificación de tipos.

Los punteros se enlazan a tipos de datos específicos, de modo que C verificará si se asigna la dirección de un tipo de dato al tipo correcto de puntero. Así, por ejemplo, si se define un puntero a float, no se le puede asignar la dirección de un carácter o un entero. Por ejemplo, este segmento de código no funcionará:

```
float *fp;
char c;
fp = &c; / * no es válido * /
```

C no permite la asignación de la dirección de c a fp, ya que fp es una variable puntero que apunta a datos de tipo real, float.
C requiere que las variables puntero direccionen realmente variables del mismo tipo de dato que está ligado a los punteros en sus declaraciones.

5.6. Arreglos: Características de los arreglos, declaraciones, almacenamiento en memoria.

Un arreglo, también llamado (array), es un conjunto de elementos dispuestos secuencialmente, que contienen datos del mismo tipo. El tamaño de un arreglo está dado por su número de elementos. El tamaño se debe fijar cuando se declara el arreglo. El tamaño y el tipo de datos del arreglo definen el espacio de memoria que ocupará dicho arreglo. Estos arreglos pueden ser de uno o varias dimensiones. No existe límite, salvo la cantidad de memoria existente en el computador.

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Índice de un arreglo

Todo arreglo esta compuesto por un número de elementos. El índice es un numero correlativo que indica la posición de un elemento del arreglo. Los indices en C++ van desde la posición 0 hasta la posición tamaño – 1.

Elemento de un arreglo

Un elemento de un arreglo es un valor particular dentro de la estructura del arreglo. Para acceder a un elemento del arreglo es necesario indicar la posición o índice dentro del arreglo. Ejemplo:

• arreglo[0] //Primer elemento del arreglo
arreglo[3] //Cuarto elemento del arreglo

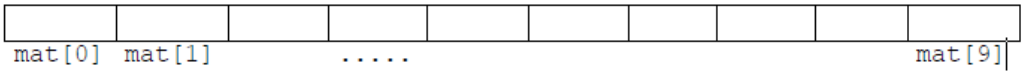
Declaración de un arreglo o matriz

Su formato es el siguiente:
Declaración de arreglo unidimensional:
tipo ident_arreglo[num_elementos];

Ejemplo:

```
void()
{
        int mat[10];
...
}
```

En este caso se ha definido un arreglo de 10 elementos enteros. Los que ocupan 20 bytes de memoria. El acceso de los elementos se hace desde el elemento 0 al elemento 9, es decir:



Acceso a los elementos de una matriz

Pueden ser accesado en forma individual o por medio de un índice, el cual debe ser entero. Para inicializar un elemento en particular:

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Ejemplo de diferentes formas de acceder a un arreglo.

Ejemplo 1:

```
void main()
{
        int matriz[10];
        matriz[3]=10;
        matriz[9]=5;
        ....
}
```

Ejemplo 2:

```
void main()
{
        int m[10];
        m[3]=m[9]=4
        m[0]=0;
        ....
}
```

Ejemplo 3:

```
void main()
{
        int m[5],i=3;
        m[i]=3;
        m[i*2-1]=65;
        ...
}
```

Utilizando un ciclo **for** para inicializar un arreglo:

```
void main()
{
        int i, m[10];
        for(i=0;i<10;i++) m[i]=0; /* pone en 0 todos los elementos del arreglo
        */
        ...
}
```

Declarando e inicializando un arreglo de 5 elementos.

```
void main()
{
        int ma[5]={3,4,6,89,10};
        ....
}
```

Si se sobrepasan los límites del arreglo, el compilador NO entregará mensaje de error alguno.

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

5.7. Operaciones con arreglos.

Asignación entre arreglos

En C++ no se puede asignar un arreglo completo a otro arreglo. Por ejemplo, este fragmento es incorrecto.

```
char a1[10], a2[10];
...
a2=a1; // Es incorrecto
```

Si desea copiar los valores de todos los elementos de una arreglo a otro debe hacerlo copiando cada elemento por separado. Por ejemplo, el siguiente programa carga a1 con los números 1 a 10 y después los copia en a2.

```
#include <iostream>
int main()
{
        int a1[10], a2[10];
        int i;
        //Inicialización de a1
        for (i=0; i<10;i++)
                a1[i]=i+1;
        //Copiar en a2
        for (i=0; i<10;i++)
                a2[i]=a1[i]
        //Mostrar a2
        for (i=0; i<10;i++)
                cout<<a2[i]<<endl;
        return 0;
}
```

Suma y Resta

Los arreglos deben tener el mismo tamano y la suma se realiza elemento a elemento. Por ejemplo C =

A + B. Donde A, B y C son arreglos de enteros de tamano 3.

C		=		A		+		B
c00=a00+b00				a00				b00
c01=a01+b01		=		a01		+		b01
c02=a02+b02				a02				b02

```
int A[3],B[3],C[3];
for (int j=0;j<3;j++)
{
        A[j] = j*3; // Asignación de valores para arreglo A
        B[j] = j-2; // Asignación de valores para arreglo B
        C[j]=A[j]+B[j]; // Asignación de valores para arreglo C
}
```


cnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Operaciones con arreglos multidimensionales

En matemáticas, una matriz es una tabla de números consistente en cantidades abstractas que pueden sumarse y multiplicarse. Las matrices se utilizan para describir sistemas de ecuaciones lineales, realizar un seguimiento de los coeficientes de una aplicación lineal y registrar los datos que dependen de varios parámetros. Pueden sumarse, multiplicarse y descomponerse de varias formas, lo que también las hace un concepto clave en el campo del algebra lineal. Las matrices son utilizadas ampliamente en la computación, por su facilidad para manipular información. En este contexto, son la mejor forma para representar grafos, y son muy utilizadas en el cálculo numérico.

Propiedades

* Asociativa

Dadas las matrices m×n A, B y C

A + (B + C) = (A + B) + C

* Conmutativa

Dadas las matrices m×n A y B

A + B = B + A

* Existencia de matriz cero o matriz nula

A + 0 = 0 + A = A

* Existencia de matriz opuesta con -A = [-aij]

A + (-A) = 0

Suma

Los arreglos deben tener el mismo orden y la suma se realiza elemento a elemento. Por ejemplo sean A,B y C arreglos de numeros punto flotante de orden 2x3. Entonces la operación C = A+B seria:

C = A + B

c00=a00+b00	c01=a01+b01	c02=a02+b02
c10=a10+b10	c11=a11+b11	c12=a12+b12
c20=a20+b20	c21=a21+b21	c22=a22+b22

=

a00	a01	a02
a10	a11	a12
a20	a21	a22

+

b00	b01	b02
b10	b11	b12
b20	b21	b22

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

```
float A[3][3],B[3][3],C[3][3];
for (int i=0;i<2;++)
{
        for (int j=0;j<3;j++)
        {
                A[i][j] = (2*i+1)/3; // Asignación de valores para el arreglo A
                B[i][j] = 2*j; // Asignación de valores para el arreglo B
                C[i][j]=A[i][j]+B[i][j]; // Asignación de valores para el arreglo C
        }
}
```

Producto por un escalar

Dada una matriz A y un escalar c, su producto cA se calcula multiplicando el escalar por cada elemento de A

$$(cA)[i, j] = cA[i, j]$$

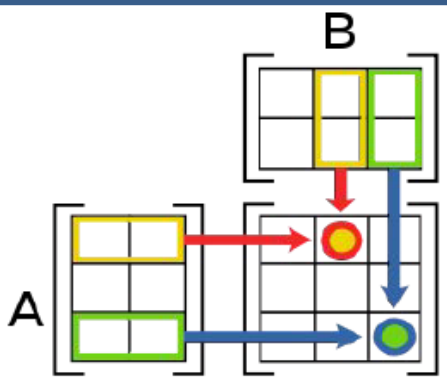
Ejemplo:

$$2\begin{bmatrix}1 & 8 & -3\\ 4 & -2 & 5\end{bmatrix} = \begin{bmatrix}2\times 1 & 2\times 8 & 2\times -3\\ 2\times 4 & 2\times -2 & 2\times 5\end{bmatrix} = \begin{bmatrix}2 & 16 & -6\\ 8 & -4 & 10\end{bmatrix}$$

Producto de matrices

El producto de dos matrices se puede definir solo si el numero de columnas de la matriz izquierda es el
mismo que el numero de filas de la matriz derecha. Si A es una matriz m×n y B es una matriz n×p,
entonces su producto matricial AB es la matriz m×p (m filas, p columnas) dada por:

$$(AB)[i,j] = A[i,1]\,B[1,j] + A[i,2]\,B[2,j] + \ldots + A[i,n]\,B[n,j] \text{ para cada par } i \text{ y } j.$$



Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

$$\begin{bmatrix} 1 & 0 & 2 \\ -1 & 3 & 1 \end{bmatrix} \times \begin{bmatrix} 3 & 1 \\ 2 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} (1\times 3+0\times 2+2\times 1) & (1\times 1+0\times 1+2\times 0) \\ (-1\times 3+3\times 2+1\times 1) & (-1\times 1+3\times 1+1\times 0) \end{bmatrix} = \begin{bmatrix} 5 & 1 \\ 4 & 2 \end{bmatrix}$$

5.8. Arreglos de caracteres.

Las cadenas de caracteres (string) son arreglos que se componen de caracteres alfanuméricos. El final de una cadena se indica con un carácter especial, éste es un byte compuesto por 8 ceros binarios, es decir 0x00.
Luego, el tamaño de la cadena debe ser lo suficientemente grande para contener dicho carácter inclusive. Una cadena declarada como cad[10], podrá contener como máximo 9 caracteres.
El archivo de cabecera **string.h** proporciona una serie de funciones que permiten el manejo de cadenas de caracteres. No existen operadores que permitan comparar cadenas o copiar una cadena en otra, por lo que se debe hacer uso de las funciones definidas en **string.h**.

```
#include<stdio.h>
#include<string.h>
void main()
{
        char cad[10], buf[10];
        strcpy(cad, "HOLA"); /* Copia en Cad la cadena "HOLA" */
        strcpy(buf, cad); /* Pasa el contenido de cad a buf */
        if (!strcmp(buf,cad)) printf("SON IGUALES!");

}
```

La función **strcpy** es una contracción de la operación **str**ing **co**pia**py**. Al realizar dicha operación la primera vez, el contenido de cad es el siguiente:

0x48	0x4f	0x4c	0x41	0x00	??	??	??	??	??
H	O	L	A						

La función **strcmp** permite comparar dos cadenas de caracteres. Esta comparación se realiza hasta el carácter 0x00 de final de cadena. Es importante distinguir entre 'A' y "A", el primero es el carácter A que se codifica en un byte (0x41) y el segundo es la cadena que contiene un carácter A y ocupa dos bytes. (0x41,0x00).

Inicializando una matríz de caracteres.

```
void main()
{
        char nombres[3][10]={ "IGNACIO","PEDRO","JUAN"};
}
```


Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

5.9. Arreglos multidimensionales.

Es una estructura de datos estática y de un mismo tipo de datos, y de longitud fija que almacena datos de forma matricial. De igual forma que los arreglos unidimensionales, el almacenamiento de los datos en la memoria se realiza de forma secuencial y son accedidos mediante índices. Los arreglos multidimensionales son también conocidos como matrices. Por lo tanto se llama matriz de **orden** "m×n" a un conjunto rectangular de elementos dispuestos en filas "m" y en columnas "n", siendo m y n números naturales. Las matrices se denotan con letras mayúsculas: A, B, C, ... y los elementos de las mismas con letras minúsculas y subíndices que indican el lugar ocupado: a, b, c, ... Un elemento genérico que ocupe la fila i y la columna j se escribe i,j. Si el elemento genérico aparece entre paréntesis también representa a toda la matriz: A (i,j).

Declaración de arreglo bidimensional
tipo ident_arreglo[num_fila][num_col];

Ejemplo:

```
void()
{
        int tabla[10][2];
...
}
```

Declaración e inicialización de una matriz de dos dimensiones.

```
void main()
{
        int a[2][3]= { {1,2,3},{4,5,6}};
}
```

En este caso es una matriz que tiene dos filas y tres columnas, donde:
a[0]: contiene la dirección de memoria de la primera fila
a[1]: contiene la dirección de memoria de la segunda fila.
a contiene la dirección de a[0].

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

UNIDAD 6 Estructuras

6.1. Estructuras: declaración e inicialización. Variables del tipo struct.

Así como los arrays son organizaciones secuenciales de variables simples, de un mismo tipo cualquiera dado, resulta necesario en múltiples aplicaciones, agrupar variables de distintos tipos, en una sola entidad.

Para definir una estructura usamos el siguiente formato:

```
struct nombre_de_la_estructura {
 campos de estructura;
 };
```

Si quisiéramos generar la variable " legajo personal " , en ella tendríamos que incluir variables del tipo: strings , para el nombre , apellido , nombre de la calle en donde vive , etc , enteros , para la edad , número de código postal , float para el sueldo, y así siguiendo . Existe en C en tipo de variable compuesta, para manejar ésta situación típica de las Bases de Datos, llamada ESTRUCTURA. No hay limitaciones en el tipo ni cantidad de variables que pueda contener una estructura, mientras su máquina posea memoria suficiente como para alojarla, con una sola salvedad: una estructura no puede contenerse a sí misma como miembro. Para usarlas, se deben seguir dos pasos. Hay que, primero declarar la estructura en sí, ésto es, darle un nombre y describir a sus miembros, para finalmente declarar a una ó más variables, del tipo de la estructura antedicha, veamos un ejemplo:

```
struct legajo {
        int edad ;
        char nombre[50] ;
        float sueldo ;
struct legajo legajos_vendedores , legajos_profesionales;
```

En la primer sentencia se crea un tipo de estructura, mediante el declarador "struct",luego se le dá un nombre " legajo " y finalmente , entre llaves se declaran cada uno de sus miembros, pudiendo estos ser de cualquier tipo de variable , incluyendo a los arrays ó aún otra estructura . La única restricción es que no haya dos miembros con el mismo nombre, aunque sí pueden coincidir con el nombre de otra variable simple, (o de un miembro de otra estructura), declaradas en otro lugar del programa. Esta sentencia es sólo una declaración, es decir que no asigna lugar en la memoria para la estructura, sólo le avisa al compilador como tendrá que manejar a dicha memoria para alojar variables del tipo struct legajo. En la segunda sentencia, se definen dos variables del tipo de la estructura anterior (ésta definición debe colocarse luego de la declaración), y se reserva memoria para ambas . Las dos sentencias pueden combinarse en una sola, dando la definición a continuación de la declaracion:

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

```
struct legajo {
        int edad ;
        char nombre[50] ;
        float sueldo ;
} legajo_vendedor , legajo_programador ;
```

Y si no fueran a realizarse más declaraciones de variables de éste tipo, podría obviarse el nombre de la estructura (legajo).
Las variables del tipo de una estructura, pueden ser inicializadas en su definición, así por ejemplo se podría escribir:

```
struct legajo {
        int edad ;
        char nombre[50] ;
        float sueldo ;
        char observaciones[500] ;
} legajo_vendedor = { 40 , "Juan Eneene" , 1200.50 ,"Asignado a zona A" } ;
```

Acá se utilizaron las dos modalidades de definición de variables, inicializándolas a ambas.

6.2. Almacenamiento y recuperación de información en estructuras.

Lo primero que debemos estudiar es el método para dirigirnos a un miembro particular de una estructura .Para ello existe un operador que relaciona al nombre de ella con el de un miembro, este operador se representa con el punto (.) , así se podrá referenciar a cada uno de los miembros como variables individuales, con las particularidades que les otorgan sus propias declaraciones, internas a la estructura. La sintaxis para realizar ésta referencia es: nombre_de_la_estructura.nombre_del_miembro, así podremos escribir por ejemplo, las siguientes sentencias:

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Ejemplo:
Almacenamiento y recuperación de los elementos de una estructura.

```
#include <string.h>
void main()
{
        struct dirección
        {
                char calle[25];
                int numero;
                char nombre[30];
        } d;
        strcpy(d.calle,"Avd. Alemania");
        d.numero=2012;
        strcpy(d.nombre,"Juan");
}
```

Se pueden realizar operaciones si los elementos son de tipo numérico.

```
void main()
{
        struct complex { float x; float y;} x;
        float modulo;
        x.x=0.5;
        x.y=10.0;
        modulo=sqr(x.x*x.x+x.y*x.y);
}
```

En el caso de ser requerido pueden definirse matrices o arrays, es decir:
struct direccion dir[100];
Para acceder a ella se puede hacer de la siguiente forma:
dir[1].numero= 1002;

Las estructuras pueden anidarse, es decir que una ó más de ellas pueden ser miembro de otra. Las estructuras también pueden ser pasadas a las funciones como parámetros, y ser retornadas por éstas, como resultados.

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

6.3. Arreglos de estructuras.

Cuando hablamos de arrays dijimos que se podían agrupar , para formarlos , cualquier tipo de variables , esto es extensible a las estructuras y podemos entonces agruparlas ordenadamente , como elementos de un array . Veamos un ejemplo :

```
typedef struct {
        char material[50] ;
        int existencia ;
        double costo_unitario ;
} Item ;
Item stock[100] ;
```

Hemos definido aquí un array de 100 elementos, donde cada uno de ellos es una estructura del tipo Item compuesta por tres variables, un int, un double y un string ó array de 50 caracteres.
Los arrays de estructuras pueden inicializarse de la manera habitual, así en una definición de stock, podríamos haber escrito:

```
Item stock1[100] = {
        "tornillos" , 120 , .15 ,
        "tuercas" , 200 , .09 ,
        "arandelas" , 90 , .01
} ;
Item stock2[] = {
        { 'i','t','e','m','1','\0' } , 10 , 1.5 ,
        { 'i','t','e','m','2','\0' } , 20 , 1.0 ,
        { 'i','t','e','m','3','\0' } , 60 , 2.5 ,
        { 'i','t','e','m','4','\0' } , 40 , 4.6 ,
        { 'i','t','e','m','5','\0' } , 10 , 1.2 ,
} ;
```


Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Analizando las diferencias entre la dos inicializaciones dadas, en la primera, el array material[] es inicializado como un string, por medio de las comillas y luego en forma ordenada , se van inicializando cada uno de los miembros de los elementos del array stock1[] , en la segunda se ha preferido dar valores individuales a cada uno de los elementos del array material , por lo que es necesario encerrarlos entre llaves . Sin embargo hay una diferencia mucho mayor entre las dos sentencias, en la primera explicitamos el tamaño del array, [100], y sólo inicializamos los tres primeros elementos los restantes quedarán cargados de basura si la definición es local a alguna función , ó de cero si es global , pero de cualquier manera están alojados en la memoria, en cambio en la segunda dejamos implícito el número de elementos, por lo que será el compilador el que calcule la cantidad de ellos, basándose en cuantos se han inicializado, por lo tanto este array sólo tendrá ubicados en memoria cuatro elementos, sin posibilidad de agregar nuevos datos posteriormente.

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

UNIDAD 7 Control de Periféricos.

7.1. Conceptos básicos sobre periféricos.

A grandes rasgos se podría decir que una computadora está compuesta por la CPU (microprocesador), la memoria (ROM, DRAM, caché) y los diversos periféricos.
Se denominan periféricos a todos aquellos módulos de una computadora encargados de realizar tareas tales como:
- Entrada y salida de datos (p.ej.: teclado, monitor, puertos de comunicación, impresoras, interfaces de red, etc.)
- Almacenamiento y recuperación de información (p.ej: los diversos tipos de discos).
- Tareas auxiliares del sistema (p.ej.: temporizadores, controladores de interrupciones, controladores de DMA, etc.)
- Otros: tarjetas de adquisición de datos, tarjetas de sonido y de vídeo, etc.

7.2. Periféricos típicos: Impresoras, monitores, teclados, puertos de comunicación serie, puertos paralelos, discos.

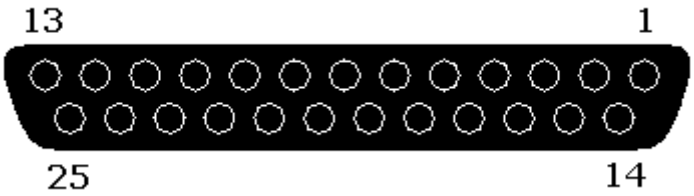
Podríamos pensar en una cierta clasificación entre periféricos lentos (p.ej. impresora) y periféricos rápidos (p.ej.: disco rígido), en función de la tasa de transferencia de información.
Esto determina la forma más conveniente de tratar a cada uno, fundamentalmente a fin de optimizar el tiempo requerido para efectuar una transferencia de la memoria al periférico o vicecersa.
Cada periférico tiene sus propias reglas de manejo, definidas por el fabricante o estandarizadas mediante alguna norma a la cual adhieren varios de ellos. En tal sentido no hay reglas universales para, por ejemplo, obtener un dato desde una placa conversora analógica-digital; si bien se aplican conceptos similares para periféricos similares, en los detalles de implementación puede haber diferencias.

Puertos paralelos:

Existen dos métodos básicos para transmisión de datos en las computadoras modernas. En un esquema de transmisión de datos en serie un dispositivo envía datos a otro a razón de un bit a la vez a través de un cable. Por otro lado, en un esquema de transmisión de datos en paralelo un dispositivo envía datos a otro a una tasa de n número de bits a través de n número de cables a un tiempo. Sería fácil pensar que un sistema en paralelo es n veces más rápido que un sistema en serie, sin embargo ésto no se cumple, básicamente el impedimiento principal es el tipo de cable que se utiliza para interconectar los equipos. Si bién un sistema de comunicación en paralelo puede utilizar cualquier número de cables para transmitir datos, la mayoría de los sistemas paralelos utilizan ocho líneas de datos para transmitir un byte a la vez, como en todo, existen excepciones, por ejemplo el estándar SCSI permite transferencia de datos en esquemas que van desde los ocho bits y hasta los treinta y dos bits

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

en paralelo Un típico sistema de comunicación en paralelo puede ser de una dirección (unidireccional) o de dos direcciones (bidireccional). El más simple mecanismo utilizado en un puerto paralelo de una PC es de tipo unidireccional.
Se distinguen dos elementos: la parte transmisora y la parte receptora. La parte transmisora coloca la información en las líneas de datos e informa a la parte receptora que la información (los datos) están disponibles; entonces la parte receptora lee la información en las líneas de datos e informa a la parte transmisora que ha tomado la información (los datos). Observe que ambas partes sincronizan su respectivo acceso a las líneas de datos, la parte receptora no leerá las líneas de datos hasta que la parte transmisora se lo indique en tanto que la parte transmisora no colocará nueva información en las líneas de datos hasta que la parte receptora remueva la información y le indique a la parte transmisora que ya ha tomado los datos, a ésta coordinación de operaciones se le llama acuerdo ó entendimiento.
El puerto paralelo de una típica PC utiliza un conector hembra de tipo D de 25 patitas (DB-25 S), éste es el caso más común, sin embargo es conveniente mencionar los tres tipos de conectores definidos por el estándar IEEE 1284, el primero, llamado 1284 tipo A es un conector hembra de 25 patitas de tipo D, es decir, el que mencionamos al principio. El orden de las patitas del conector es éste:



El segundo conector se llama 1284 tipo B que es un conector de 36 patitas de tipo centronics y lo encontramos en la mayoría de las impresoras; el tercero se denomina 1284 tipo C, se trata de un conector similar al 1284 tipo B pero más pequeño, además se dice que tiene mejores propiedades eléctricas y mecánicas, éste conector es el recomendado para nuevos diseños. La siguiente tabla describe la función de cada patita del conector 1284 tipo A:

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Patita	E/S	Polaridad activa	Descripción
1	Salida	0	Strobe
2 ~ 9	Salida	-	Líneas de datos (bit 0/patita 2, bit 7/patita 9)
10	Entrada	0	Línea acknowledge (activa cuando el sistema remoto toma datos)
11	Entrada	0	Línea busy (si está activa, el sistema remoto no acepta datos)
12	Entrada	1	Línea Falta de papel (si está activa, falta papel en la impresora)
13	Entrada	1	Línea Select (si está activa, la impresora se ha seleccionado)
14	Salida	0	Línea Autofeed (si está activa, la impresora inserta una nueva línea por cada retorno de carro)
15	Entrada	0	Línea Error (si está activa, hay un error en la impresora)
16	Salida	0	Línea Init (Si se mantiene activa por al menos 50 micro-segundos, ésta señal autoinicializa la impresora)
17	Salida	0	Línea Select input (Cuando está inactiva, obliga a la impresora a salir de línea)
18 ~ 25	-	-	Tierra eléctrica

Todos los ordenadores tipo PC están equipados, al menos, con una tarjeta de interface paralelo, frecuentemente junto a un interface serie. Como sistema operativo, el DOS puede gestionar hasta cuatro interfaces de puertos paralelo, LPT1, LPT2 , LPT 3 y LPT4, además, reserva las siglas PRN como sinónimo del LPT 1, de modo que puede ser tratado como un archivo genérico. En el byte 0040:0011 del BIOS almacena el número de interfaces de puertos paralelo que se hayan instalado en el equipo. La dirección de entrada/salida de cada uno de los puertos paralelo y el número de puertos instalados en un PC se muestra en la pantalla inicial de arranque del equipo es frecuente, casi estandard que las direcciones de los dos primeros puertos paralelo sean las siguientes:
LPT1 = 0x378 Hexadecimal
LPT2 = 0x278 Hexadecimal
Las tarjetas del puerto paralelo tiene una estructura muy simple; consta de tres registros: de control, de estado y de datos. Todas las señales que intervienen en el puerto tienen asociado un bit en uno de esos registros, de acuerdo con

las funciones asignadas a cada línea en particular

Puerto serie:

En lenguaje C, existe una instrucción especial para manejar las comunicaciones seriales. Esta instrucción posee la siguiente sintaxis:

```
int bioscom (int cmd, char abyte, int port);
```


Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

En realidad, esta instrucción acude a la interrupción 14H para permitir la comunicación serial sobre un puerto. Para este caso, cada uno de los parámetros tiene el siguiente significado:

cmd	Operación a realizar.
abyte	Carácter que se enviará por el puerto serial.
port	Identificación del puerto serial (desde 0 para COM1 hasta 3 para COM4)

El parámetro cmd puede tener los siguientes valores y significados:
0 Inicializa el puerto port con los valores dados por abyte
1 Envía el caracter abyte por el puerto port
2 Lee el caracter recibido por el puerto port
3 Retorna el estado del puerto port

Para la inicialización del puerto, el caracter abyte tiene las siguientes interpretaciones:

0x02 0x03	7 bits de datos 8 bits de datos
0x00 0x04	1 bits de parada 2 bits de parada
0x00 0x08 0x18	Sin paridad Paridad impar Paridad par
0x00 0x20 0x40 0x60 0x80 0xA0 0xC0 0xE0	110 baudios 150 baudios 300 baudios 600 baudios 1200 baudios 2400 baudios 4800 baudios 9600 baudios

El canal serie del PC es uno de los recursos más comunes para la conexión de periféricos, como pueden ser dispositivos de puntero (mouse) o de comunicación (modem, cables de conexión entre PCs).
Para configurar el puerto con algunos parámetros, bastará con realizar una operación OR con los deseados, por ejemplo, para 1200 baudios, sin bit de paridad, sin bit de parada y 8 bits,
bastará con seleccionar la palabra dada por: abyte = 0x80 | 0x00 | 0x00 | 0x03 o abyte = 0x83
Para la lectura de un carácter que se haya recibido o del estado del puerto, se deben utilizar variables en las cuales se almacenarán los valores de retorno; en ambos caso se obtienen valores de 16 bits. Para la lectura de un dato recibido, los 8 bits menos significativos corresponden al dato y los 8 más significativos al

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

estado del puerto; si alguno de estos últimos está en "1 ", un error ocurrió; si todos están en "0", el dato fue recibido sin error.
Cuando el comando es 2 ó 3 (leer el carácter o el estado del puerto), el argumento abyte no se tiene en cuenta. Para configurar el puerto COM1 con los parámetros del ejemplo dado anteriormente, bastará con la instrucción: bioscom (0,0x83,0); /*(inicializar, parámetros, COM1)*/
La utilización de los comandos y las instrucciones para la configuración de los puertos aquí expuestos sólo tendrán sentido en la medida en que utilicemos el puerto serial para establecer una comunicación bien con otros computadores o bien con dispositivos electrónicos como microcontroladores.
La siguiente tabla nos muestra el ejemplo de 4 puertos COM típicos. Lo mas normal es que tanto el COM1 como el COM2 estén ahora integrados en la placa base, o puestos en una tarjeta controladora. Los otros 2 se suelen configurar con el MODEM (interno).
El Com1 y el Com3 comparten la misma IRQ. Lo mismo pasa con Com2 y Com4.

Com	Dirección Base	IRQ
Com1	3F8	4
Com2	2F8	3
Com3	3E8	4
Com4	2E8	3

Para enviar una palabra por el puerto COM debemos enviar el dato a THR (a la dirección de Hardware base del puerto). Si hablamos del COM2, deberemos hacer: outportb (0x2F8,dato);
El dato será enviado por el puerto serie, SIEMPRE QUE ESTE LIBRE Y NO OCUPADO.
Para saber si esta ocupado debemos mirar en el LSR (Line Status Register, 2FD) de la siguiente manera:
Si ((LSR&0x20) es igual a 0x20) entonces podremos enviar el dato. Si hablamos del COM2, la pregunta en lenguaje C seria: if ((outportb (0x2FD)&0x20)== 0x20) Poner un getch, un scanf o cualquier tipo de función de consola típica (esperar que el usuario introduzca un dato) puede ser muy perjudicial para el programa, porque si ese momento alguien envía un dato, ¡no se va a leer! va a llegar, pero si durante el tiempo que el usuario esta dudando que poner llega otro dato por el canal serie la información anterior seria sobre escrita por el nuevo dato, y perderíamos parte de la información!!!

Leer datos del puerto serial

Para leer una palabra del puerto COM debemos leer del registro RDR (de la dirección base del puerto). Si hablamos del COM2 deberemos hacer:
inport (0x2F8);
Claro esta que la lectura deberemos guardarla en algún sitio (un char seria lo más indicado).
Pero como en el caso de la escritura en el puerto, también nos interesaría que el puerto nos avisara cuando el puerto tiene un dato nuevo. Deberemos saberlo de la siguiente manera:

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

Si ((LSR&0x01) es igual a 0x01) entonces podremos recibir el dato. Si hablamos del COM2, la pregunta en lenguaje C seria:
if ((outportb (0x2FD)&0x01)== 0x01)

Ejemplo: Un programa que configura el puerto serie COM2, envía todo el rato el dato 'A' por el canal, y en caso de que reciba algo lo muestra por pantalla.

```
#include <stdio.h>
#include <dos.h>
void main (void)
{
        outportb (0x2FB,0x83);
        outportb (0x2F8,0x60);
        outportb (0x2F9,0x00);
        outportb (0x2FB,0x03);
        //Puerto configurado para 1200 baudios, 1 stop bit, 8 bits de datos y no
        paridad
        for(;;)
        {
                if ((outportb (0x2FD)&0x01)== 0x01) //en caso de recibir
                {
                        printf ("%c",inportb(0x2F8)); //Lo sacamos por pantalla
                }
                if ((outportb (0x2FD)&0x20)== 0x20) //en caso de poder enviar
                {
                        outportb (0x2F8,'A'); // Lo mandamos por el puerto
                }
        }
}
```

7.3 Manejo Periféricos usando lenguaje C.

Los parámetros básicos a tener en cuenta para el manejo de un periférico son los siguientes:
- Rango de direcciones de entrada-salida: habitualmente se ubican sobre el mapa de entradasalida los registros de control y operación del periférico. Estos registros pueden servir a los fines de programar y controlar el dispositivo, verificar su estado, enviar o recibir información.
- Rango de direcciones de memoria: algunos periféricos pueden tener o bien registros o bien algún tipo de memoria ubicada en el mapa de memoria del sistema. Un ejemplo típico es la controladora de vídeo en una PC, la cual tiene una cierta cantidad de memoria en la cual se almacenan los códigos ASCII de los caracteres que se desea aparezcan en la pantalla.
- Nº de línea de requerimiento de interrupción (IRQ): algunos periféricos tienen la capacidad de generar un requerimiento de interrupción, a fin de que el procesador suspenda lo que está haciendo en un momento dado, para pasar a atender el requerimiento de dicho periférico.
Un ejemplo podría ser la interface del puerto de comunicación serie de una PC, el cual se puede habilitar para que genere una interrupción avisando que ha sido recibido un nuevo dato. Si, por el contrario, un periférico no tiene la

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

capacidad de generar pedidos de interrupción, deberá consultarse periódicamente (polling) para conocer si se ha producido algún cambio de interés en el mismo.
El uso de interrupciones permite obtener una mayor performance, a costa de un cierto aumento en la complejidad de los programas.

Nº de canal de acceso directo a memoria (DMA): los periféricos que tienen una alta velocidad de transferencia de información, y además deben mover cantidades importantes de datos (p.ej.: el disco rígido), habitualmente no realizan esas transferencias bajo el control del procesador debido a la pérdida de tiempo que se ocasionaría. Veamos por ejemplo que sucede si se debe transferir un bloque de información desde el disco a la memoria: el procesador debe leer un dato (p.ej.: un byte) desde la interface del disco, y posteriormente debrá escribir el mismo en la memoria, con lo cual se realizaron dos accesos al bus del sistema. Una alternativa mejor es contar con un controlador de DMA, bajo la supervisión del cual la transferencia de información se podría realizar directamente entre el periférico y la memoria, sin pasar por el procesador central.
Debido a la complejidad de este tipo de operaciones, que requieren conocimientos más avanzados de la arquitectura y estructura interna de la computadora, este tipo de transferencias se menciona solamente a título informativo.

Manejo de periféricos usando Lenguaje C:

El lenguaje C es muy adecuado para el manejo de periféricos, debido a que si bien brinda un cierto nivel de abstracción, también cuenta con las herramientas necesarias para acceder a cualquier lugar de la memoria (punteros de diverso tipo), para leer y escribir en direcciones de entrada/salida (funciones inportb() y outportb()), como así también para efectuar operaciones a nivel de bits o grupos de bits.
Asimismo es sencillo implementar en C rutinas para la atención de interrupciones.

7.4 Funciones para el control de puertos de comunicación.

Directiva del preprocesador
#include <dos.h>

- Funciones de envío de datos

void outport (unsigned portid, unsigned value); //envío de 2 bytes
void outportb (unsigned portid, unsigned char value); //envío de 1 byte

- Funciones de lectura de datos

unsigned inport (unsigned portid); //lectura de 2 bytes
unsigned char inportb (unsigned portid); //lectura de 1 byte

Funciones de lectura y envío de datos por el puerto paralelo

/* Ejemplo básico de E/S digital mediante el puerto paralelo

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

```
Salidas: líneas de datos (bit D0-D7 del registro de datos)
Entradas: líneas de estado (bit S3-S7 del registro de estado)
Suponemos puerto estándar en la dirección 0x378 */
#include <stdio.h>
#include <dos.h>
main ()
{
        unsigned char byte; /* byte para operaciones de E/S */
        printf ("Introduce el byte que se enviar al puerto: ");
        scanf("%u", &byte);
        getchar();
                outportb (0x378, byte); /* envía un byte a las líneas de datos */
        printf ("Polariza las líneas de estado y pulsa una tecla\n");
        getchar();
        byte = inportb (0x378+1); /* lee un byte de las l¡neas de estado */
        printf ("El valor leído es %i", byte);
        return 0;
}

/* Ejemplo básico de E/S digital mediante el puerto paralelo
Suponemos puerto bidireccional (comprobar en BIOS) en la dirección 0x378 */
#include <stdio.h>
#include <dos.h>
#define LPT_BASE 0x378 /* Dirección base del puerto paralelo */
#define DATOS LPT_BASE /* Direcci¢n de E/S del reg. de datos */
#define CONTROL LPT_BASE+2 /* Direcci¢n de E/S del reg. de control */
#define C5_ON 0x20 /* Bit 5 de CONTROL a 1 */
#define C5_OFF 0x00 /* Bit 5 de CONTROL a 0 */
main ()
{
        unsigned char byte; /* byte para operaciones de E/S */
        printf ("Introduce el byte que se enviar al puerto: ");
        scanf("%u", &byte); getchar();
        /* pone el puerto en modo salida */
        outportb (CONTROL, C5_OFF );
        outportb (DATOS, byte); /* envía un byte */
        printf ("Polariza las l¡neas del puerto y pulsa una tecla\n");
        getchar();
        /* pone el puerto en modo entrada */
        outportb (CONTROL, C5_ON );
        byte = inportb (DATOS); /* lee un byte */
        printf ("El valor le¡do es %i", byte);
        return 0;
}
```

Funciones de lectura y envío de datos por el puerto serie

Manejo por “polling” de un periférico ubicado en el mapa de E/S En la dirección de E/S 3F8h se encuentra el registro de datos del puerto serie COM1. Si se escribe un dato de un byte en dicho puerto, el dato es transmitido. Por otra

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

parte, si por la línea de recepción se ha recibido un byte, se lo puede obtener leyendo en la misma dirección (0x3F8).
Además, en la dirección de E/S 3FDh se encuentra el registro de estado del puerto, que mediante los bits 0 y 5 indica las siguientes condiciones:
- bit 0 : cuando está a “1” significa que hay un dato recibido en espera de ser leido; luego de la lectura el mismo se pone a “0” automáticamente.
- bit 5: cuando está a “1” indica que está disponible para enviar un nuevo dato; al escribir un nuevo byte para ser enviado el mismo se pone a “0” automáticamente.

Transmisión de todos los caracteres que se ingresan por el teclado (también los muestra en pantalla), y envía a la pantalla los datos recibidos.

```
#include <stdio.h>
#include <conio.h>
#define ESCAPE 27
// union usada para consultar el estado del puerto
union estado_puerto
{
	unsigned char byte;
	struct
	{
		unsigned int b0 : 1;
		unsigned int unused : 4;
		unsigned int b5 : 1;
	}bit;
}estado;
void main()
{
	int t, r;
	while( 1 )
	{
		estado.byte =inportb(0x3FD); //lee estado del puerto
		if (estado.bit.b5)
		{
			// se puede transmitir,
			// ¿hay algún dato para enviar?
			if( kbhit() )
			{
				// hay un carácter en el buffer de teclado
				t = getch(;) // lee el carácter
				if( t==27 )
					break; // ESCAPE = sale del loop
				outportb(0x3F8, t); //transmite el dato
				putchar(t); //y lo envía a su propia pantalla
			}
		}
		if ( estado.bit.b0 )
		{
			// se ha recibido un dato por el puerto
			r=inportb(0x3F8); // se obtiene el dato
```


Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5

```
                    recibido
                    putchar(r); // y se lo envía a la pantalla
                }
        }
}
```

BIBLIOGRAFÍA:

Titulo	Autores	Editorial	Año Edición.
Como programar en C/C++	Deitel H. M. y P. J. Deitel	Prentice Hall	2000
Programación en C	Aguilar L. J. – Martinez I. Z.	Mc. Graw-Hill	2001
Programación en C. 2º edición	Byron S. Gottfried	Mc. Graw-Hill	2005
Programación en C++. Un enfoque práctico	Joyanes Aguilar y Sánchez García	Mc Graw Hill, serie Schaum	2006
El lenguaje de programación C++	Stroustrup Bjarne	Pearson Educación	2002
Programación en C++ para ingenieros. 2º edición	Xhafa fatos, Gómez Jordi, M. Prta A., Molinero Albareda X., Vazquez Alcocer Pere-Pau	PARANINFO	2006

Universidad Nacional de Catamarca – Secretaría de Ciencia y Tecnología
Editorial Científica Universitaria
ISBN: 978-987-661-127-5