



Clase 1

Contenidos

1	Lenguaje C	3
1.1	Introducción general	3
1.1.1	Ventajas	3
1.1.2	Desventajas	3
1.2	Reseña histórica	3
1.3	Estructura de un programa C	3
1.4	Tipos, Operadores y expresiones semánticas	4
1.4.1	Tipos de Datos	4
1.4.2	Definición y alcance de las Variables	5
1.4.3	Operadores	5
1.4.3.1	Operadores unarios	5
1.4.3.2	Operadores binarios	5
1.4.3.3	Operador ternario	5
1.5	Ubicación de variables en memoria	6
1.6	Control de flujo	6
1.6.1	If else	6
1.6.2	Switch case	6
1.6.3	For	6
1.6.4	While	6
1.6.5	Do while	6
1.6.6	Break, continue	6
1.6.7	Goto	7
1.7	Arreglos y Strings	7
1.8	Matrices	8
1.9	Modificadores del tipo de una variable	9
1.9.1	Atributo STATIC	9
1.9.1.1	Static .vs. Global	11
1.9.2	Atributo CONST	11
1.10	Tipos enumerados	11
1.10.1	struct	11
1.10.2	typedef	13
1.10.3	enum	13
1.10.4	union	14
1.10.5	Struct .vs. Union	15
1.11	Funciones & Alcance de funciones	16
1.11.1	Pasaje de parámetros	17
1.12	Entrada y salida standard	17
1.13	Archivos	19
1.13.1	Ejemplo de aplicación	20

1 Lenguaje C

1.1 Introducción general

El lenguaje de programación C es un lenguaje de propósito general de nivel medio ya que combina elementos de lenguaje de alto nivel con la funcionalidad del ensamblador, con una versatilidad que lo hacen indicado para ciertas tareas en las que lenguajes de alto nivel no son suficientemente óptimos o no dan al programador el control necesario.

1.1.1 Ventajas

- Programación estructurada
- Lenguaje nivel medio
- Lenguaje nativo de la plataforma UNIX
- Muy similar a lenguajes de scripting (p.e. perl, php)
- Economía de las expresiones.
- Abundancia de operadores y tipos de datos.
- Producción de código objeto optimizado

1.1.2 Desventajas

- No es 100% portable
- Según las buenas practicas de programación puede presentar dificultades al momento de leer el código, legibilidad.
- No ofrece multiprogramación, operaciones paralelas ni sincronización.
- Delega las instrucciones de entrada salida en las bibliotecas de rutinas generando perdida de portabilidad.

1.2 Reseña histórica

El lenguaje C fue originalmente diseñado e implementado sobre los sistemas operativos UNIX en la DEC PDP-11 por Dennis Ritchie, tanto el sistema operativo como el compilador de C, y todas las aplicaciones del sistema fueron escritas en C. El mismo nació en los laboratorios Bell de AT&T y se lo asoció con el sistema UNIX por todo lo que describimos anteriormente.

El lenguaje fue inspirado en el lenguaje B escrito por Ken Thompson en 1970 , B era un lenguaje evolucionado e independiente de la maquina inspirado en el lenguaje BCPL concebido por Martin Richard en 1967.

En 1972 Dennis Ritchie modifica el lenguaje B creando el lenguaje C , agregando los tipos y estructuras de datos. Pero recién en 1983 ANSI (American National Standards Institute) estableció un comité para crear una definición no ambigua del lenguaje C e independiente de la maquina que pudiera utilizarse en todos los tipos de C.

1.3 Estructura de un programa C

Comentarios

Inclusión de archivos

```
int main(int argc, char** argv)
{
    variables locales
    flujo de sentencias
}
```

Definición de funciones creadas por el programador utilizadas en main()

Aclaración!! : En ANSI C, las expresiones son Case Sensitive, esto muchas veces lleva a errores en tiempo de compilación para quien se inicia con el lenguaje, esto significa que :

Printf <> printf <> PRINTF

Dni <> DNI <> dni

1.4 Tipos, Operadores y expresiones semánticas

1.4.1 Tipos de Datos

C soporta los siguientes tipos de datos básicos, definidos por el compilador. Notar que puede variar el significado de cada uno de ellos según la plataforma sobre la que se esté trabajando.

Tipo de dato	Significado	Longitud habitual (en bits)
char	Caracter / Entero (-128 a 127)	8
unsigned char	Caracter / Entero (0 a 255)	8
short int	Entero reducido (-127 a 128 ó -32768 a 32767)	8 ó 16
unsigned short int	Entero reducido (0 a 255 ó 0 a 65535)	8 ó 16
int	Entero con signo	16 ó 32
unsigned int (ó unsigned)	Entero sin signo	16 ó 32
long int	Entero largo con signo	32
unsigned long int (ó unsigned long)	Entero largo sin signo	32
long long ó long64 ó _int64	Entero largo. No forma parte del standard pero varios compiladores lo implementan	64
float	Punto flotante	IEEE 32
double	Punto flotante doble precisión	64
long double	Punto flotante doble precisión	80 ó 96
void	Sin valor	indeterminado

El tipo de dato puede variar de un sistema a otro. Por ejemplo, un entero (int), se compone de 16 bits en sistemas operativos de 16 bits, y de 32 bits en sistemas operativos de 32 bits.

Si bien tantos tipos de datos pueden parecer difícil de recordar, basta saber que existen básicamente dos tipos de datos: enteros y de punto flotante (reales), llamados int y float/double respectivamente, y que

existen modificadores (prefijos) unsigned, long, etc, para controlar el uso de signo y su longitud. Existe también el prefijo signed, opuesto a unsigned, pero rara vez se lo usa ya que, por defecto las variables son con signo (puede especificarse al compilador que no sea así, pero por convención nunca se lo hace).

Es bueno tener en cuenta que si bien el compilador elige de acuerdo a la plataforma la cantidad de bits que utiliza para un short, o long se debe cumplir la siguiente regla :

```
short int < int <= long int
```

1.4.2 Definición y alcance de las Variables

Las variables se definen escribiendo su nombre (identificador), que puede constar de hasta 32 caracteres alfanuméricos, precedidos por el tipo de variable. Toda definición de variable debe necesariamente terminar en punto y coma (;) y no pueden comenzar con números .

Las variables definidas en cualquier función o bloque de código, es decir, en cualquier bloque delimitado por llaves {}, tiene un alcance local relativo a ese mismo bloque. Esto quiere decir que al salir del bloque donde fueron definidas dejarán de existir.

La única excepción son las variables definidas fuera de cualquier función, que tienen alcance global.

En cualquier parte de una función, al comienzo de un bloque delimitado por llaves, se pueden declarar variables, las cuales tendrán el alcance antes mencionado.

No puede definirse ninguna variable en un bloque, luego de ejecutarse alguna instrucción o sentencia¹.

1.4.3 Operadores

1.4.3.1 Operadores unarios

Operador	Nombre
~ !	Negacion / complemento
* &	Indireccion y direccion
sizeof	Tamaño
++ —	Incremento y decremento

1.4.3.2 Operadores binarios

Operador	Nombre
* / %	Multiplicación, division y resto
+ -	Suma y resta
<< >>	Despazamiento de bits
< > <= >= == !=	Operadores de relacion
& ^	AND, OR y NOT para bits
&&	AND Y OR logicos

1.4.3.3 Operador ternario

Operador	Nombre
()?:	Condicional

¹ Esto no se cumple para el ANSI ISO C99, el cual permite entre otras cosas declarar las variables en cualquier parte del bloque de código.

```
(condicion)?sentencia1:sentencia2;
```

1.5 Ubicacion de variables en memoria

Es importante tener claro en que sector de la memoria se ubican cada una de las variables que utilizaremos sin contar que esto afecta su comportamiento de forma determinante.

Code Segment	Data Segment	Stack Segment	Heap
<ul style="list-style-type: none"> ▪ Código fuente ▪ Constantes 	<ul style="list-style-type: none"> ▪ Variables Globales ▪ Variables Static 	<ul style="list-style-type: none"> ▪ Variables Automáticas, Locales no static. 	<ul style="list-style-type: none"> ▪ Zonas alocadas dinámicamente ,mediante malloc y liberadas mediante free.

1.6 Control de flujo

1.6.1 If else

```
if( condicion )
    sentencia
else
    sentencia
```

1.6.2 Switch case

```
switch( variable )
{
    case valor1: sentencia; break;
    case valor2: sentencia; break;
    case valor3: sentencia; break;
    case valor4: sentencia; break;
    default: sentencia; break;
}
```

1.6.3 For

```
for( inicializacion de variable ; condicion de corte; modificacion variable )
    sentencia
```

1.6.4 While

```
while( condicion )
    sentencia
```

1.6.5 Do while

```
do
    sentencia
while( condicion )
```

1.6.6 Break, continue

Para ver un ejemplo de estas dos instrucciones, estudiemos el siguiente código:

```
# include < stdio.h >

int main()
{
    int xx;

    for(xx = 5 ; xx < 15 ; xx = xx + 1)
    {
        if(xx == 8)
            break;
        printf("Este bucle se ejecuta cuando xx es menor de 8, "
            "ahora xx es %d\n", xx);
    }
    for(xx = 5 ; xx < 15 ; xx = xx + 1)
    {
        if(xx == 8)
            continue;
        printf("Ahora xx es diferente de 8, xx tiene el valor de %d\n", xx);
    }
    return 0;
}

/* Resultado de la ejecución:
Este bucle se ejecuta cuando xx es menor de 8, ahora xx es 5
Este bucle se ejecuta cuando xx es menor de 8, ahora xx es 6
Este bucle se ejecuta cuando xx es menor de 8, ahora xx es 7
Ahora xx es diferente de 8, xx tiene el valor de 5
Ahora xx es diferente de 8, xx tiene el valor de 6
Ahora xx es diferente de 8, xx tiene el valor de 7
Ahora xx es diferente de 8, xx tiene el valor de 9
Ahora xx es diferente de 8, xx tiene el valor de 10
Ahora xx es diferente de 8, xx tiene el valor de 11
Ahora xx es diferente de 8, xx tiene el valor de 12
Ahora xx es diferente de 8, xx tiene el valor de 13
Ahora xx es diferente de 8, xx tiene el valor de 14
*/
```

1.6.7 Goto

Si bien existe y tiene la sintaxis *goto Etiqueta*, es bueno conocerlo solo a fines informativos, y se recomienda fuertemente no utilizarlo. Para los fines de la materia tomar como goto == Prohibido!!

1.7 Arreglos y Strings

Los strings en C tienen la particularidad de ser terminados con un carácter especial de finalización: el valor NULL = '\0'. Supongamos que queremos armar un arreglo que contenga la cadena "Hola Mundo";

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

int main()
{
    char msg[]="Hola Mundo";

    printf("%s\n",msg);
    printf("%c\n",msg[0]);
    printf("%c\n",msg[strlen(msg)-1]);

    return 0;

/*
    La salida sera
    Hola Mundo
    H
*/
```

```

*/
    ○
}

```

Lo que ocurre es que en memoria se almacena “Hola Mundo\0”, por eso al llamado de strlen se le debe restar una posición, ya que este incluye el \0 para calcular el valor de la cadena.

¿Qué ocurre si agregamos un ‘\0’ en la mitad de la cadena?

```

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

int main()
{
    char msg[]="Hola Mundo";

    msg[4] = '\0';
    printf("%s\n",msg);

    return 0;

/*
    La salida sera
    Hola
*/
}

```

Esto es debido a que la función printf lee hasta encontrar el ‘\0’ de finalización. Si este no existe, imprimiría “Hola” seguido de basura.

Por supuesto que pueden declararse matrices de (por ejemplo) enteros utilizando int[tamaño] siguiendo los mismos lineamientos. Vale lo mismo con las matrices, posibilitando la declaración de ellas de esta manera int[j][k].

Atención!! : strcpy copia hasta que encuentra un ‘\0’ en la cadena origen, que si es superior al tamaño de la cadena de destino, pisa memoria. Es importante al manejar cadenas siempre recordar el ‘\0’ ya que es el símbolo de fin de string.

1.8 Matrices

Definimos una matriz de m x n como un arreglo de m elementos de dimensión n. Podemos ver como ejemplo las siguientes declaraciones de variables:

```

int          bidimension[10][20];
short int    tridimension[2][3][4];

char         tridimensionc[2][2][3] = {
                { {'A','B','C'}, {'D','E','F'} },
                { {'G','H','I'}, {'J','K','L'} }
            };

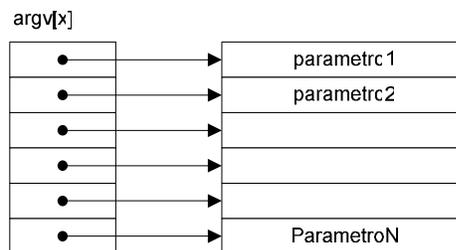
```

Sabiendo esto podemos como ejemplo volver a analizar la primera línea del main e interpretar la línea de comandos que nos llega en el argv :

```
int      main(int argc , char argv[][])
{
  ...
}
```

int argc = Representara la cantidad de parámetros que recibimos por línea de comandos + 1.

char argv[][] = Representara una matriz de dimensión argc x 4bytes (puntero a un string, podríamos verlo como si fuera un vector, al cual para acceder a su primer elemento lo hago mediante argv[nroparametro][0]) viéndolo en un gráfico :



Esto significa que si quisiéramos acceder al primer parámetro (recordando que el parámetro 0 sera el nombre y ruta de nuestra aplicación) deberíamos escribir **argv[1]** lo que nos daría el puntero al primer elemento del arreglo de caracteres del primer parámetro.

Supongamos que quisiéramos copiarlo a una variable de tipo string de 20 caracteres:

```
int      main(int argc , char argv[][])
{
  char parametrol[20];
  strcpy(argv[1], parametrol);
}
```

1.9 Modificadores del tipo de una variable

1.9.1 Atributo STATIC

El atributo static tiene 2 propiedades principales :

- 1) Para globales restringir el alcance de una variable al archivo en que está declarada, para que la misma no sea vista desde otros modulos del proyecto.
- 2) Ubicar la variable en el segmento de datos (Data Segment) como si fuera una variable global, aun si la declaramos dentro de la apertura de llaves de una función. Esto trae

aparejado por ejemplo que la variable no pierda su valor al abandonar el “scope” de la función, propiedad que las variables automáticas no poseen

```
static      tipo_variable      nombre_variable;
```

Un ejemplo como global :

```
static      float      Pi = 3,1416;

int main(int argc, char** argv)
{
    /* Esta variable solo puede ser utilizada dentro de este modulo */
    ...
}
```

Tomemos como caso de estudio los módulos lista.c y su header, y main.c y su header, para poder ver el alcance como modificador de una variable global

Listas h

```
#ifndef __LISTAS__
#define __LISTAS__
#endif
```

Main h

```
#include 'Listas h'
#include 'Files h'

#ifndef __MAIN__
#define __MAIN__
#endif
```

Listas c

```
#include 'Listas h'

static long int filepointer
```

Main c

```
#include 'Main h'

long int filepointer

int main(int argc char** argv)
{
    return -1
}
```

Un
segunda
este

ejemplo de la
propiedad seria

```
void Proc(void)
{
    static int acum = 0;
    printf( "%i", acum++);
}

int main(int argc, char** argv)
{
    int j;

    for ( j=0; j < 20 ; ++j)
    {
        Proc();
    }
    /* La salida por stdout de este programa sera :
       012345678910111213141516171819
       Por que?, esto es por ser acum, una variable de tipo static */

    return 0;
```

}

1.9.1.1 Static .vs. Global

Lo dicho anteriormente nos lleva a pensar en similitudes y diferencias entre variables static y globales, para empezar ambas se ubican en el segmento de datos, y ambas son inicializadas con 0 al comenzar la aplicación. Lo que cambia fundamentalmente es el alcance de las mismas. Las globales solo tienen importancia dentro del ámbito global del programa, y pueden ser accedidas desde cualquier parte del mismo (siempre y cuando utilicemos correctamente las sentencias extern para que el compilador sepa que existen, si están en otro módulo diferente al que son utilizadas). En cambio las static pueden ser accedidas solo desde el módulo en el cual están declaradas, y por otro lado, si tienen sentido dentro del ámbito local de una función.

1.9.2 Atributo CONST

El atributo const transforma la variable en una constante para nuestro programa. La misma es almacenada dentro del segmento de código (Code Segment).

```
const tipo_variable    nombre_variable;
```

Una variable const puede ser local o global, puede ser de tipo static en cuyo caso estaríamos forzando a que la misma se ubique en el segmento de datos.

```
const    float    Pi    = 3,1416;
```

```
const    char    Id    = 'D';
static   const int Dni  = 27451456;
```

1.10 Tipos enumerados

1.10.1 struct

La cláusula **struct**, nos permite definir una estructura en ANSI C, esto es una agrupación de campos dentro de la misma con diferentes o iguales tipos de datos cada uno de ellos, agrupadas bajo un mismo nombre para mayor claridad. Pudiendo acceder al valor de cada uno de los mismos mediante el operador “.” como nombre_estructura.nombre_campo. Son los Record de pascal.

La sintaxis de esta cláusula es la siguiente :

```

struct nombre_structura {
    tipo_campo_1    nombre_campo_1;
    tipo_campo_2    nombre_campo_2;
    ...
    tipo_campo_n    nombre_campo_n;
};

```

Por ejemplo

```

#include <stdio.h>

int main( void )
{
    struct Coordenada {
        int x;
        int y;
    };

    struct Coordenada miCoordenada;
    struct Coordenada* ptrmiCoordenada;

    miCoordenada.x = 10;
    miCoordenada.y = 33;
    printf("%d %d\n",miCoordenada.x, miCoordenada.y);

    ptrmiCoordenada = &miCoordenada;
    ptrmiCoordenada->x = 20;
    ptrmiCoordenada->y = 66;
    printf("%d %d\n",miCoordenada.x, miCoordenada.y);

    return 0;
}

```

```

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

struct    datos_personales {
    char    nombre[30];
    int     edad;
    char    sexo;
    char    telefono[10];
};

struct    complejo {
    float   real;
    float   imaginario;
};

int main(int argc, char** argv)
{
    struct datos_personales dat;
    struct datos_personales Vdat[10];
    struct complejo    x, y;

    dat.edad = 24;
    strcpy("Juan",dat.nombre);
    strcpy("4555-6767",dat.telefono);
    dat.sexo = 'M';

    x.imaginario = 2; x.real = 10;
    // Aqui vemos como se utiliza el struct
    // Asignando un string a nombre
}

```

```

y.inmagenario = x.inmagenario;
y.real = x.real;

return 0;
}

```

1.10.2 typedef

La cláusula *typedef* nos permite crear tipos nuevos dentro de nuestros programas, esto nos da la posibilidad de por ejemplo en ANSI C definir un nuevo tipo bool.

La sintaxis de esta cláusula es la siguiente :

```
typedef      tipo_integral_o_estructura      nombre_tipo;
```

Por ejemplo :

```

typedef      int      bool;
typedef      unsigned long int ulint;
typedef int TEntero;
TEntero Entero;
Bool      result;
Entero = 1;
result = 0;

```

```

typedef struct TPersona {
    char Nombre[50];
    char Apellido[50];
    short int edad;
} Personal;

typedef struct TPersona* ptrPersona;

```

Este ultimo crea 2 nuevos tipos: TPersona y ptrPersona (puntero a TPersona).

1.10.3 enum

Es un set de constantes enteras nombrados, que especifica los valores posibles que puede tomar una variable de ese tipo. Por defecto la enumeración comienza en cero, pero podemos forzarla a que comience con el numero que deseemos como vemos en el ejemplo.

La sintaxis de esta cláusula es la siguiente :

```

enum nombre_enumerado { valor1 , valor2, ... , valorN };

enum nombre_enumerado { valor1 = 1, valor2 = 10 , ... ,valorN = 10 };

```

Por ejemplo

```
enum meses {marzo=3,abril,mayo, junio}
```

```
enum meses MiCuatrimestre;
MiCuatrimestre=abril;
MiCuatrimestre=5; /* =mayo */
```

```
enum edias { lunes = 1 , martes , miércoles, jueves, viernes, sabado, domingo };
enum emeses { enero = 1, febrero, marzo, abril, mayo, junio, julio, agosto,
              septiembre, octubre, noviembre, diciembre};

int main(int argc, char** argv)
{
    enum edias vdias = lunes;
    enum emeses vmeses = enero;

    if ( argc > 2 )
        vdias = viernes;
    else
        vdias = martes;

    return 0;
}
```

1.10.4 union

La cláusula **union**, nos permite definir una estructura similar a la clausula **struct** , pero con la salvedad que cada uno de los campos es de tipo exclusivo, esto quiere decir que tendremos un valor u otro, y no ambos al mismo tiempo. Esta restricción tiene sentido, si pensamos a un **union** como una estructura en memoria que se crea guardando espacio para el valor de mayor longitud, y siendo capaz de almacenar cualquiera de los tipos que se especificamos en la declaración .

La sintaxis de esta cláusula es la siguiente :

```
union nombre_union {
    tipo_campo_1      nombre_campo_1;
    tipo_campo_2      nombre_campo_2;
    ...
    tipo_campo_n      nombre_campo_n;
};
```

Por ejemplo

```
#include <stdio.h>

int main( void )
{
    union Compartido{
        int entero; /* 32 bits (4 bytes) en i386 */
        char caracter; /* 8 bits (1 byte) en i386 */
    } variableA;

    union Compartido variableB;
    variableA.entero = 0xFFFFFFFF;
```

```

    printf ("%x : %d - %x : %c\n",variableA.entero, variableA.entero,
variableA.caracter, variableA.caracter);

    variableA.caracter = 0xAA;

    printf ("%x : %d - %x : %c\n",variableA.entero, variableA.entero,
variableA.caracter, variableA.caracter);

    /*Salida
    * ffffffff : -1 - ffffffff : "y
    * ffffffff : -86 - ffffffff : a
    */

    return 0;
}

```

```

union udatos_personales {
    int     dni;
    char  sexo;
    float promedio;
};

union registro{
    int     error1;
    short errors;
    char  errorc;
};

int main(int argc, char** argv)
{
    union udatos_personales dat;
    union registro reg;

    dat.sexo = 'C';
    dat.dni = 27941166;
    dat.promedio = 12.45;
    /* Aqui dat contiene solo el valor 12.45 en el campo promedio todos
    los anteriores fueron borrados, osea que solo conserva el ultimo
    valor */

    reg.errorc = 4;
    reg.errors = 10;
    reg.error1 = 233333445;
    /* Aqui lo que conservaremos sera el error1 ya que fue el ultimo que
    guardamos */

    return 0;
}

```

Es importante destacar la propiedad de manejar múltiples tipos de datos pero solo conservar el valor de uno de ellos del **union** característica principal que lo distingue de un **struct**.

1.10.5 Struct .vs. Union

Una tema importante a analizar llegado el momento es: cuales son las diferencias entre un **struct** y un **union** ya al momento de describir cada uno vimos un poco como es que era su sintaxis y que uno conservaba todos los valores y otro solo el ultimo, pero ahora seria bueno analizar también como se comportan en memoria miremos como se ubican cada uno de ellos en memoria (tomaremos como referencia una plataforma 32bits) :

<i>struct</i>	<i>union</i>						
<pre>struct registro { int dni; double promedio; char sexo; };</pre>	<pre>union registro { int dni; double promedio; char sexo; };</pre>						
<p style="text-align: center;">Organización en memoria</p> <table border="1" style="margin: auto;"> <tr><td style="text-align: center;">dni</td></tr> <tr><td style="text-align: center;">promedio</td></tr> <tr><td style="text-align: center;">sexo</td></tr> </table> <p>Vemos como se reserva lugar en memoria para las 3 variables.</p>	dni	promedio	sexo	<p style="text-align: center;">Organización en memoria</p> <table border="1" style="margin: auto;"> <tr><td style="text-align: center;">promedio</td></tr> <tr><td> </td></tr> <tr><td> </td></tr> </table> <p>Vemos como se reserve lugar para la variable mas grande</p>	promedio		
dni							
promedio							
sexo							
promedio							

1.11 Funciones & Alcance de funciones

En ANSI C las funciones se declaran de la siguiente forma

```
tipo_valor_de_retorno  Nombre_Función(  tipo_parametro, X,  tipo_parametro, X,  ....,
                                         tipo_parametro, Xn );
```

Esto nos esta indicando que para declarar la función Suma, Resta, Multiplicación y Division podríamos escribir lo siguiente :

```
float    Suma( float x, float y);
float    Resta( float x, float y);
float    Multiplicacion( float x, float y);
float    Division( float x , float y);
```

Ahora una función en C se define de la siguiente manera, se repite la declaración, pero en vez de finalizar la línea con un ; para terminar la sentencia lo que haremos será abrir una llave e implementar la misma. Para el ejemplo anterior de las operaciones matemáticas tendremos:

```
float    Suma( float x, float y)
{
    return    x + y; }

float    Resta( float x, float y)
{
    return    x - y; }
```

```
float      Multiplicacion( float x, float y)
{
    return      x * y; }

float      Division( float x , float y)
{
    return      x / y; }
```

1.11.1 Pasaje de parámetros

Como sabemos C posee el pasaje de parámetros por valor, esto genera una problemática, ya que simplemente no podemos modificar el valor de una variable que pasamos como parámetro a una función, como es que se soluciona esto?, utilizando punteros , como este tema lo veremos mas adelante solo introduciremos los elementos básicos para hacer el pasaje dejando todas las explicaciones teóricas para la unidad de punteros.

Analicemos el clásico ejemplo de la función swap

```
void      Swap( int x, int y)
{
    int temp = x;
    x = y; y = temp;
};
```

Lo que podemos observar es que una vez cerrada la llave del bloque de la función swap, perderemos los cambios, ya que como son variables automáticas, al llegar a la llave, se desapilaran y serán destruidas.

Como debemos hacer para modificar realmente los valores?, de la siguiente forma :

```
void      Swap( int *x, int *y)
{
    int temp = *x;
    *x = *y; *y = temp;
};
```

Y ademas debemos reemplazar en la llamada a la funcion :

```
int a, b;
a = 10; b =7;
/*Swap(a,b); reemplazamos esta llamada por */
Swap(&a, &b);
```

1.12 Entrada y salida standard

La funcion printf de la stdio, es la utilizada (como ya vimos) para imprimir por pantalla con formato. Toma como argumento una string a ser formateada y opcionalmente una lista indeterminada de variables para imprimir. Las variables son impresas de acuerdo a las especificaciones de la cadena. Veamos varios ejemplos:

```
#include <stdio.h>

int main()
{
    int numero = 5;
    float radio = 2;
    char nombre[15] = "Jacinto";
```

```

char inicial = 'J';
printf("Hola mundo\n");

printf("Mi numero es %d\n", numero);
printf("Mi nombre es %s\n", nombre);
printf("Mi inicial es %c\n", inicial);

printf("El area de un circulo con radio %f es %f\n", radio, 3.14*radio*radio);

printf("Hola %s, tu inicial es %c\n", nombre, inicial);
return(0);
}

```

A continuacion una lista de modificadores de formato:

Especificador	Tipo de argumento
%d	int
%f	float o double
%e	float o double, salida en notacion cientifica.
%c	char
%s	string

En cuanto a la entrada standard, utilizamos la funcion scanf. Tiene similar funcionamiento a printf en cuanto a sus parámetros. Veamos un ejemplo.

```

#include <stdio.h>

int main()
{
    float radio;
    char nombre[15];
    int numero;

    printf("Ingresar el radio del circulo: ");
    scanf("%f",&radio);

    printf("Un circulo de radio %f tiene area %f\n",radio,3.14*radio*radio);

    printf("Ingresar un numero de 1 a 100: ");
    scanf("%d",&numero);

    printf("El numero es el %d\n",numero);

    printf("Ingresa tu nombre: ");
    scanf("%s",nombre);

    printf("Hola %s\n",nombre);

    return(0);
}

```

También existen las siguientes funciones útiles para la entrada y salida en C.

- getc
- putc
- gets
- puts

1.13 Archivos

El puntero llamado file handler (FILE*), apunta a una estructura en memoria que contiene datos como la dirección de un buffer con el contenido del archivo, la posición en ese buffer, errores que se pudieran haber producido en la lectura y/o escritura.

```
#include <stdio.h>

int main( void )
{
    FILE* fp;
    int caracter;

    if ( (fp=fopen("test.cpp", "r"))==NULL )
    {
        printf("ERROR\n");
        exit(1);
    }

    fseek(fp,0,SEEK_END);
    printf("Tamaño del archivo : %d bytes.\n",ftell(fp));
    rewind(fp);

    while( (caracter=fgetc(fp))!=EOF )
        putchar(caracter);

    fclose(fp);
    return 0;
}
```

Los modos de apertura de archivo, se indican a continuación:

Modificador	Modo
r	Lectura
w	Escritura(crea el archivos si no existe)
a	Append(crea el archive si no existe)
r+	Apertura para lectura y escritura, empieza al principio del archivo
w+	Apertura para lectura y escritura (sobreescribe el archivo)
a+	Apertura para lectura y escritura (hace append si existe el archivo)
b	Para lectura escritura de archivos binarios (usar con alguno de los anteriores)
t	Para lectura escritura de archivos en modo texto (la contrapartida del "b").

Las funciones fread y fwrite sirven para leer y escribir respectivamente en un archivo. Debe especificarse un buffer, su tamaño, la cantidad de lecturas y el file handler de donde leer o escribir.

```
#include <stdio.h>

#define SIZE 1000

int main() {
    FILE *sourceFile = NULL;
    FILE *destinationFile = NULL ;
    char buffer[SIZE];
    int n = 0;

    sourceFile = fopen("file.c", "r");
    destinationFile = fopen("file2.c", "w");
```

```

if(sourceFile==NULL) {
    printf("Error: No se pudo leer file.c.\n");
    return 1;
}
else if(destinationFile==NULL) {
    printf("Error: no se puede crear el archivo para escritura\n");
    return 1;
}
else {
    n = fread(buffer, 1, SIZE, sourceFile); /* Leo todo el texto */
    fwrite(buffer, 1, n, destinationFile); /* y lo pongo en file2.c */
    fclose(sourceFile);
    fclose(destinationFile);

    destinationFile = fopen("file2.c", "r");
    n = fread(buffer, 1, SIZE, destinationFile); /* leo file2.c */
    printf("%s", buffer); /* muestro el contenido */

    fclose(destinationFile);
    return 0;
}
}

```

1.13.1 Ejemplo de aplicación

Se cuenta con un archivo que contiene los datos de empleados de la compañía, se quiere hacer una aplicación que lea desde el disco los mismos, y que muestre por pantalla todos los nombres de los empleados y luego el promedio de edad de todos ellos.

La estructura de registro del archivo es la siguiente :

Nombre	30 caracteres
Apellido	30 caracteres
edad	short unsigned
sexo	char
domicilio	30 caracteres
dni	int unsigned

Se requiere procesar uno a uno los registros de todo el archivo hasta su fin, e ir mostrando los nombres, al finalizar, mostrar el promedio de edades. Se recibe por línea de comandos el nombre del archivo a procesar.

Resolución:

```

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

typedef struct    datos_personales {
    char          nombre[30];
    char          apellido[30];
    unsigned short edad;
    char          sexo;
    char          domicilio[30];
    unsigned int  dni;
} dpersonales;

int main(int argc, char** argv)
{

```

```
char          filename[40];
FILE*        fsource;
dpersonales  registro;
unsigned short SumaEdad, TotalRec;

SumaEdad = TotalRec = 0;

strcpy(filename, argv[1]); /* Como restriccion tenemos que el filename
                           no sera mayor a 39 caracteres */

fsource = fopen (filename, "r+b"); /* Lectura binaria */

if ( fsource == NULL )
    return -2; /* No se pudo abrir archivo */

while (!(feof(fsource) ))
{
    if ( fread( (void*) &registro, sizeof(registro), 1, fsource ) );
    {
        SumaEdad += registro.edad;
        ++TotalRec;
        printf("%s %s \r\n",registro.nombre, registro.apellido);
    }
}
printf("El promedio de edades es : %i", SumaEdad / TotalRec);

fclose(fsource); /* Liberamos los recursos */

return 0;
}
```