APUNTE TEÓRICO DE PROGRAMACIÓN II (C/C++)

UNIDAD I

Consideraciones Preliminares

C no es un lenguaje dócil para enfrentarlo intuitivamente por primera vez; se requiere un mínimo conocimiento de sus fundamentos antes de poner las manos sobre el teclado.

En este aspecto es particularmente importante comprender los diferentes tipos de datos y las reglas que rigen su operación.

La idea directriz de C es la definición de procedimientos (funciones), que en principio devuelven un valor. Lo que para nosotros es -conceptualmente- el programa principal, también es en C una función (la más externa). Incidentalmente, su valor es devuelto al sistema operativo como código de conclusión del programa.

Ante todo, C está diseñado con vistas a la compatibilidad. En este sentido, todas las definiciones que puedan hacerse no serán concretas, pues son adaptables de acuerdo con la implementación. Un entero, por ejemplo, es una entidad con ciertas características generales, pero su implementación diferirá en distintos equipos.

C maneja los datos en forma de variables y constantes, conceptos con los que supondremos que el alumno está familiarizado. Las variables, simbolizadas mediante alfanuméricos (cuyas reglas de construcción trataremos más adelante), presentan características que será muy importante considerar:

- **Tipo de dato**: cada variable (también las constantes) está caracterizada por el tipo de dato que representa.
- **Visibilidad:** en un programa C, cada variable tiene un rango de visibilidad (procedimientos en los que es reconocida), que depende de cómo se la haya declarado.
- **Existencia:** relacionado con la anterior característica, es posible que el contenido de una variable perdure, o que se pierda, por ejemplo, al terminarse un procedimiento.

Set de caracteres

C emplea dos sets (conjuntos) de caracteres:

- El primero de ellos incluye todos los caracteres que tienen algún significado para el compilador.

- El segundo incluye todos los caracteres representables.

C acepta sólo ciertos caracteres como significativos. Sin embargo, otros caracteres pueden formar parte de expresiones literales (constantes literales, nombres de archivo, etc.) que no serán analizadas por C.

Caracteres interpretados por C

Los caracteres a los que C asigna especial significado se pueden clasificar en alfanuméricos y signos especiales. Los caracteres alfanuméricos incluyen las letras (alfabeto inglés, de A a Z), mayúsculas y minúsculas, los dígitos, y el guión bajo (underscore: '_').

En todos los casos, **las mayúsculas son consideradas distintas de las minúsculas.** Toda cadena alfanumérica con significación en C está compuesta exclusivamente por estos caracteres.

Los signos especiales son los listados en la figura 1. Ellos se emplean como delimitadores, operadores, o signos especiales.

Mayúsculas:	A - Z	Signo más	+
Minúsculas:	a - z	Signo menos	-
Dígitos:	0 - 9	Paréntesis izquierdo	(
Guión bajo:	_	Paréntesis derecho	)
Coma	,	Corchete izquierdo	[
Punto	.	Corchete derecho	]
Punto y coma	;	Llave izquierda	{
Dos puntos	:	Llave derecha	{
Signo de interrogación	?	Signo Mayor	>
Signo de admiración	!	Signo Menor	<
Comilla simple	'	Signo igual	=
Comilla doble	"	Asterisco	*
Barra vertical	|	Ampersand	&
Barra	/	Porciento	%
Barra invertida	\	Caret	^
Tilde	~		

Fig. I . Set de caracteres

Estructura De Un Programa En C:

Para escribir un programa en C hay que respetar una estructura cuyas secciones tienen un cometido bien definido.

Es imperativo respetar esta estructura, (y otras normas que más adelante se presentan), ya que el compilador de C, para poder generar nuestro programa objeto, (archivo ejecutable), analiza de forma muy minuciosa.

En forma general, se podría decir que la estructura de un programa en C es la siguiente.

```
Inclusión de archivos de cabecera
Declaración de constantes (opcional)
Definición de tipos (opcional)
Declaración de variables globales (opcional)
Prototipos de funciones del usuario (opcional)
Void main (void)
{
  Declaración de variables.
  Sentencias...
}
Definición de funciones del usuario (opcional)
```

El programa fuente en C lleva como extensión la letra C, o sea miprog.c y en caso de ser escrito en C++ puede alternativamente usar la extensión .cpp. En ambos casos son archivos de texto que pueden ser escritos con cualquier editor de texto aunque por lo general se usa el entorno, (IDE), que provee el lenguaje.

Archivos De Cabecera:

Los archivos de cabecera son archivos cuya extensión es .h, (ejemplo stdio.h), y en principio uno incluye en su programa aquellos archivos necesario. Un archivo de cabecera contiene declaraciones de variables y constantes, prototipos de funciones, macros, etc.

El lenguaje C ofrece una cantidad de importante de estos archivos para que uno pueda escribir los programas y hacer uso de diversas funciones que permitan, por ejemplo, ingresar datos por teclado, utilizar funciones matemáticas, utilizar funciones para manipular cadenas, funciones gráficas, funciones para manipular archivos, la BIOS, placa video, y muchos etc.

Descripción De Algunos Archivos De Cabecera:

STDIO.H

El programa más sencillo de escribir en C necesita la inclusión de este archivo, ya que aquí se encuentran las funciones básicas de entrada/ salida, (en el futuro E/S), (stdio significa "Standar Input Output). Funciones para poder ingresar datos por teclado y para mostrar por pantalla, además de algunas otras.

CONIO.H

Este es otro de los archivos de cabecera más usados, aquí hay más funciones de E/S y constantes.

STRING.H

Funciones para manipular cadenas.

STDLIB.H

Funciones y macros más usadas.

TIME.H

Funciones relacionadas con la hora del sistema, incluso la función para generar números aleatorios.

DOS.H

Funciones del sistema operativo. Búsqueda de archivos, creación de directorios, estructuras de archivo.

BIOS.H

Ofrece funciones para acceder a la configuración de la BIOS y obtener información del sistema.

GRAPHICS.H

Funciones gráficas. Detectar la placa de vídeo, su configuración, funciones de dibujo.

DIR.H

Más funciones para la manipulación de la estructura de almacenamiento del sistema.

MATH.H

Funciones matemáticas tipo seno, coseno, potencia, etc.

Los archivos de cabecera se incluyen de la siguiente forma:

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

Se utilizan los símbolos < > cuando el archivo de cabecera se encuentra en el directorio por defecto del lenguaje C instalado, el directorio por defecto para Borland C/C++ 3.1 es "BC31\INCLUDE".

Ahora si usted creo el archivo .h, (uno puede crear sus propios archivos de cabecera) lo incluye de la siguiente forma:

```
#include "miarchivo.h"
```

"Hola C"

Veamos ahora un ejemplo sencillo, un programa que muestra por pantalla el texto "Hola C".

```
/*Programa que muestra por pantalla el mensaje Hola C*/

#include <stdio.h> // Se incluye este archivo ya que en él se encuentra
                   //    la función de salida printf().
#include <conio.h> //Aquí se encuentra la función para borrar la pantalla.

void main (void)       //función principal
{
  clrscr();      //borro pantalla
  printf("Hola C"); //Muestro el mensaje
}
```

Comentarios:

En C tradicional los comentarios se colocan entre /* y */, pero como este apunte está orientado para el uso de Borland C/C++ 3.1 o Microsoft Visual C++ también se puede usar para comentarios // que permiten comentarios en una sola línea, este juego de caracteres para presentar comentarios es propio de C++.

```
/*Esto es un comentario*/
//y esto también
```

Puntos Y Comas/ Llaves:

Las sentencias ejecutables en C/C++ terminan en **;**

Las llaves agrupan un conjunto de sentencias ejecutables en una misma función o en alguna estructura de iteración o comparación (ver más adelante).

La Función main:

Todo programa en C/C++ tienen una función llamada **main()**.

En esta función se encuentran las sentencias ejecutables entre { }.

Más adelante se aclara el significado de **void** al definir la función main().

Otro ejemplo:

```
(1)  /*Programa que calcula el cuadrado de un número ingresado por teclado.*/
(2)  #include <stdio.h>

(3)  void main(void)
(4)  {
(5)   int a;
(6)   int r;

(7)   printf("Ingrese número: ");
(8)   scanf("%d", &a);

(9)   r=a*a;

(10) printf("\nEl resultado de %d elevado al cuadrado es %d", a, r);
(11) }
```

Veamos línea X línea:

1- Comentarios. Es de buen programador escribir un par de líneas al principio explicando que hace el programa y alguna que otra cosa útil.
2- Se incluye en este caso el archivo stdio.h para el correcto funcionamiento del programa ya que está haciendo uso de las funciones **printf()** y **scanf()**. Se deben incluir todos los archivos de cabecera necesarios, de acuerdo con las funciones a utilizar.
3- Como ya se dijo, todos los programas en C comienzan con las ejecución de la función **main**. El primer **void** indica que la función no retorna valores y el **void** entre () que no recibe valores o parámetros. (más adelante, cuando se vean funciones, se extenderá esta explicación).
4- Llave que abre el conjunto de sentencias pertenecientes a **main**().
5- Y (6) Declaración de variables. En este caso dos variables de tipo int, (enteras).
6- Uso de la función printf() con una cadena de caracteres como único argumento. Salida del mensaje por pantalla.
7- Uso de la función scanf(). El %d como primer argumento, dice que lea un entero y lo guarde en la variable a. El símbolo & se explica más adelante.
8- El cálculo a*a se almacena en r.
9- En este caso la función printf() tiene tres argumentos, la cadena de caracteres y las variables a y b. Los %d en la cadena le dice que intercale los contenidos de la variables enteras en esas posiciones de la cadena. El \n delante de la cadena indica que se debe realizar un salto de línea y retorno de carro antes de imprimir el texto. La salida en caso que a =4 sería:
10- El resultado de 4 elevado al cuadrado es: 16.
11- La llave que cierra la función **main**().

Identificadores:

Los identificaremos se utilizan para identificar, (valga la redundancia): variables, constantes, funciones, etc.

Deben comenzar con una letra. Máxima longitud: 32 caracteres.

Sólo pueden contener letras y números, pero no caracteres especiales, salvo el guión bajo, (underscore).

No se deben confundir con palabras reservadas de C, (una variable, por ejemplo no puede llamarse int, ni main, etc.) y hay diferencias entre mayúsculas y minúsculas. Como norma se utilizan las minúsculas; las mayúsculas se usan para las constantes.

Tipos De Datos Básicos:

El estándar Ansi define un conjunto de tipos básicos y su tamaño mínimo. En distintas implementaciones del lenguaje C estos tamaños puede cambiar, así por ejemplo: el int se define en Ansi C como un tipo de dato que almacena enteros en un rango de –32767 hasta +32767, En ciertas impelementaciones de C para entornos de 32 bits el tipo int posee un rango de -2147483648 a 2147483647.

TIPO	ANCHO EN BITS	BYTES	RANGO
char (carácter)	8	1	-128 a 127
int (entero)	16	2	-32758 a 32767
float (real)	32	4	3.4e-38 a 3.4e+38
double (real de doble precisión).	64	8	1.7e-308 a 1.7e+308
void	0	0	-

Los tipos de datos se pueden modificar utilizando algún **modificador de tipo**:
signed, unsigned, long y short.

TIPO	ANCHO EN BITS	BYTES	RANGO
unsigned int	16	2	0 a –65535
long int	32	4	-2147483648 a 2147483647
Unsigned long int	32	4	0 a 4294967295

Operadores Aritméticos:

OPERADOR	ACCIÓN
-	Resta
+	Suma
*	Multiplicación
/	división.
% (sólo para enteros)	Resto de la división entera
--	Decremento
++	Incremento.

Los operadores de decremento e incremento equivalen a:

```
a=a + 1 → a++
a=a - 1  → a - -
```

En caso de presentarse a con el operador delante:

```
A = 8;
B = ++A;
```

b toma el valor de 9.

Pero de plantearse lo siguiente:

```
A = 8;
B = A++;
```

b toma el valor de 8.

O sea que en este último caso, primero ocurre la asignación y luego el incremento en a.

Operadores Relacionales:

OPERADOR	ACCIÓN
>	Mayor que
>=	Mayor igual que
<	Menor que
<=	Menor igual que
==	Igual que
!=	Distinto que

Operadores Lógicos:

OPERADOR	ACCIÓN
&&	And
||	Or
!	Not

En C, cualquier valor distinto de 0 es VERDADERO. FALSO es 0 (cero).

Declaración De Variables:

En C siempre se deben declarar las variables.

La declaración consiste en un tipo de dato, seguido por el nombre de la variable y el punto y coma:

```
int a;     int b,c,d;   int a = 10;
```

Los tres casos son definiciones correctas de variables, en el último además de declarar la variable se le asigna un valor inicial.

En caso de existir una expresión con variables de diferentes tipos, el resultado obtenido es del tipo de operando de mayor precisión.

Todos los char se convierten a int.

Todos los float se convierten a double.

(hay que tener en cuenta que el tipo char es en realidad un int de menor precisión).

Conversión De Tipos (Cast):

A veces es útil, o necesario, realizar conversiones explícitas para obligar que una expresión sea de un cierto tipo.

La forma general de esta conversión en C es:

```
(tipo) expresión;
```

en C++.

```
tipo(expresión);
```

siendo **tipo**, el tipo de datos al que se convertirá la expresión.

NOTA: Esta conversión de tipos, (también llamada CAST), permite **convertir expresiones** no variables, esto es, si tengo una variable x de tipo int y le aplico **(float)x** lo que se convierte es el resultado, en caso que yo asigne esta operación a otra variable, pero **no se convierte la variable x a float.**

Supongamos que hacemos un programa que divide 10 por 3, uno sabe que el resultado será flotante: 3.333, y como 10 y 3 son enteros uno escribiría:

```
int a=10, b=3;
float r;
r=a/b;
printf("El resultado es %f", r);
```

pero se encontraría que el resultado no es el deseado, esto ocurre porque en C la división entre enteros da como resultado un entero y en la realidad no siempre es así, (sólo en el caso que b sea divisor de a). Pero cambiando el cálculo de la división por:

```
r=(float)a/b;
```

así se garantiza que el resultado será flotante.

Las Funciones printf() Y scanf():

Son dos funciones que están incluidas en el archivo de cabecera stdio.h y se utilizan para mostrar información por pantalla, (o el dispositivo de salida deseado) e ingresar datos por teclado.

FUNCIÓN printf():

El número de parámetro pasados puede variar, dependiendo de la cantidad de variables a mostrar. El primer parámetro indica, por un lado, los caracteres que se mostrarán por pantalla y además los formatos que definen como se verán los argumentos, el resto de los parámetros son las variables a mostrar por pantalla.

Ejemplo:

```
int a=100, b=50;
printf("%i es mayor que %i", a, b);
```

se visualizará por pantalla:

```
"100 es mayor que 50"
```
(sin las comillas).

Los formatos más utilizados con printf() son:

CODIGO	FORMATO
%c	Un solo carácter
%d	Decimal (un entero)
%i	Un entero
%f	Punto decimal flotante
%e	Notación científica
%o	Octal
%x	Hexadecimal
%u	Entero sin signo
%s	Cadena de caracteres
%%	Imprime un signo %
%p	Dirección de un puntero

Los formatos pueden tener modificadores para especificar el ancho del campo, el número de lugares decimales y el indicador de alineación a la izquierda.

Ejemplos:

%05d, un entero de 5 dígitos de ancho; rellenará con ceros. Alineado a la derecha.

%10.4f, un real de 10 dígitos de ancho, con 4 decimales. Alineado a la derecha.

%-10.2f, un real de 10 dígitos de ancho, con 2 decimales. Alineado a la izquierda.

En la función printf() también se pueden encontrar "caracteres de escape" que permiten intercalar algún carácter especial en la cadena.

Ejemplo:

```
printf("\n\aHola mundo.\n");
```

Aquí antes de imprimir el texto "Hola mundo", \n obliga a un salto de línea - retorno de carro, (ENTER) y \a hace sonar un "beep" en el speaker de la pc. Y luego de imprimir el texto, hace otro salto de línea - retorno de carro.

Caracteres de escape:

CÓDIGO	DESCRIPCIÓN
\n	Salto de línea – retorno de carro (ENTER)
\t	Tabulado horizontal
\v	Tabulado vertical
\a	Hace sonar el speaker, (sólo un beep).
\r	Retorno de carro.
\b	Backspace.
\f	Alimentación de página.
\'	Imprime una comilla simple.
\"	Imprime una comilla doble.
\\	Imprime una barra invertida, (\).
\xnnn	Notación hexadecimal
\nnn	Notación octal

FUNCIÓN scanf():

Esta función está asociada a la corriente de entrada "stdin".

El número de parámetros puede variar, pero el primero es una cadena que especifica los formatos de los datos a ingresar.

Ejemplo:

```
int a;
scanf("%i", &a);
```

Esto permite ingresar un valor para la variable a.

ATENCIÓN: Todas las variables usadas para recibir valores a través de scanf(), deben ser pasadas por sus direcciones, lo cual significa que todos los argumentos deben apuntar a las variables que los van a contener. La función scanf() necesita conocer la dirección de memoria de las variables para poder "cargar" el dato ingresado, es por eso que se coloca el símbolo **&** delante de las variables, ya que éste es un operador unario que precediendo al nombre de las variables indica que nos referimos a la dirección de la misma y no a su contenido. (en los temas "punteros" y "funciones" se termina de comprender satisfactoriamente este punto).

UNIDAD II

Estructuras De Control Y Comparación:

Sentencia if:

Forma general:

```
if (condición)
        sentencia1;
else
        sentencia2;
```

Si la condición es verdadera se ejecuta la sentencia1, de lo contrario la sentencia2.

Ejemplo:

Efectuar la división, sólo si el denominador es distinto de 0.

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

```
void main(void)
{
 float a, b;
 clrscr();
 printf("Ingrese numerador: ");
 scanf("%f", &a);
 printf("\nIngrese denominador: ");
 scanf("%f", &b);

 if (b!=0)  (*)
    printf("\nEl resultado es: %f", a/b);
 else
    printf("\nNo se puede dividir por cero");
}
```

(*) esta línea también se podría haber escrito: if (b), ya que una condición es verdadera para todo valor distinto de 0.

La condición en un if siempre debe ir entre paréntesis.
Si alguna de las ramas tiene más de una sentencia estas deben ir encerradas entre { }.

```
if (b)
  {
    r = a/b;
    printf("\nEl resultado es: %f", r);
  }
else
    printf("\nNo se puede dividir por cero");
```

También se pueden escalonar los if.

```
if (condición)
        Sentencia1;
else if (condición)
        Sentencia2;
else if (condición)
        Sentencia3;
else
        Sentencia4;
```

Y **anidar**:

```
if (x)
   if (a<b)
       printf("a es mayor");
   else
       printf("a no es mayor que b");
```

en caso que el else esté asociado al if(x) se debería hacer:

```
if (x)
  {
   if (a<b)
       printf("a es mayor");
  }
else
      printf("x es 0");
```

Sentencia switch:

La sentencia switch permite evaluar diferentes valores para una misma variable:
Su forma general es:

```
switch(variable)
{
   case valor1:
            sentencias;
    break;
   case valor2:
            sentencias;
    break;
   case valor3:
    sentencias;
    break;
      .
      .
      .
   default:
            sentencias;
```

```
}
```

El switch evalua cada caso, cuando coincide uno de ellos con el contenido de la variable, ejecuta las sentencias del caso y termina el switch. En caso de no encontrar ningún case que corresponda, en igualdad, con el contenido de la variable, ejecuta las sentencias de la cláusula default, si esta ha sido especificada, sino termina el switch.

Ejemplo:

El siguiente programa pide el ingreso de un valor y luego ofrece un menú de opciones para elegir que operación se desea relizar: averiguar el cuadrado del número, el cubo y si es par o no.

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

void main(void)
{
 int opcion, valor, res;
 clrscr();
 printf("Introduzca un valor entero mayor que 0:\n");
 scanf("%i", &valor);
 printf("\n\n");
 printf("*** MENU DE OPCIONES ***\n\n");
 printf("1 - Averiguar el cuadrado:\n");
 printf("2 – Averiguar el cubo:\n");
 printf("3 – Averiguar si es par o no:\n");
 printf("\nIngrese opción: ");
 scanf("%i", &opcion);
 printf("\n\n");
 switch(opcion)
 {
  case 1:
    re = valor*valor;
    printf("El cuadrado de %i es %i\n", valor, res);
    break;
  case 2:
    re = valor*valor*valor;
    printf("El cubo de %i es %i\n", valor, res);
    break;
  case 3:
    res = valor % 2;
    if (res)
      prinf("El número %i es impar\n", valor);
    else
      printf("El número %i es par\n", valor);
    break;
  default:
    printf("Opción erronea");
 }
}
```

El bucle for:

Forma general:

`for(`*`inicialización`*`; `*`condición`*`; `*`incremento`*`)`
`sentencia;`
	`for(`*`inicialización`*`; `*`condición`*`; `*`incremento`*`)`
	`{`
	`  sentencias;`
	`}`

El siguiente ejemplo muestra los primeros 100 números enteros:

```
#include <stdio.h>

void main(void)
{
 int i;
 for(i=1; i<=100; i++)
   printf("%d", i);
}
```

También puede funcionar al revés, (los primeros 100 enteros mostrados de 100 a 1);

```
for(i=100; i>=1; i - -)
    printf("%d", i);
```

Se pueden evaluar más de una variable:

```
int i, j;
for(i=1, j=100; i<=100, j>0; i++, j - -)
   printf("i = %d, j= %d\n", i, j);
```

El siguiente bucle no termina nunca, (bucle infinito):

```
for( ; ; )
```

Si la cantidad de sentencias, (equivalencias, operaciones aritméticas, llamadas a funciones, etc.), pertenecientes al bucle son más de una, estas deben ir entre { }.

El bucle while:

Forma general:

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

Ejemplo:

```
char salir;
salir = 'n';

while (salir != 'n')
{
  printf("Estoy dentro del mientras\n");
  scanf("%c", &salir);
}
printf("\nYa salí");
```

El bucle también puede estar vacío. El siguiente ejemplo funcionará hasta que se pulse la letra 'A':

```
while ((letra = getche()) != 'A');
```

El bucle do – while:

Forma general:

```
do
{
 sentencia;
} while (condición);
```

La diferencia con el while es que en do – while por lo menos el flujo del programa entra una vez al bucle y luego, (al llegar a la cláusula while), decide si continúa iterando.

Operaciones A Nivel De Bits, (Bitwise):

El lenguaje c proporciona la posibilidad de manipular los bits de un byte, realizando operaciones lógicas y de desplazamiento.

Operadores a nivel de bits:

OPERADOR	ACCIÓN
&	And entre bits
|	Or entre bits
^	Xor entre bits, (or exclusivo).
~	Not , (si es 1 pasa a ser 0 y viceversa)
<<	Desplazamiento a izquierda
>>	Desplazamiento a derecha

```
var << n = se desplaza n bits a la izquierda.
var >> n = se desplaza n bits a la derecha.
1 & 1 = 1
1 | 0 = 1, 0 | 1 = 1 y 1 | 1 = 1
1 ^ 0 = 1 y 0 ^ 1 = 1
~1 = 0 y ~0 = 1
```

De forma general conviene tener siempre presente estos resultados

```
X & 1 = X  ,      X & 0 = 0
X | 1 = 1  ,      X | 0 = X
X ^ 1 = ~X ,      X ^ 0 = X
```

Ejemplo de operaciones bit a bit:

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

void main(void)
{
    int n1 = 29;  //29 decimal es |0|0|0|1|1|1|0|1 en un byte
    int desp1, desp2, desp3, n1and, n1not, n1or, n2=18; //|0|0|0|1|0|0|1|0
    int n1xor;

    clrscr();
    printf("\t\t****** OPERACIONES A NIVEL DE BITS ******\n");

//************ DESPLAZAMIENTOS BIT A BIT
    desp1 = n1 << 3; //desplazo tres bits a la izquierda
                     //obtengo entonces |1|1|1|0|1|0|0|0 (232 decimal)
    printf("n1 = %d y desp1 = %d\n", n1, desp1);
```

```
    desp2 = n1 >> 3; //desplazo tres bits a la derecha
                     //obtengo |0|0|0|0|0|0|1|1 (3 decimal)
    printf("n1 = %d y desp2 = %d\n", n1, desp2);

    desp3 = n1 >> 2; //desplazo 2 bits a la derecha
                     //obtengo |0|0|0|0|0|1|1|1 (7 decimal)
    printf("n1 = %d y desp3 = %d\n", n1, desp3);

    //************ OPERACION AND
    n1and = n1 & n2; //|0|0|0|1|1|1|0|1 (29 decimal)
                   //& |0|0|0|1|0|0|1|0 (18 decimal)
                   //= |0|0|0|1|0|0|0|0 (16 decimal)
    printf("n1 = %d y n2 = %d ==> n1and = n1 & n2 es %d\n", n1, n2, n1and);

    //*********** OPERACION OR
    n1or = n1 | n2; //|0|0|0|1|1|1|0|1 (29 decimal)
                  //| |0|0|0|1|0|0|1|0 (18 decimal)
                  //= |0|0|0|1|1|1|1|1 (31 decimal)
    printf("n1 = %d y n2 = %d ==> n1and = n1 | n2 es %d\n", n1, n2, n1or);

    //********** OPERACION XOR
    n1xor = n1 ^ n2; //|0|0|0|1|1|1|0|1 (29 decimal)
                   //^ |0|0|0|1|0|0|1|0 (18 decimal)
                   //= |0|0|0|0|1|1|1|1 (15 decimal)
    printf("n1 = %d y n2 = %d ==> n1and = n1 ^ n2 es %d\n", n1, n2, n1xor);

    //********** OPERACION NOT
    /*Ojo que ~ (not) me cambia el bit de paridad lo que implica que cambia el
    signo, en este ejemplo me muestra -30 si pongo %d pero en realidad se
    trata del 226 cuyo caracter corresponde a -> â*/
    n1not = ~n1;  //~|0|0|0|1|1|1|0|1 (29 decimal)
                  //=|1|1|1|0|0|0|1|0 (226 decimal)
    printf("n1 = %d y ~n1 = n1not ==> %c\n", n1, n1not);
    getch(); //hace una pausa
}
```

UNIDAD III

Estructuras De Datos:

Arreglos (arrays):

Un array es una colección de elementos de un mismo tipo, que se referencian usando un nombre de variable común. En C, el array ocupa posiciones de memoria contiguas. La dirección más baja corresponde al primer elemento y la más alta al último. Para acceder a un elemento específico se usan índices.

Arrays unidimensionales:

Forma general de declararlos:

tipo nombre-variable[tamaño];

Ejemplo:

```
int numeros[10];
```

Es un arreglo de 10 elemento enteros, donde el primero es numeros[0] y el último numeros[9].

Guardemos en este array los números dígitos, (0 al 9):

```
for(i=0; i<=9; i++)
    numeros[i] = i;
```

Para averiguar el tamaño de un array se suele usar la función sizeof(), que devuelve un número que equivale al tamaño del array en bytes, (sizeof() se puede usar con cualquier variable).

```
BytesTotales = sizeof(tipo) * cantidad de elementos
```

En el ejemplo anterior:

```
BytesTotales = sizeof(int)*10 → 20
```

Ejemplo: Cargar una array de 20 números enteros y averiguar el promedio:

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

void main(void)
{
 int numeros[20], i;
 float promedio;
 clrscr();
 promedio = 0;
 for(i=0; i<20; i++)
     {
       printf("\nIntroducir un número: ");
       scanf("%d", &valor[i]);
     }

for(i=0; i<20; i++)
     promedio = promedio + valor[i];

printf("\nEl promedio es: %f", promedio/20);
}
```

ATENCIÓN: El lenguaje C no hace comprobación de límites, es decir que se pueden seguir ingresando valores, por encima del tamaño asignado al array, con consecuencias peligrosas, ya que puede grabar sobre el propio programa, ó aún peor, dañar el sistema operativo. El C no emite mensaje, ni en tiempo de compilación ni en tiempo de ejecución, para señalar este error. Queda en manos del programador asignar el tamaño necesario a los arrays que utilicen sus programas.

Otro ejemplo:

```
#include <stdio.h>

void main(void)
{
  char letras[7];
  int i;
  for(i=0; i<7; i++)
      letras[i] = 'A' + i;
}
```

Este programa genera un vector con las letras de la A a la G.

Cadenas (Strings):

En C no existe un tipo de datos específico para declarar cadenas, en su lugar la idea de cadena surge de un array de caracteres que siempre termina con el carácter nulo, (\0) y cuya posición debe contemplarse al dimensionar el array.

Para guardar una cadena de 10 caracteres:

```
char cadena[11];
```

Cuando se introduce una constante de cadena, (encerrada entre dobles comillas), no es necesario terminar con el carácter nulo, ya que el C lo crea automáticamente. Lo mismo sucede cuando se introduce una cadena desde el teclado, utilizando la función gets(), incluida en stdio.h, que genera el carácter nulo con el retorno de carro, (enter).

Ejemplo: Programa que muestra una cadena introducida desde el teclado:

```
#include <stdio.h>

void main(void)
{
 char cadena[100];

 printf("Ingrese cadena, de hasta 100 caracteres\n");
 gets(cadena);
 printf("Usted ingresó: %s", cadena);
}
```

FUNCIONES DE CADENAS, (archivo de cabecera string.h):

Se toma por convención que los strings terminan con el carácter nulo para estas funciones.

strcpy():

Se utiliza para copiar sobre una cadena:

```
strcpy(cadena, "Hola");
```

Guarda la constante "Hola" en la variable cadena.

ATENCIÓN: En C no se puede realizar entre cadenas la asignación cadena = "Hola" ya que recordemos que son arreglos de caracteres.

strlen():

Devuelve la cantidad de caracteres que posee la cadena, sin contar el carácter nulo.

```
a = strlen(cadena);
```

strcat():

Concatena dos cadenas.

```
strcat(cadena1, cadena2);
```

Resultado: cadena1 es la suma de cadena1 + cadena2

strcmp():

Compara los contenidos de dos cadenas.

```
strcmp(cadena1, cadena2),
```

Si son iguales devuelve 0.

Si cadena1 es mayor que cadena2: devuelve un valor mayor a 0.

Si cadena1 es menor que cadena2: devuelve un valor menor a 0.

Ejemplo:

Programa que compara un password:

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

void main(void)
{
 char clave[80];
 int x = 0;
 do
   {
     clrscr();
     if (x!=0)
       {
        printf("Intente nuevamente\n");
        printf("%d", x);
       }
     printf("Ingrese palabra clave\n");
```

```
        gets(clave);
     } while(x==strcmp("Abrete sesamo", clave));
}
```

Arrays Multidimensionales:

La declaración de un array de más de una dimensión se declara en forma general de la siguiente manera:

```
tipo nombre-variable[d1][d2][d3]...[dn];
```

Por ejemplo, un arreglo bidimencional de enteros de 5x3:

```
int m[5][3];   //5 filas x 3 columnas
```

Ejemplo: Programa que carga los números aleatorios del 1 al 50, en un array de 4 filas y 5 columnas.

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

void main(void)
{
 int mat[4][5], i, j;
 randomize();
 for(i=0; i<=4; i++)
      for(j=0; j<=5; j++)
    mat[i][j] = random(50);
 printf("\nLos valores de la matriz son: \n");
 for(i=0; i<=4; i++)
      for(j=0; j<=5; j++)
    printf("mat[%d][%d] = %d\n", i, j, mat[i][j]);
}
```

La cantidad de bytes utilizados por un array bidimensional se calcula de la siguiente forma:

```
Bytes = filas*columnas*sizeof(tipo)
```

Inicialización De Arrays:

Cuando se declara un array, también se le puede asignar valores iniciales.

Ejemplos:

```
int numeros[10] = {1,2,3,4,5,6,7,8,9,10};
char cadena[6] = "hola";
char cadena[5] = {'h', 'o', 'l', 'a', '\0'};
int matriz[4][2] = {1,1,2,4,3,9,4,16};
```

La Sentencia *#define:*

Esta sentencia permite declarar constantes y es muy usada para definir los límites de los arreglos:

```
#include <stdio.h>
#define FILA 5  //define una constante FILA cuyo valor es 5
#define COL 10 // define una constante COL cuyo valor es 10

void main(void)
{
 int m[FILA][COL];
 int x, y;
 for(x=0; x<FILA; x++)
    for(y=0; y<COL; y++)
    scanf("%d", %m[x][y]);
}
```

Punteros:

Dirección de Memoria

La memoria de una computadora se representa habitualmente mediante direcciones expresadas en valores hexadecimales. Una variable, por ejemplo x, es una dirección de memoria donde se guarda algún valor. En los lenguajes de programación uno usa identificadores, (nombres simbólicos), para reservar lugares de la memoria necesarios para almacenar datos que permitan realizar operaciones. Estos lugares son tan amplios como el tipo de dato en que fue declarada la variable, esto es:

Si declaro una variable x de tipo entera, (int), estaría reservando 2 bytes en memoria, si declaro una de tipo char sería 1 byte, etc.

En ciertas implementaciones de sistema operativo, como el DOS, la memoria se asigna utilizando un método de compatibilidad con las PC XT, que utiliza dos WORDS (16 bits) para representar una dirección de 20 Bytes y alcanzar el MegaByte de memoria.

Direcciones lejanas (far):

Se codifican como dos enteros de dos bytes cada uno, un segmento y un desplazamiento (offset). La dirección efectiva surge de la operación:

```
SEGMENTO * 16 + OFFSET
```

que, empleando notación hexadecimal se puede expresar como una suma en la que el segmento se desplaza una posición a la izquierda:

```
    SSSS0
+   dddd
    xxxxx
```

La dirección efectiva (xxxxx) va, entonces, de 0 a 0XFFFFF, o sea, de 0 a 1 Mb.

En general cuando debemos indicar al programa en C que se debe alcanzar una dirección de memoria absoluta (independiente del programa) para leer o modificar el contenido debe obligatoriamente utilizarse un puntero far.

Direcciones cercanas (near):

Se codifican en dos bytes (en hexadecimal, bastan 4 dígitos). Se trata de una dirección para la cual el valor del segmento se supone dado. Por ejemplo para acceder a una posición de memoria dentro de la memoria de trabajo asignada al programa.

Esto implica que en una implementación PC, C manejará dos tipos de punteros, justamente far (de cuatro bytes, segmento + desplazamiento) y near (de dos bytes, desplazamiento solamente).

Punteros

Un puntero es una variable de algún tipo que permite guardar una dirección de memoria, esto permite, indirectamente, acceder al contenido de esa dirección.

Un puntero se declara de forma general de la siguiente manera:

```
tipo *nombre-variable;
```

Ejemplo:

```
int *p;
char *q;
```

Veamos un pequeño programa que ilustre las cualidades de los punteros:

```
#include <stdio.h>

void main(void)
{
  int *p;  (1)
  int x = 5;  (2)

  p = &x;   (3)

  printf("El valor de x es: %d\n", x);  (4)
  printf("p apunta a x, entonces en x hay: %d", *p);  (5)
}
```

Este ejemplo dará como resultado la visualización del contenido de la variable x, o sea 5, dos veces.

Veamos línea X línea:

En (1) se declara un variable puntero, p, a los enteros y en (2) una variable x entera con el valor inicial de 5.

En (3) se almacena en el puntero, p, la dirección de memoria de x; el operador & permite ver la dirección de memoria de una variable.

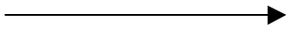
Luego en (4) se muestra el contenido de x.

Como en (3) se guardó la dirección de x en p, éste, que es un puntero, tienen la cualidad de "ver" el contenido de esa dirección de forma indirecta por medio del operador *. Esto es lo que se hace en (5), ver el contenido de x, pero desde el puntero, ya que él tiene almacenada la dirección de memoria de x. Por eso se visualiza dos veces el contenido de x.

Supongamos que las variables son celdas en memoria, dentro de cada celda se almacenan los valores y al pié de ellas está la dirección de memoria de la misma.

Al declarar las variables:

Al hacer p = &x:



p apunta a x

p= 0xbb x = 0xff

Como la variable-puntero, p, tiene almacenada la dirección de x, haciendo *p se puede ver el valor almacenado allí.

No es algo muy complejo, y aparentemente inútil, pues, ¿para qué quiero acceder con un puntero al contenido de otra variable si lo puedo hacer con la misma variable?.

El tema es que, la capacidad de un puntero de acceder al contenido de diferentes lugares de memoria, lo hace extremadamente útil en muchísimos casos.

Recuerde, por ejemplo, que los sistemas operativos reservan diferentes direcciones de memoria para, por ejemplo, manipular los puertos de e/s de la pc, para guardar ciertos datos útiles, (la versión del SO), etc; sólo con un puntero uno puede acceder a tales lugares.

Ejemplos de utilización de puntero far:

Un ejemplo de manejo directo de un puntero se tiene apuntándolo sobre la RAM de video. Obviamente se tratará de un puntero far; el objeto apuntado dependerá de lo que deseemos hacer. En este caso, lo definiremos como un char; en modo texto cada objeto contendrá el carácter, para las direcciones pares o el atributo para las impares:

```
#include <stdio.h>
void main(void)
{
 char far * video;
 video= (char far *) 0xB8000000;     /* B800:0000 */
 *video= 65;
}
```

La segunda instrucción carga el puntero, apuntándolo al buffer de video color (para el monocromo, la constante debería ser 0xB0000000). Observe que codificamos la dirección en la forma segmento+desplazamiento (corresponde a B800:0000), y que para evitar un mensaje de advertencia forzamos la conversión de tipo (de entero a puntero) declarando entre paréntesis el tipo esperado.

La tercera instrucción pone el 65 (ASCII de la letra "A") en la posición apuntada por video, que es el primer carácter de pantalla.

Otro: El programa BIOS está grabado en una memoria eprom cuya dirección final coincide con el Mega de memoria, los 8 bytes anteriores a los 2 bytes finales almacenan la fecha de grabación del programa en la memoria, ya que la eprom se comporta como una ROM eso nos dará una idea de cuan antigua es nuestra PC.

```
#include <stdio.h>
void main(void)
{
  char far *pBIOS= (char far *) 0xFFFF0005;
  short int i=0;

  for( ; i<8; i++)
    putchar(*(pBIOS+i));
}
```

Arreglos y punteros:

Otra cosa a tener en cuenta es que:

"El nombre de un array es un puntero al primer elemento del mismo."

Esto implica que se puede usar sólo el nombre del array, (sin los []), para poder moverse a través de sus elementos, vea el siguiente ejemplo:

```
/*RECORRIDO DE UN VECTOR USANDO LA NOTACION ESTANDAR
Y MEDIANTE LA DEFINICION DE PUNTEROS*/
#include <stdio.h>
#include <conio.h>

void main(void)
{
    int v[5] = {10, 5, 3, 8, 9};  //se define un array de 5 elementos enteros
    int i=0;
    clrscr();
    for (i=0; i<=4; i++)
          printf("\nPosicion %d de v contiene: %d", i, v[i]);
    printf("\n");
    getch();
    i=0;
    do
      {
          printf("\nUsando notacion de punteros v: %d", *(v + i));
          i++;
      }
    while (i <=4);
    getch();
```

```
}
```

En el bucle for se muestran los contenidos de las diferentes posiciones del array de la forma tradicional, o sea, usando los corchetes con el índice, haciendo v[i].

Pero en el bucle do, se aprovecha la definición, que dice que el nombre de un array es un puntero al primer elemento, para hacer lo mismo, utilizando *(v+i).

```
*(v+0) equivale a v[0]
*(v+1) equivale a v[1]
*(v+2) equivale a v[2]
*(v+3) equivale a v[3]
*(v+4) equivale a v[4]
```

Estructuras:

Una estructura es un conjunto de variables de diferentes tipos referenciadas bajo el mismo nombre.

Ejemplo:

```
struct empleado
{
 char nombre[30];
 int edad;
 float sueldo;
}
```

De esta forma se define una estructura llamada empleado, ahora hay que declarar una variable de este tipo:

```
struct empleado e;
```

En este caso la variable e es de tipo empleado y se pueden acceder a los campos miembros de la estructura de la siguiente forma:

```
e.nombre
e.edad
e.sueldo
```

Ejemplo:

Programa que carga por teclado y muestra una variable de tipo empleado:

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

struct empleado
{
 char nombre[30];
 int edad;
 float sueldo;
}

void main(void)
{
 struct empleado e;
 clrscr();
 //Ingreso de datos
 printf("Ingrese nombre: ");
 gets(e.nombre);
 printf("\nIngrese edad: ");
 scanf("%i", &e.edad);
 printf("\nIngrese sueldo: ");
 scanf("%f", &e.sueldo);
 printf("\n\n");
 //Se muestran los datos

 printf("Nombre: %s\n", e.nombre);
 printf("Edad: %i\n", e.edad);
 printf("Sueldo: %4.2f", e.sueldo);
 getch();
}
```

Array De Estructuras:

Una variable de de tipo estructura como la del ejemplo anterior permite almacenar los datos de sólo un empleado, qué tal si se desea almacenar los datos de los 10 empleados de la empresa.

Para eso hay que declarar un arreglo de estructuras:

El siguiente ejemplo tiene la misma función que el anterior pero para los 10 empleados de la empresa:

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

struct empleado
{
 char nombre[30];
 int edad;
 float sueldo;
}
```

```
void main(void)
{
 struct empleado e[10];  //Array e de 10 posiciones de tipo empleado
 int i;
 clrscr();
 //Ingreso de datos
 for(i=0; i<10; i++)
     {
       printf(“Ingrese nombre: “);
       gets(e[i].nombre);
       printf(“\nIngrese edad: “);
       scanf(“%i”, &e[i].edad);
       printf(“\nIngrese sueldo: “);
       scanf(“%f”, &e[i].sueldo);
       printf(“\n\n”);
     }
 //Se muestran los datos.
 for(i=0; i<10; i++)
    {
     printf(“Nombre: %s\n”, e[i].nombre);
     printf(“Edad: %i\n”, e[i].edad);
     printf(“Sueldo: %4.2f”, e[i].sueldo);
    }
 getch();
}
```

UNIDAD IV

Funciones:

Las funciones son porciones de código que facilitan la claridad de desarrollo del programa.

Todas las funciones retornan un valor y pueden recibir parámetros.

La estructura general de un función en C es la siguiente:

```
Tipo_de_retorno nombre_función (tipo param1, tipo param2, ..., tipo paramn)
{
 sentencias
 return(valor_de_retorno);
}
```

Los posibles tipos de retorno son los tipos de datos ya vistos: (int, float, void, char,etc).

Para crear una función en C, primero hay que declarar el prototipo de la misma antes de la función main() y luego de la llave final del programa se define la función.

Ejemplo:

Programa con una función que recibe 2 parámetros enteros y retorna la suma de los mismos:

```
#include <stdio.h>

int suma(int x, int y);    //prototipo de la función

void main(void)
{
 int a, b;
 printf("Ingrese valor de a: ");
 scanf("%i", &a);
 printf("\nIngrese valor de b: ");
 scanf("%i", &b);

 printf("\nLa suma de a y b es: %i", suma(a,b));
}
//Ahora viene la definición de la función
int suma(int x, int y)
{
 return x+y;
}
```

Se retorna de una función cuando se llega a la sentencia return o cuando se encuentra la llave de cierre de la función.

Cuando lo que se desea escribir es un procedimiento que, por ejemplo, realice un dibujo o muestre un texto por pantalla o cargue una arreglo, o sea, que no devuelva ningún valor se escribe como tipo de retorno void, (tipo vacío).

El siguiente programa consta de una función que se encarga de cargar un arreglo de caracteres:

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

void carga(void);

char v[10];

void main(void)
{
 int i;
 clrscr();
 carga();  //llamo a la función que carga el arreglo
 for(i=0; i<10; i++)
    printf("%c, ", v[i];
}
//definición de la función
void carga(void)
{
 int i;
 for(i=0; i<10; i++)
   v[i] = getche();    //getche() permite ingresar un caracter mostrándolo
además por pantalla, (eco).
}
```

En este caso la función se comporta como un procedimiento, por eso carece de la sentencia return, que estaría de más pues el retorno es void.

Ámbito de las variables:

Variable global:

Conocida por todas las funciones. Se puede utilizar en cualquier punto del programa. Se declara fuera del main.

Variable local:

Se declara apenas abrir una llave en el código, cuando la llave se cierra esta variable desaparece.

Variable declarada en los parámetros formales de una función:

Tiene el mismo comportamiento de las variables locales.

Paso De Parámetros:

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.

```
#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.

ATENCIÓN: Los arrays, (entiéndase también cadenas), siempre se pasan por referencia y no hace falta anteponerle el símbolo &, pues como habíamos dicho el nombre de un array es un puntero al primer elemento del mismo.

Asignación Dinámica De Memoria:

Existen dos funciones de librería, que utilizan el archivo de cabecera stdlib.h, son: malloc() y free(), que permiten asignar de forma dinámica, (o sea en tiempo de ejecución), memoria.

Hasta ahora hemos visto que uno consigue memoria para su programa sólo con las variables declaradas, cosa que ocurre en tiempo de compilación, durante la ejecución hay que limitarse a esas cantidades de memoria, (un par de enteros por acá, otros bytes por variables char declaradas por allá, etc).

Cuando lo que se quiere es asignar memoria de forma dinámica, uno debe primero pensar en la utilización de un puntero que almacene la primer dirección del segmento de memoria a asignar y que luego permita recorrerlo.

malloc()

es la función encargada de asignar el segmento de memoria necesario. Necesita un parámetro que le indique que cantidad de memoria se desea asignar. Retorna un puntero a void que es preciso convertir al tipo de

puntero que estemos utilizando, (ejemplo: si se desea asignar 100 bytes de memoria se podría declarar un puntero a char y luego usar malloc() especificando que son 100 bytes y que se lo asignamos al puntero a char, precisamente deberemos convertir a este último tipo el retorno de malloc()).

Si por algún motivo no se puede asignar la cantidad de memoria requerida, el puntero tomará el valor NULL.

free()

libera la memoria asignada. Su único parámetro es el puntero utilizado.

El siguiente ejemplo asigna memoria para 10 enteros, (10 * 2 bytes = 20 bytes).

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

void main(void)
{
 int *ent, x;
 ent = (int*)malloc(10*sizeof(int));   //Se realiza la asignación
 if(ent==NULL)
    printf("Falta memoria\n");
 else
    {
     for(x=0; x<10; x++)
           *(ent+x)=x;    //se almacena en esa porción de memoria 10 enteros
     for(x=0; x<10; x++)
           printf("%d\n", *(ent+x));  //se muestran los contenidos de la
memoria asignada.
     free(ent);
     }
 getch();
}
```

Una aplicación de esto son las listas dinámicas. Listas a las cuales se le puede ir asignando continuamente memoria para que pueda almacenar un elemento más.

Si en el ejemplo anterior, en lugar de tener una cantidad fija de memoria a asignar, (10), se pide el ingreso del tamaño y se lo almacena en una variables n; reemplazando donde dice 10 por n obtendría un array dimensionado en cada ejecución.

EJERCICIO INTERESANTE: Pruebe de hacer un programa que asigne un char, (1 byte), de memoria sucesivamente hasta que el puntero sea NULL y cuente cuantas asignaciones hizo. Por fin de programa muestre el contador. Habrá averiguado cuanta memoria, aproximadamente, dispone el sistema operativo para su Programa.

Funciones De Video:

El archivo graphics.h nos ofrece una amplia cantidad de funciones para graficar, además de detectar nuestro tipo de placa de video, (VGA, EGA, CGA, etc). Borland C/C++ 3.1 proporciona una extensa lista de drivers para placas, (archivos .bgi), localizados en el directorio BGI.

Para usar las funciones gráficas, primero se debe inicializar dicho modo gráfico.

Ejemplos:

El siguiente ejemplo inicializa el modo gráfico, averigua cuantos colores soporta la placa y muestra este valor con letras góticas.

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

void main(void)
{
    int placa, modo, e, mc;
    char salida[80];
    placa=VGA;
    modo =VGAHI;
    // se inicializa la placa de video
    initgraph(&placa, &modo, "vga.bgi");   (1)

    // se verifica el intento de inicialización
    e = graphresult();   (2)
    if (e != grOk)   (3)
    {
          printf("Error al inicializar la placa de video: %s\n",
grapherrormsg(e));
          getch();
          exit(1);
    }
    mc=getmaxcolor();  (4)
    settextstyle(GOTHIC_FONT, HORIZ_DIR, 5);  (5)
    sprintf(salida, "Cantidad de colores: %i", mc);   (6)
    outtextxy(200, 200, salida);   (7)
    getch();
    closegraph();   (8)
```

```
}
```

En la línea (1) se llama a la función initgraph() a la cual se le pasa las constantes VGA para la tarjeta, (CGA si fuera necesario) y VGAHI para el modo, (podría haber sido VGALO, por ejemplo). El tercer parámetro es la localización del archivo .bgi necesario.

Luego, (2), usando graphresult() inmediatamente después de llamar a initgraph(), se obtiene el resultado del intento de inicialización del modo gráfico. El cual se verifica en (3) si es distinto de grOK, (constante predefinida), de ser así se muestra un mensaje de error y se termina la ejecución.

Si se pudo inicializar el modo gráfico, se obtiene la cantidad de colores con getmaxcolor() en (4), se prepara el estilo del texto, en (5), con settextstyle(), indicando que será una fuente gótica, de forma horizontal y de tamaño 5. Borland provee un par de fuentes.

Con la función sprintf(), en (6), se formatea el texto a mostrar, (ya que no se puede mostrar información con printf() durante el modo gráfico), y se la guarda en la variable salida.

La función outtextxy() es la que finalmente muestra el texto en las coordenadas, (pixeles), deseadas.

Se cierra el modo gráfico en (8) con la función closegraph().

En modo gráfico se pueden realizar interesantes gráficos usando funciones como:

ellipse(), circle(), arc(), line(), putpixel(), rectangle(), etc.

El siguiente programa hace uso de las funciones circle() y arc(). Mientras no se pulse un tecla, en un ciclo for que varía los radios de las figuras, va dibujando un círculo y arcos de diferentes colores.

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

void main(void)
{
    int placa, modo, e, mc, i,j;
    char salida[80];
    placa=VGA;
    modo =VGAHI;
    /* se inicializa la placa de video*/
    initgraph(&placa, &modo, "egavga.bgi");

    /* se verifica el intento de inicialización */
    e = graphresult();
    if (e != grOk)
    {
           printf("Error al inicializar la placa de video: %s\n",
grapherrormsg(e));
           getch();
           exit(1);
    }


    while(!kbhit())   //Mientras no se pulse un tecla
           {
            setbkcolor(1);   //Color de fondo
            for(i=1;i<=240;i++)
                 {
                  setcolor(6);   //color de primer plano
                  circle(320, 240, i);
                  setcolor(5);
                  arc(320, 480, 0, 180, i);
                  setcolor(7);
                  arc(320, 0, 180, 360, i);
                  setcolor(9);
                  arc(640, 240, 90, 270, i);
                  setcolor(11);
                  arc(0, 240, 270, 90, i);

                 }
           cleardevice();
      }
    getch();
    closegraph();
}
```

Mire la ayuda de Borland C/C++ 3.1 para el archivo de cabecera graphics.h, para más información sobre funciones y constantes.

Paso De Argumentos Desde La Línea De Comandos:

La función main() también puede recibir parámetros, esto ocurre cuando se ejecuta el programa desde la línea de comandos, (el prompt del DOS), agregándole parámetros, (un ejemplo de esto es el format.com, que cuando uno lo ejecuta debe, entre otras cosas, pasarle como parámetro la unidad).

Si se va a escribir un programa que tenga en cuenta los parámetros pasado desde la línea de comandos, hay que agregar entonces dos cosas a la definición de la función mai().

```
void main(int argc, char *argv[])
```

El retorno no cambia, pues no pretendo retornar nada por fin de programa, ahora la función recibe dos parámetros:

int argc:

Es un contador que indica la cantidad de parámetros escritos al momento de ejecutar la aplicación, (ATENCIÓN: que cuenta también el nombre del programa). Supongamos que nuestro ejecutable es PEPE.EXE y se lo ejecuta de la siguiente forma:

```
C:\pepe hola
```

El parámetro argc es 2, (PEPE.EXE y hola).

```
pepe hola mundo cruel
```

El parámetro argc es 4, (PEPE.EXE, hola, mundo y cruel).

char *argv[]:

Es un array de punteros a cadenas de caracteres. Cada posición del array almacena el parámetro pasado, siendo argv[0] el nombre del ejecutable, incluyendo el path. Supongamos nuevamente que nuestro ejecutable es pepe.exe, que se encuentra en el directorio Windows, y se lo ejecuta de la siguiente forma:

```
C:\pepe hola
```

El parámetro argc es como dijimos 2, y en argv[0] hay “c:\windows\pepe.exe” y argv[1] = “hola”.

Ahora veamos un ejemplo.

El siguiente programa espera 3 parámetro, (en realidad son 4 contando el ejecutable), a la hora de ejecutarlo.

El programa se llama argcalc. Exe y los parámetros son por ejemplo: 3 + 8 ó 5 * 9, etc. O sea operaciones aritméticas básicas entre enteros. Si uno escribe: argcalc 6 + 4, el programa debería dar la siguiente salida:

La suma es: 10.

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


void main(int argc, char *argv[])
{
 int a, b;
 clrscr();
 if (argc < 2) //se chequea la cantidad de argumentos.
    printf("\nNo hay argumentos para %s\n", *argv);
 else if (argc==4)       //0,1,2 y 3
    {
     if (strcmp(argv[2], "+") == 0)
           {
            a=atoi(argv[1]);
            b=atoi(argv[3]);
            printf("\nLa suma es: %d", a + b);
           }
     else if (strcmp(argv[2],"-")==0)
           {
            a=atoi(argv[1]);
            b=atoi(argv[3]);
            printf("\nLa resta es: %d", a - b);
           }
     else if (strcmp(argv[2], "*")==0)
           {
            a=atoi(argv[1]);
            b=atoi(argv[3]);
            printf("\nLa multiplicaci¢n es: %d", a * b);
           }
     else if (strcmp(argv[2], "/")==0)
           {
            if (strcmp(argv[3], "0") !=0)
                 {
                  a=atoi(argv[1]);
                  b=atoi(argv[3]);
                  printf("\nLa divisi¢n es: %d", a / b);
                 }
            else
                 printf("\nNo se puede dividir por 0");
           }
    }
```

```
  getch();
 }
```

Básicamente lo que se hace es verificar los argumentos pasados. En este caso si la cantidad esperada es distinta de 4, obviamente no se puede hacer la operación y se informa el error.

Una vez que la cantidad de argumentos es la correcta se verifica que en argv[2] haya un “+”, “-“, “*” ó “/” y se realiza la operación, convirtiendo los contendidos de argv[1] y argv[3] a enteros, (ya que son cadenas dentro del array), por medio de la función atoi(), (array to integer).

UNIDAD V

Archivos:

En C se asocian corrientes, (stream), con archivos, esto es, se crean corrientes y estas se enlazan a archivos que se grabarán ó leerán desde un disco.

Para manipular archivos en C, se declara una variable puntero de tipo FILE, cuyas características son incluir el nombre del archivo, su estado y posición actual. La corriente asociada al archivo, lo utiliza para dirigir cada una de las funciones de memoria intermedia de E/S, al lugar donde se realizan las operaciones.

Declaración:

```
FILE *nombre-variable;
```

Ejemplo: FILE *pf;

Apertura de un archivo:

Se utiliza la función fopen(), que cumple 2 funciones:

- Abre una corriente y enlaza el archivo con la misma.
- Devuelve el puntero al archivo, asociado con ese archivo. Si aparece un error devolverá NULL.

```
FILE *archi;

archi = fopen("a:\miarchivo.dat", "w");
if(archi ==NULL)
  {
   printf("No se pudo abrir el archivo");
   exit(1);
  }
```

En fopen() el primer parámetro es el path completo del archivo a abrir y el siguiente es el tipo y modo de apertura, (archivo de texto, escritura).

Modos de aperturas:

MODO	SIGNIFICADO
"r"	Abre en modo sólo lectura, (debe existir el archivo)
"w"	Abre para grabar, (si no existe lo crea sino lo sobreescribe)
"a"	Abre para agregar datos, (debe existir el archivo)
"rb"	Abre archivo binario para lectura
"wb"	Abre archivo binario para escritura
"ab"	Abre archivo binario para agregar
"r+"	Abre archivo como lectura/escritura
"w+"	Crea o abre para lectura/escritura
"a+"	Crea o abre para lectura/escritura
"r+b"	Ídem "r+" pero para binario.
"a+b"	Ídem "a+" pero para binario.
"w+b"	Ídem "w+" pero para binario.
"rt"	Abre archivo de texto para lectura
"wt"	Abre archivo de texto para escritura
"at"	Abre archivo de texto para agregar
"r+t"	Abre archivo de texto para lectura/escritura
"w+t"	Crea archivo de texto para lectura/escritura.
"a+t"	Crea o agrega, (lectura/escritura) para archivos de texto.

Funciones para manipular archivos:

Funciones feof() y fclose():

La función feof() detecta el fin de archivo, (de texto ó binario). Devuelve verdadero cuando ha alcanzado el fin de archivo, de lo contrario devuelve 0.

```
while(!feof(archi))
      .....
```

La función fclose() cierra el archivo:

```
fclose(archi);
```

Lectura y escritura en archivos:

Función fwrite():

Permite escribir datos en un archivo de texto o binario.

```
int n;
```

```
n=5;
fwrite(&n, sizeof(n), 1, archi);
```

Parámetros:

Primero: dirección de memoria donde se encuentra almacenada la información a escribir.

Segundo: tamaño de la variable a escribir, (en el ejemplo se podría haber puesto simplemente 2).

Tercero: Cantidad de ítems escritos, (habitualmente es 1).

Cuarto: La variable FILE.

La función devuelve un valor, (que si a uno le resulta necesario lo utiliza), que es la cantidad de ítems escritos.

Ejemplo:

supongamos que queremos grabar un archivo con la información de amigos, para eso usaremos un archivo de estructuras.

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

struct registro
{
  char nom[20];
  char ape[20];
  char tel[15];
};

void main(void)
{
FILE *archi;
struct registro agenda;
char salir;
// se abre "amigos.dat"
if ((archi = fopen("AMIGOS.DAT", "a+b")) == NULL)
{
printf("No se pudo abrir el archivo.\n", stderr);
exit(1);
}
//Se pide el ingreso de datos
while (salir!='s' )
      {
       fflush(stdin);  //Ver nota
       printf("\nIngrese nombre: ");
       gets(agenda.nom);
       printf("\nIngrese apellido: ");
       gets(agenda.ape);
       printf("\nIngrese nro. de tel,fono: ");
       gets(agenda.tel);
       //Se escriben los datos en el archivo
       fwrite(&agenda, sizeof(agenda), 1, archi);
       salir = getch();
     }
fclose(archi);
}
```

NOTA: La función fflush() permite limpiar el buffer de entrada, (stdin), en este caso el teclado. Esta función se debe usar cada vez que se intercalan ingresos con scanf() o getche() junto con gets(), ya que las primeras funciones descartan el carácter de fin de línea – retorno de carro, producido al pulsar ENTER, el mismo queda varado en el buffer de entrada y si de repente ocurre una llamada a una función como gets(), (que precisamente como permite ingresar cadenas lee una pulsación de ENTER ya que con eso genera el carácter '\0' para el fin de la cadena), ésta toma ese carácter varado en el buffer y da por terminada la cadena, sin haber ingresado aún carácter alguno.

Función fread()**:**

Permite leer datos de un archivo de texto o binario.

```
int n;
fread(&n, sizeof(dato), 1, archi);
```

En este caso lee un valor entero y lo almacena en n.

Parámetros:

Primero: dirección de memoria donde se almacena lo leído.

Segundo: tamaño de lo leído, (en el ejemplo se podría haber puesto simplemente 2).

Tercero: Cantidad de ítems leídos, (habitualmente es 1).

Cuarto: La variable FILE.

La función devuelve un valor, (que si a uno le resulta necesario lo utiliza), que es la cantidad de ítems leídos.

Ejemplo:

El siguiente programa permite leer la información ingresada en el ejemplo anterior.

```
#include <stdio.h>
#include <conio.h>
#include <process.h>
struct registro
{
  char nom[20];
  char ape[20];
  char tel[15];
};

void main(void)
{
FILE *archi;
struct registro agenda;
// se abre "amigos.dat"
archi = fopen("AMIGOS.DAT", "rb");
clrscr();
if ((archi == NULL))
{
 printf("No se pudo abrir el archivo.\n", stderr);
 exit(1);
}
//Leo desde el primer registro hasta fin de archivo
fread(&agenda, sizeof(agenda), 1, archi);
while (!feof(archi))
  {
    printf("\n-->%s\t\t%s\t\t%s", agenda.nom, agenda.ape, agenda.tel);
    fread(&agenda, sizeof(agenda), 1, archi);
  }

fclose(archi);
getch();
}
```

El programa lista todos los registros del archivo.

Funciones ferror() y rewind():

ferror() devuelve verdadero si hubo error y falso si no lo hubo. Puede utilizarse después de cada operación con un archivo.

```
ferror(archi);
```

rewind() vuelve al comienzo del archivo.

```
rewind(archi);
```

Función fseek()**:**

Esta función permite posicionarse en determinado registro, (o dato), dentro del archivo.

```
fseek(archi, offset, desde)
```

archi = la variable FILE.

offset = la diferencia en bytes entre el parámetro desde y la nueva posición.

desde = desde dónde comienza el fseek(), los posibles valores son:

- SEEK_SET = desde el principio del archivo.
- SEEK_CUR = desde la posición actual.
- SEEK_END = desde el final del archivo.

Función ftell()**:**

Retorna la posición actual del puntero dentro del archivo.

Si el archivo es binario el retorno es en bytes tomados desde el comienzo del archivo.

ftell() se suele usar junto con fseek().

```
long pos;
pos = ftell(archi);
```

Función remove()**:**

Elimina un archivo:

```
remove(char *nombrearchivo);
```

El Lenguaje C Y El Sistema Operativo:

El lenguaje C permite realizar una extensa gama de actividades propias del sistema operativo. En librerías como BIOS.H, DOS.H, DIR.H entre otras se pueden encontrar funciones que permiten desde un sencillo cambio de la hora y la fecha del sistema, pasando por obtener información de las unidades de disco, información de la memoria, etc.

A modo de ejemplo se presentarán dos programas que ilustran algunas de estas características:

El primero envía datos al puerto de comunicaciones, (COM 2), con la idea de marcar un número de teléfono en caso que exista un modem conectado allí.

El segundo lista todos los archivos del disco donde fue ejecutado el programa, (si el programa es ejecutado en la unidad C mostrará todos los archivos que allí se encuentren).

Ejemplo1: Enviar datos al puerto de comunicaciones:

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

void main(void)
{
int puerto = 1;
int i;
char dato[12] = {'A','T','D','T','4',’5','5','4','4','0','4','4'};
for (i=0; i<=11; i++)
    outportb(puerto, dato[i]);

printf(“%s ha sido enviado al puerto  %d\n", dato, puerto);
}
```

La función utilizada aquí es outportb() que permite enviar un byte, (carácter), al puerto especificado en el parámetro puerto, (0 - COM1, 1 - COM 2). Como lo que se pretende enviar es una cadena con el comando “ATDT99999” que le indica al modem que disque tal número, se prepara un array de caracteres y se envía elemento por elemento del mismo.

La función compañera de esta es inportb() que lee un byte del puerto de comunicaciones.

La estructura ffblk:

Ejemplo2: Listar todos los archivos de la unidad actual.

Este programa hace uso de la estructura ffblk, que C incorpora, (archivo de cabecera DIR.H). Esta estructura almacena la información de un archivo, (atributos, fecha, hora, tamaño, etc.), cada vez que se localiza alguno usando las funciones findfirst(), findnext(), también disponibles agregando DIR.H

```
#include <stdlib.h>
#include <fcntl.h>
#include <io.h>
#include <stdio.h>
#include <dir.h>
#include <conio.h>
#include <dos.h>
#include <string.h>

// Estas son las librerías necesarias para la ejecución de este programa

void entradir(void); // Rutina de entrada a directorios

void mostrar(void); // Rutina para mostrar los archivos del directorio donde
entro

char uarchi[13];
int raiz=1;
void main(void)
{
  clrscr();
  chdir("\\");
 // es necesario poner 2 barras "\\" ya que el C usa la "\" para funciones
 // especificas como las que se ven arriba de "\n" eso manda un CR LF
 // caracteres 10 y 13 en decimal o Ah y Dh en hexa de la tabla ASCII
  entradir(); // Llamada a entradir
}


void mostrar(void)
{
struct ffblk ffblk;
int listo, f=2;
char sale;
if (raiz)
  {
   printf("\n*** Directorio: Raiz ***\n");
   raiz=0;
  }

// Busco en el directorio actual "*.*" incluyendo archivos ocultos
listo = findfirst("*.*",&ffblk,FA_HIDDEN); //retorna 0 si encuentra
// Mientras haya archivos..

while (!listo)  //mientras no haya terminado sigue listando
 {
  if (strcmp(uarchi, ffblk.ff_name)!=0)
     {
    printf("\n %s - %ld",ffblk.ff_name, ffblk.ff_fsize);  //muestro nombre y
tamaño
```

```
    strcpy(uarchi, ffblk.ff_name);
    f++;
    if (f==23)
    {
            delay(2000);  //ó getch() aquí para hacer una pausa entre cada
pantalla.

     f=2;
     clrscr();
    }
  }
listo = findnext(&ffblk);       // Que pase el que sigue
 }
}

void entradir(void)
{
struct ffblk ffblk;
int listo;

mostrar();  // Llamo a mostrar para mostrar los del directorio raíz

// Busco en el directorio actual "*.*" incluyendo directorios
listo = findfirst("*.*",&ffblk,FA_DIREC); //también 16
strcpy(uarchi, "");
while (!listo)
 {
//16
if((ffblk.ff_name[0] != *".") && (ffblk.ff_attrib == FA_DIREC))
  // Si es un directorio.. que no sea que su primer caracter sea "." o ".."
entonces entro
   {
   chdir(ffblk.ff_name); // Entro al directorio que encontre
   printf("\n*** Directorio: %s ***\n", ffblk.ff_name);
   mostrar();  // muestro los archivos
   entradir();   // Me llamo de nuevo
   chdir("..");   // Vuelvo al directorio anterior

   }
  listo = findnext(&ffblk); // Busco el próximo

  }
}
```

Estructura ffblk, (file control block structure):

```
 struct ffblk {
     char ff_reserved[21];  //reservado para DOS
     char ff_attrib;       // atributo
     int  ff_ftime;        // hora
             int  ff_fdate;        // fecha
     long ff_fsize;        //tamaño
     char ff_name[13];     // nombre
  };
```

En el programa básicamente lo que se hace es, por medio de findfirst() y/o findnext() ubicar cada archivo o directorio, si es un archivo se muestra el nombre y tamaño, si se trata de un directorio, (dato arrojado por el campo ff_attrib de ffblk), hay que entrar, (función chdir()), y listar los archivos de este nuevo directorio haciendo una llamada recursiva, (una función que se llama a sí misma), hasta que finaliza cuando findnext() retorna un valor nulo porque no encontró más archivos ni directorios.

UNIDAD VI

Introducción A C++:

A mediado de los '80 se alcanzó un nuevo paradigma en la programación, la POO, (programación orientada a objetos), introdujo conceptos nuevos y con ellos una nueva interpretación de los problemas a resolver.

El lenguaje C se expandió con nuevas características y dio origen a un subconjunto que llamaron C++, (++ proviene de la idea de que básicamente es C pero con algunas cosas más).

En torno a un nuevo concepto, "objeto", se edifico no sólo este lenguaje sino que a partir de aquí todos lo lenguajes tuvieron la necesidad de comprender este nuevo tipo de programación.

Un programa en C++ se escribe de la misma forma que uno en C, esto es, tiene una función main(), se deben incluir archivos de cabecera, (los mismo que se usan en C), las sentencias terminan en ";", etc. En fin no existe diferencias en ese sentido, el compilador de C++ interpreta todas las sentencias de C.

C++ admite todas las funciones del C clásico y agrega algunas nuevas, necesarias para este nuevo paradigma, que es la POO.

C++ es una, se podría decir, "modernización" del C clásico.

Fundamentalmente C++ enriquece el lenguaje C con:

Concepto de Clase, (que precisamente es el pilar de la POO).

Flujos de entrada y salida.

Concepto de "polimorfismo", (esencialmente en funciones y operadores).

Existen muchos detalles que incorpora C++, pero fundamentalmente los tres ítems presentados son las características más sobresalientes.

Flujos De Entrada Y Salida, (Cabecera iostream.h):

Tanto printf() como scanf() se usan en C++, pero también se incorpora un nuevo método de describir las salidas y entradas, que son los flujos. Por esto un programa C++ siempre incluye el archivo de cabecera iostream.h.

Veamos como sería un "Hola Mundo" en C++:

```
#include <iostream.h>

void main(void)
{
 cout << "Hola Mundo";
}
```

En lugar de printf("Hola mundo") se usa cout y el operador de inserción <<.

Imagínese que la palabrita cout es la pantalla, lo que se quiere hacer es enviar el texto "Hola mundo" allí, entonces la dirección del operador de inserción indica que*: "Hola mundo" va hacia la pantalla*.

Sería incorrecto escribir: cout >> "Hola mundo", pues el operador de inserción debe apuntar hacia donde va el flujo de la información, o sea, *del texto a la pantalla*.

Exactamente lo mismo ocurre si uno desea ingresar un dato desde el teclado y guardarlo en una variable. El flujo va desde el teclado hacia la variable. Por lo tanto para ingresar por teclado, por ejemplo, un valor entero se escribe:

```
int var;

cin >> var;
```

En este caso imagine que cin es el teclado, entonces como quiere guardar el valor que ingresa en la variable var, el operador de inserción apunta hacia la variable.

La biblioteca iostream.h define cuatro flujos estándar:

El flujo cin, definido por la clase istream, está conectado al periférico de entrada estándar, (el teclado, representado por el archivo STDIN).

El flujo cout, definido por la clase ostream, está conectado al periférico de salida estándar, (la pantalla, representado por el archivo STDOUT).

El flujo cerr, definido por la clase ostream, está conectado al periférico de error estándar, (la pantalla, representado por el archivo STDOUT).

El flujo clog, definido por la clase ostream, está conectado igualmente al periférico de error estándar, (la pantalla). Al contrario que cerr, el flujo clog se realiza a través de un buffer.

Tanto cout como cin se pueden presentar con operadores de inserción en cascada, por ejemplo:

```
#include <iostream.h>
```

```
void main(void)
{
 char texto[10] = “Hola Mundo”;

 cout << texto << “ Cruel”;

}
```

La salida sería: Hola Mundo Cruel.

Los saltos de líneas se pueden realizar con ‘\n’ ó con endl. Ejemplo:

```
cout << “To be” <<endl <<”or not to be.”;
```

Y la salida es:

```
To be
or not to be.
```

Clases:

Las clases son el pilar de la POO, pues, comúnmente se dice que una clase es algo así como la plantilla con la cual se construye un objeto.

¿Qué es un objeto?.

Un objeto que se crea a partir de una clase, tiene sus atributos y sus propias funciones con las cuales manipula esos atributos.

Y, ¿cómo se crea una clase?.

Veamos:

En lenguaje C tradicional existen las estructuras de datos, las cuales se definen con la palabra clave **struct**, Ejemplo:

```
struct Coordenadas
{
    int x;
    int y;
    int z;
}
```

Con una estructura uno crea un tipo de dato nuevo, en este caso, se puede declarar una variable de tipo Coordenadas, la cual puede almacenar 3 valores enteros:

```
struct Coordenadas coo;    //Declaración de la variable coo de tipo Coordenadas
coo.x=7;                           //Valores iniciales para los datos
miembros.
coo.y=15;
coo.z=55;
```

x, y, z son los "datos miembros" de la estructura. Para manipular estos datos, (asignarles un valor inicial, cargarlos, mostrarlos, etc.), uno puede escribir funciones globales en su programa. Ejemplo:

```
void Carga(void)
void Muestra(void)
```

Bueno, se podría decir que una estructura es el "antepasado" más directo de una clase.

¿Por qué?.

Que tal si las funciones con las cuales uno manipula los datos de la estructura formaran parte de ella, o sea, una estructura tal que además de definir sus datos miembros también definiera las funciones para manipularlos. Este tipo de estructuras existe en C++ y se definen igual que las estructuras de C pero además uno puede declarar las funciones.

Mire el siguiente ejemplo:

```
//Estructura con funciones miembros.
#include <iostream.h>

struct Coordenadas
{
    int x,y,z;

void Cargar(void)        //Función miembro que carga los datos.
    {
    x=8;
    y=9;
    z=10;
    }
void Mostrar(void)        //Función miembro que muestra el contenido de los
datos.
    {
```

```
    cout << x <<endl;
    cout << y <<endl;
    cout << z <<endl;
    }
};

void main(void)
{
    struct Coordenadas coo;    //Se define una variable, (coo), de tipo
Coordenadas.

    coo.Cargar();                        //Llamadas a las funciones de coo.
    coo.Mostrar();
}
```

Ahora examine el siguiente programa y encuentre las diferencias con el anterior:

```
//Lo mismo pero con una clase.
#include <iostream.h>

class Coordenadas
{
    int x,y,z;

public:
    void Cargar(void)
    {
        x=8;
        y=9;
        z=10;
    }
    void Mostrar(void)
    {
       cout << x <<endl;
       cout << y <<endl;
       cout << z <<endl;
    }

};

void main(void)
{
    Coordenadas coo;  //Mi objeto coo, (una instancia de Coordenadas)

    coo.Cargar();
    coo.Mostrar();

}
```

¿Encontró las diferencias?.

La verdad, no son muchas. En lugar de **struct** se pone **class**, luego se agrega la etiqueta **public**, antes de definir las funciones miembros, ya que para **una estructura** los datos miembros y funciones miembros son **por defecto públicos**, pero **en una clase** por defecto los datos miembros son **privados**, (esto forma parte, entre otras cosas, de lo que se llama "encapsular"), y sólo las funciones públicas pueden tener acceso a los datos privados.

Y la otra diferencia es en el momento de definir(*) la variable coo, no hace falta especificar la palabra class así como se hizo con struct.

(*) En la POO, utilizando clases, ya no se habla de "definir" una variable de una clase en particular, sino que se crea una "instancia" o un objeto de dicha clase.

¿Por qué usar clases y no estructuras?.

A veces la diferencia, aparte de la sintaxis, no es del todo "pesada" como para justificar una clase. En este ejemplo no hacía falta definir una clase, la versión de la estructura es más que suficiente.

Pero cuando el concepto del objeto a crear es un tanto más complejo, y preocupa, por ejemplo, la protección de los contenidos de los datos miembros, o se tiene una gran cantidad de funciones miembros, o simplemente se pretende en serio programar según POO, es cuando una clase se hace presente.

Pues como supongo astutamente dedujo, la Programación Orientada a Objetos, consta de objetos, y una clase, define o es como la "plantilla" sobre la cual se construyen los tan mentados.

Constructores:

En una clase existe una función miembro muy particular llamada **Constructor**.

Un constructor es una función que debe tener el mismo nombre que la clase y no debe retornar ningún valor, (ni siquiera void), y se encarga de asignarle valores iniciales, (o simplemente inicializar), a los datos miembros.

En el ejemplo descubrirá que allí no hay ningún constructor definido, cuando ocurre esto el compilador de C++ crea en ejecución el constructor.

No obstante hubiera sido correcto haber definido un constructor que se encargara de, por ejemplo, inicializar con 0 los datos miembros.

Un constructor es invocado automáticamente cuando se crea la instancia, o sea que no hay que llamarlo explícitamente desde el programa principal.

Existen 3 tipos de constructores:

- *Constructor por defecto.*
- *Constructor común.*
- *Constructor de copia.*

El **constructor por defecto** es, en caso que no lo haya definido, el que C++ en tiempo de ejecución le asigne, o bien:

```
class Coordenadas
{
    int x,y,z;

public:
    Coordenadas();     //Constructor por defecto

};
```

También le podríamos haber agregado a este constructor, encerrados entre llaves, los valores iniciales para los datos:

```
{x=0;y=0;z=0;}.
```

Cuando se crea el objeto se escribe:

```
void main(void)
{
    Coordenadas coo;
    ....
}
```

El **constructor común** es aquel que recibe parámetros para asignarles como valores iniciales a los datos miembros, o sea que al crear la instancia, se pasó unos parámetros para inicializar.

```
class Coordenadas
{
    int x,y,z;

public:
    Coordenadas(int p, int q, int t) {x=p; y=q; z=t;}     //Constructor común.

};
```

Cuando se crea el objeto se escribe:

```
void main(void)
{
    Coordenadas coo(6,7,22);       //Se le pasa los valores para inicializar.
    .....
}
```

El constructor de copia se utiliza para inicializar un objeto con otro objeto de la misma clase.

```
class Coordenadas
{
    int x,y,z;

public:
     Coordenadas ( int p, int q, int t) {x=p; y=q; z=t;}        //Constructor
común.
    Coordenadas(const Coordenadas c)    //Constructor de copia.
        {
            x=c.x;
            y=c.y;
            z=c.z;
        }

};
```

Cuando se crea el objeto se escribe:

```
void main(void)
{
     Coordenadas k(1,2,3);     //Creación de un objeto
                               // con lo valores iniciales 1, 2 y 3.
     Coordenadas coo=k;        //Se llama al constructor de copia para que le
                               // asigne a coo los valores de k.
     ....
}
```

Sobrecarga De Funciones, (polimorfismo):

Habrá advertido en el último ejemplo de la clase, donde se ve el *constructor de copia*, que también se define un *constructor común*. Bueno, eso es posible, una clase puede tener varios constructores, que se irán usando de acuerdo a como uno cree el objeto, (pasándole o no parámetros).

Pero, observe nuevamente, esta vez más detalladamente, la clase..., ¿no encuentra otra cosa extraña?.

Los constructores son funciones, ¿¿¿cómo permite el compilador dos funciones con el mismo nombre???.

Ahh, buena pregunta.

El compilador de C++ permitiría 100 funciones con el mismo nombre, el único requisito es que cada una de ellas tenga diferente número y/o tipo de parámetros.

Esta cualidad, que no se aplica solamente a los constructores y funciones miembros de una clase, sino que a cualquier función de un programa de C++, se llama *Sobrecarga de funciones* o *Polimorfismo*.

Cuando se llama a la función, C++ selecciona de todas las funciones sobrecargadas aquella que se ajusta de acuerdo con los parámetros pasados, en cantidad y tipo.

Funciones InLine:

También se puede estar preguntando, si las funciones miembros de una clase pueden estar definidas fuera de la clase.

La respuesta es sí, por lo general las funciones miembros están definidas fuera de la clase, dentro de ésta última sólo se declararían los prototipos.

En el caso que la función esté definida dentro de la clase, ésta se llama **función inline**, como las funciones Cargar() y Mostrar() de nuestra clase Coordenadas. Se podría incluso agregar la cláusula inline, pero no hace falta.

¿Qué diferencia hay entre una función inline y otra, (definida dentro o fuera de la clase)?

Se define una función inline cuando es muy corta y simple, como los constructores y esas funciones del ejemplo. Declarar una función en línea significa que el compilador puede, si así lo decide, reemplazar cada invocación por la función, con la frecuencia que sea, por el código encerrado entre llaves.

Hay que tener en cuenta que funciones inline extensas consumen más memoria, a pesar que elimina el tiempo que lleva hacer la invocación.

Cuando se escribe una función fuera de la clase se especifica el acceso de la siguiente forma:

```
NombreClase::Función()    //Note que se accede con ::
```

Así quedaría nuestro programa, con la clase con un constructor por defecto y con las funciones miembro fuera de la clase.

```
 #include <iostream.h>

class Coordenadas
{
    int x,y,z;
public:
    Coordenadas(){x=0;y=0;z=0;}     //Constructor por defecto.
    void Cargar(void);                        //Prototipo de las funciones.
    void Mostrar(void);
};

void Coordenadas::Cargar(void)        //Definición de las funciones fuera de la
clase
{
    x=8;
    y=9;
    z=10;
}

void Coordenadas::Mostrar (void)
{
    cout << x <<endl;
    cout << y <<endl;
    cout << z <<endl;
}

void main(void)
{
    Coordenadas coo;

    coo.Cargar();
    coo.Mostrar();
}
```

Destructores:

Existe una función especial más para las clases, y se trata de los destructores.

Un destructor es una función miembro que se llama cuando se destruye la clase.

Todas las clases tienen un destructor implícito, incluso aunque no esté declarado. El destructor implícito no hace nada en particular, pero si uno quiere, puede declarar un destructor de forma explícita. Su sintaxis sería:

```
class NombreClase
{
    ...
 public:
        ~NombreClase();
        ...
}
```

El destructor debe comenzar con el caracter "ñuflo", (~), seguido por el nombre de la clase, (igual que el constructor). Además el destructor no puede recibir parámetros ni retornar nada, (ni siquiera void).

No puede haber más de un destructor para una clase y si no se define uno explícitamente, el compilador crea uno automáticamente.

El destructor se llama automáticamente siempre que una variable de ese tipo de clase, (una instancia u objeto), sale fuera de su ámbito, (por ejemplo cuando termina el programa).

Especificadores de acceso:

Ya había dicho que **por defecto** los **datos miembros** de una clase son **privados**. ¿Qué significa esto?.

Que sólo las funciones miembros públicas de la misma clase tienen acceso a ellos. Si lo desea puede escribir la cláusula *private* al momento de declarar los datos.

En cambio la cláusula *public* es obligatoria cuando se desea declarar un dato público y este dato estará disponible para cualquier función del programa.

Existe una cláusula más, *protected*. Los datos definidos a continuación de esta cláusula están restringidos para cualquier función externa a la clase, pero son públicos para la propia clase y los miembros de clases derivadas.

Herencia:

La potencia de las características de la POO proviene esencialmente de la capacidad de derivar clases a partir de clases existentes. Una clase descendiente hereda los miembros de sus clases ascendientes y puede anular alguna de las funciones heredadas.

La herencia permite la continua construcción y extensión de clases desarrolladas por usted u otras personas, sin límite aparente.

C++ permite dos tipos de herencia:

Herencia simple: una clase se deriva de una y sólo una clase.

Herencia múltiple: una clase puede ser derivada de más de una clase al mismo tiempo.

Cuando una clase se hereda de otra clase, la clase original se llama *clase base* o *clase madre* y la nueva clase se llama *clase derivada* o *clase hija.*

En una clase derivada las características heredadas de la clase base pueden ser cambiadas, eliminadas, ampliadas o simplemente utilizadas.

Una clase derivada se declara de la siguiente forma:

```
Class D : modificador_acceso B   //por defecto el modificador de acceso es
private.
{
  declaraciones de miembro, (datos y funciones miembro).
}
```

donde D es el nombre de la clase derivada, modificador_acceso es opcional, (public, private ó protected) y B es el nombre de la clase base.

Ejemplo:

En el siguiente ejemplo se declara una clase Caja cuyos datos miembros son la anchura y peso, y las funciones miembro PonerAnchura(), PonerPeso(), VerAnchura() y VerPeso().

Además se deriva una nueva clase a partir de Caja que se llama CajaDeColor.

```
#include <iostream.h>

class Caja
{
 public:
       int anchura, peso;
 void PonerPeso(int p) {peso =p;}
```

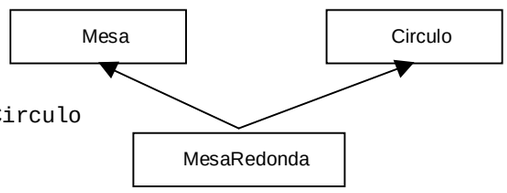
```
 void PonerAnchura(int a) {anchura = a;}
 void VerPeso(void) { cout << peso;}
 void VerAnchura(void) { cout << anchura;}
};

class CajaDeColor : public Caja   //CajaDeColor se deriva de Caja
{
 public:
    int color;
    void PonerColor(int c) { color = c;}
    void VerColor(void) { cout << color;}
};

void main(void)
{
 CajaDeColor cdc;   //se construye un objeto CajaDeColor
 cdc.PonerColor(10);
 cdc.PonerPeso(4);     //función miembro heredada
 cdc.PonerAnchura(20);  //función miembro heredada
}
```

En el ejemplo se aprecia, que si bien en la definición de CajaDeColor no se encuentran las funciones miembro PonerPeso() y PonerAnchura(), estas forman parte de la clase ya que han sido heredadas de Caja y por eso el objeto cdc las puede usar en el programa.

Herencia múltiple:

En herencia simple, una clase tiene una y sólo una clase base; mientras que en herencia múltiple, una clase derivada puede tener dos o más clases bases.

Por ejemplo se tienen dos clases, Mesa y Circulo, de ellas se puede derivar una clase llamada MesaRedonda aprovechando los datos miembros de ambas clases padre.



La sintaxis sería:

```
Class MesaRedonda : public Mesa, public Circulo
{
 declaraciones de miembros...
};
```

Operadores new Y delete:

Operador new:

El operador new de C++ equivale a la función malloc() del lenguaje C tradicional.

El operador new asigna un bloque de memoria que es el tamaño del tipo de dato. El dato u objeto dato puede ser un int, float, una estructura, un array o cualquier otro tipo de dato. El operador new devuelve un puntero, que es la dirección del bloque asignado de memoria. El puntero se utiliza para referenciar el bloque de memoria.

El formato del operador es:

```
puntero = new nombreTipo (tamaño de inicialización opcional)
```

O bien:

```
tipo *puntero = new tipo                  //datos básicos, estructuras
tipo *puntero = new tipo[dimensiones]     //arrays
```

El compilador realiza una verificación de tipo para asegurar que el tipo del puntero especificado en el lado izquierdo del operador es el tipo correcto de la memoria que se asigna en la derecha.

```
int *p;

p = new int;
```

otro ejemplo:

```
int *BloqueMem;

BloqueMem = new int[100];
```

Si un bloque del tamaño solicitado está disponible, new devuelve un puntero al principio de un bloque de memoria del tamaño solicitado. Si no hay espacio suficiente, new devuelve cero o NULL.

```
int *BloqueMem;

BloqueMem = new int[1000];

if (p==NULL)  //ó p ==0
   cout << "No se pudo asignar el bloque de memoria pedido" <<endl;
```

Operador delete**:**

El operado delete elimina, (libera), la asignación hecha con new. El bloque de memoria suprimido se devuelve al espacio de almacenamiento libre.

```
delete puntero
```

Siempre es conveniente usar delete luego de utilizar el bloque de memoria asignado con new, pues se podrían presentar inconvenientes de forma impredecible.

Notación Húngara Básica

Breve introducción

La Notación Húngara es una convención para determinar el nombre de un *identificador* anteponiendole un prefijo en minúsculas para identificar el tipo o utilización. Fue inventado por el notable programador húngaro Charles Simonyi. No existe un estándar de **HN** (Hungarian Notation), la idea es crear un sistema de notación consistente que facilite la lectura del código fuente.

Entre los identificadores más comunes se hallan los nombres de variables, funciones y procedimientos. Un identificador en HN se divide en 4 partes: el *Prefijo* (o constructor), *tipo base* (o TAG), el *Nombre* propiamente dicho y un *Cualificador*. No todos estos elementos son obligatorios tan solo el nombre y en general el *tipo base* deberian estar presentes.

Tipo Base (TAG)

El tipo base no es necesariamente un tipo provisto por el lenguaje de programación, puede estar definido por la aplicación (por ejemplo un TAG *dbr* puede identificar a una estructura de un registro de una base de datos). Los TAGS son breves recordatorios descriptivos (normalmente entre una y tres letras) acerca del tipo de valor almacenado por una variable o devuelto por una función. Este tipo puede ser utilizado solamente por quienes conocen la aplicación y conocen los tipos básicos que utiliza la aplicación; por ejemplo, un tag *co* puede ser referencia a una *co*ordenada o a un *co*lor. Dentro e la aplicación, no obstante, el *co* siempre tiene una aplicación exclusiva -- todos los *co* deben referenciar al mismo tipo de objeto, y todas las refrencias a ese objeto deben comenzar con el tag *co*.

TAGS comunes

Aquí hay una lista de tipos básicos que pueden utilizarse en una aplicación. Normalmente y por (buena o mala) costumbre suelen ser abreviaturas del inglés.

f	flag de tipo booleano, el Cualificador debería ser utilizado para describir la condición de encendido del flag, por ejemplo *fError* indicaría que la bandera se enciende cuando es encontrado un error.
ch	un caracter en un byte.
b	un byte (generalmente 8 bits)
i	un entero (integer)
li	un entero largo (long int)
ui	un entero sin signo (unsigned int)
r	un valor real en simple precisión (float)
d	un valor real en doble precisión (double)
v	un valor void (un puntero a algo indeterminado por ejemplo)
sz	un string terminado en cero

Prefijos (Constructores)

El tipo base no suele ser suficiente para describir una variable, pues las variables pueden hacer referencias a valores complejos. Por ejemplo, usted puede tener un puntero a un registro de una base de datos, o a un array de coordenadas, o a un contador de colores.

En HN esos tipos extendidos son descriptos mediante el prefijo de la variable. puede ser tambien que tenga mas de un prefijo, por ejemplo un puntero a un array de registros.

p	un puntero
lp	un puntero lejano (far pointer) utilizado en maquinas con arquitectura de memoria segmentada.
rg	un Array.
i	un indice (dentro de un array por ejemplo)
c	un contador
aq	un acumulador
g	una variable global o pública

Estos tipos y prefijos deben combinarse con un identificador cuya parte significativa comienze con mayúsculas:

`nOpcion` - Almacena la opción elegida por el usuario en un menú numérico

`cOpcion` - Almacena la opción elegida por el usuario en un menú Alfabético

`szNombreArchivo` - Un string terminado en cero con un nombre de archivo en él.

`pfsArchivoClientes` - Un puntero a un archivo de datos.

`ctBytesProcesados` - contador del trabajo realizado.

Si usted necesita utilizar un tipo que no estuviera indicado, invente un nuevo tag o prefijo y utilicelo **consistentemente**

Procedimientos

Estas simples reglas para nombrar variables no simpre trabajan bien sobre procedimientos. Esto es a causa de que lo importante es lo que el procedimiento *hace* y no tanto el tipo de dato que devuelve. También el contextopara los procedimientos es usualmente el programa entero, por lo tanto hay mayor probabilidad de conflictos de nombres. Para manejar ese problema se hacen algunas modificaciones en las reglas:

1. Los Nombres de procedimientos se distinguen de los nombres de variables por la utilización de signos de puntuación. Por ejemplo los nombres de la funciones tienene la primera letra en mayúsculas mientras los nombres de variables comienzan por minúsculas.
2. Si el procedimiento explicitamente retorna un valor, entonces el nombre puede comenzar con el tipo de valor que retorna.
3. Si el procedimiento es una verdadera función (como esto: el opera con sus parámetros y retorna un valor sin otros efectos), entonces es tipico nombrarlo (en inglés) como `XfromYZ...`, donde `X` es el tipo de valor retornado y `Y`, `Z`, etc. Son los tipos de los parámetros. Por ejemplo SzFechaFromDMA(iDia, iMes, iAño) puede nombra a una función que devuelve un String terminado en cero a partir de un dia, mes y año
4. Si el procedimiento tiene efectos adicionales entonces siguiendo al tipo (si existiera) con varias palabras que describen que hace el procedimiento. Por ejemplo: FObtieneRaiz(dRaiz1, dRaiz2, dTerminoA, dTerminoB, dTerminoC) será un procedimiento que obtiene las raices de una función cuadrática a través de los coeficientes de los términos, pero además devuelve un flag (valor Booleano) indicando si las raices son reales.
5. Si el procedimiento opera con un objeto, el tipo del objeto debe ser añadido al nombre, por ejemplo IniciaFoo(pFoo) puede indicar un procedimiento que inicializa una estructura denominada Foo cuyo puntero es pasado como parámetro.

Macros y Constantes

Las Macros son usualmente manejados como si fueran procedimientos, Las constantes pueden ser manejadas como si fueran variables (como por ejemplo fTrue y fFalse), no obstante usted puede ver constantes definidas todo en mayúsculas (PI, LIMITE_SUPERIOR por ejemplo). Si mal no recuerdo, esta forma en mayúsculas no es parte de la HN, pero es utilizada muy corrientemente por muchos programadores para distinguir constae de variables (y macros de funciones).

Etiquetas (Labels)

Si usted necesita una etiqueta por alguna razón, esta puede ser considerada como una variación de un procedimiento (Las etiquetas son identificadores de un trozo de código). Como las Etiquetas no reciben parámetros ni devuelven un valor, no se debe especificar ningún tipo. FinLoop o FueraDeMemoria son típicos ejemplos.

Bibliografía consultada:

Ayuda de Visual C++ 6.0, (MSDN).
Ayuda de Borland C++ Builder 5.0
Ayuda de Borland C/C++ 3.1
“C++”, de Joyanes Aguilar y Castan Rodríguez, (Mc Graw Hill).

Para mayor referencia visite: www.visualc.8k.com