Una Breve Introducción al Lenguaje “C”

Ricardo Becerril
Instituto de Física y Matemáticas
Universidad Michoacana de San Nicolás de Hidalgo
Abril de 2005

Índice

1. Introducción **3**

2. Corriendo un Programa en C **4**

3. Tipos de Variables, y las funciones printf() y scanf() **5**

3.1. Variables tipo int 9
3.2. Variables tipo float y double 10
3.3. Variables tipo char 10
3.4. printf() y scanf() 11

4. Flujos de Control **13**

4.1. Condicionales if() e if()-else 13
4.2. Ciclos o “Loops” *for* y *while* 14
4.3. El uso del *break* 18

5. Algunos Operadores **19**

5.1. Operadores de Incremento y Decremento 19
5.2. Operador Modulo % 21
5.3. Operadores Lógicos y Relacionales 21
5.4. Operador Condicional ? : 23
5.5. Jerarquía o Prioridad de las Operaciones 23

6. Arreglos y Apuntadores **25**

6.1. Apuntadores 27

6.2. Relación entre Arreglos y Apuntadores 29

7. Funciones **30**

7.1. Funciones llamadas por Referencia 36

7.2. Arreglos como Argumentos de Funciones 37

8. Arreglos 2-Dimensionales **37**

8.1. Declaración de Parámetros, e Inicialización 38

8.2. El Uso Apropiado de Arreglos 2-Dimensionales 39

9. Guardando Datos en un Archivo **43**

10. Funciones Matemáticas **45**

10.1. Compilando cuando se usan funciones matemáticas 46

11. Uso de Assertions **46**

12. Algunos Ejercicios **47**

1. Introducción

En estas notas se da una breve introducción al lenguaje de programación C. El propósito de este mini-curso es proveer lo básico del lenguaje para que el lector pueda escribir y/o leer programas en C. Al aprender un lenguaje, la primera pregunta que puede venir a la mente es: por qué este lenguaje?, en este caso la pregunta es *por qué C?*. Algunas de las virtudes y características generales de este lenguaje son

- C es el idioma natal del sistema operativo más usado en el ámbito científico: Unix. Cuando uno usa FORTRAN o Pascal en una máquina Unix, es un programa C quien produce el programa ejecutable final.

- C es un idioma muy portable. Un código escrito en un sistema se puede compilar y correr en otro con una simple modificación ó ninguna. Esto no ocurre con otros idiomas tradicionales. C es indiscutiblemente, el lider en portabilidad. Los compiladores de C están disponibles para varios sistemas.

- C es un lenguaje breve. Tiene un conjunto de operadores poderosos y eficientes que hace que su escritura sea breve, resultando en códigos mas compactos, que pueden correr más rápido.

- C es la base de C++ y Java. Si uno aprende C, adentrarse a los lenguajes C++ y/o Java ya es relativamente sencillo.

- Por supuesto que este lenguaje no está excento de crítica. Sin embargo, es un lenguaje elegante, y uno puede vivir facilmente con sus imperfecciones.

Durante los últimos 15 años, C se ha convertido en uno de los lenguajes de programación más importantes. Su uso ha aumentado porque la gente lo prueba y le gusta. En el mundo científico su uso va en aumento también. Otros

lenguajes ampliamente usados hasta ahora, como FORTRAN, han tratado de cambiar para parecerse a C, dando por resultado FORTRAN-90.

Las presentes notas lo invitarán a la experimentación, se proveerán de ejemplos y su descripción detallada, así como ejercicios para que usted practique.

Mientras usted aprende C, irá reconociendo las virtudes del lenguaje y usándolo ampliamente.

Bienvenidos al mundo del idioma C

2. Corriendo un Programa en C

El primer paso en la escritura de un programa es definir los objetivos. Debe establecer claramente que quiere que haga su programa. Con esa idea en mente, uno organiza su programa en modulos, como por ejemplo: obtener información, hacer algún tipo de cálculos, y reportar o almacenar los resultados. Una vez que usted sabe que quiere hacer y cómo va a realizarlo, necesitará un editor (uno de los editores más usados es *emacs*) para escribir su código en el lenguaje C, éste es conocido como *archivo de código fuente*. Es altamente recomendable documentar su código, el uso de comentarios permitirá hacer entendible a otros y a usted mismo, qué hace su código y cómo lo hace.

El paso siguiente es compilar su código fuente. El *compilador* es un programa cuyo trabajo es convertir el código fuente en uno *ejecutable*, un código que la computadora puede entender. El compilador también checa si el *codigo fuente* está bien escrito en el lenguaje usado. Si hay errores, se los reportará y no se creará un *ejecutable* hasta que el *código fuente* este libre de errores.

Suponga que usted ha escrito un código fuente llamado *programa.c*, para compilarlo uno da la siguiente instrucción:

```
cc -o program program.c
```

Si no hay errores, se producirá el ejecutable *program*. Si sólo se escribe

```
cc program.c
```

el ejecutable que se crea, se llamaría *a.out*. Dependiendo del camino (path) para llegar a su programa, es posible que para correr su ejecutable tenga que escribir un punto diagonal antes del nombre del ejecutable

```
./program
```

Si se usa el archivo *math.h* donde se tiene información de funciones matemáticas, se deberá compilar con la instrucción

```
cc -o program program.c -lm
```

3. Tipos de Variables, y las funciones printf() y scanf()

Considere el siguiente programa

```
/* Maratón en Kilometros */
#include <stdio.h>

main( ) {
        int millas, yardas;
        float kms;
        millas = 26;
        yardas=385;
```

```
    kms = 1.609*(millas+yardas/1760.0);
    printf("un maraton tiene %f kilometros \n",kms);
}
```

A continuación se proveen los detalles de este programa

```
/* Maratón en Kilometros */
```

Este es un comentario, inicia con el simbolo /* y termina con */. Como hemos mecionado, es muy importante documentar el programa para hacerlo entendible para quien lo use.

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

Incluye una copia del archivo *stdio.h* en este punto del código. Este archivo lo provee el sistema C. *stdio* en Inglés significa *standard input and output* (entrada y salida estandard). Se incluye este archivo porque contiene información de la función printf() que se usa en el programa.

```
main()
```

Es la función donde se escribe el programa principal. La ejecución de un programa siempre empieza con esta función.

```
{
```

Con un paréntesis curvo { inicia una función, y se termina con }. También delimita expresiones como condicionales o ciclos (loops).

int millas, yardas;

Declaración de variables. *int* es una palabra-clave (palabras reservadas) que el programador *no puede usar como variables* pues ya tienen un significado especial para el compilador. En este caso, le dicen al compilador que millas y yardas son enteros.

float kms;

Es una declaración de otra variable, llamada "flotante". float es también una palabra-clave (hay 32 en en el lenguaje C). Le dice al compilador que las variables que le siguen son números reales.

millas = 26;
yardas = 385;

Son expresiones de asignación. El signo = es un operador de asignación. No es un signo de igualdad, pues una expresión como millas=millas+1 no tiene sentido en matemáticas si = significara igualdad. Así que millas=millas+1 debe entenderse como que a "millas se le da el valor que traía más uno". En el programa, se le asigna a las variable millas y yardas los números enteros constantes 26 y 385

kms = 1.609*(millas + yardas/1760.0);

Es otra expresión de asignación. Los operadores *, +, /, se refieren a la multiplicación, suma y división.

printf("un maraton tiene %f kilometros \n",kms);

Es una expresión que llama a la función printf(). Entre comillas está la *cadena de control*. Dentro de la cadena está la especificación de conversión o formato %f , que indica que ahí se colocará un número del tipo flotante, en este caso, se colocará el valor de kms calculados ya en el programa. Nótese que cada expresión en main() termina con un punto y coma. No es indispensable que después de un punto y coma uno cambie de renglón, uno podría, si lo desea, escribir seguido y el compilador no se molesta, pero tal programa sería difícil de leer.

En C, división de enteros resulta en enteros y el residuo se desecha. Así que 9/2 tiene un valor entero de 4. La expresión 9.0/2 es un real dividido por un entero y al evaluarse se convierte en un real cuyo valor será 4.5

kms = 1.609*(millas + yardas/1760);

Esto producirá un error (o "bug") de difícil detección, pues yardas (385) es de tipo entero dividido por otro entero (en este caso 1760), lo cual daría por resultado 0, y eso no es lo que se quiere. La expresión correcta, que dará lo que queremos es

kms=1.609*(millas + yardas/1760.0);

pues un entero, en este caso yardas, dividido por el real 1760.0 se promueve a double.

printf() y scanf()

Son funciones de entrada y salida (input y output). Estas funciones son parte del sistema C, existen en una librería y están disponibles dondequiera que se encuentre un sistema C. Los prototipos de estas funciones están en el archivo *stdio.h* que se incluyó en un principio.

Los argumentos de estas funciones son dos: la cadena de control (que van siempre entre comillas) y los argumentos segundos (separados por una coma).

3.1. Variables tipo int

int se refiere a los números enteros. Típicamente un entero se almacena en 2 bytes (= 16 bits) o bien en 4 bytes (= 32 bits), asi que los valores que una variable tipo entero puede almacenar depende del sistema. Si i es una variable tipo *int* entonces

$N_{min} \leq i \leq N_{max}$

donde en máquinas de 2 bytes se tiene

$N_{min} = -2^{15} = -32768 \approx -32$ mil
$N_{max} = 2^{31} - 1 = 32767 \approx 32$ mil

y en máquinas de 4 bytes se tiene

$N_{min} = -2^{31} = 2147483648 \approx -2$ billones
$N_{max} = 2^{31} - 1 = 2147483647 \approx 2$ billones

Considere el siguiente código, qué piensa ud. que se aparecerá en la pantalla?

```
#include <stdio.h>
#define GRANDE 32000

main()
{
    int a = GRANDE, b;
    b = 2*a;    /* se sale del rango de los enteros? */
    printf(“b = %d \n”,b);
}
```

Cambie la definición de GRANDE por 2000000000 (2 billones) y vea que pasa.

3.2. Variables tipo float y double

Las variables tipo flotante y doble son números reales. En muchas máquinas, una variable tipo *float* tiene una precisión aproximada de 6 cifras significativas y un rango (aproximado) de 10^{-38} a 10^{38}. En otras palabras, una variable float se representa por

$$0.d_1d_2d_3d_4d_5d_6 \times 10^n$$

donde d_i es un digito y $-38 < n < 38$. Una variable tip *double* se representa (aproximadamente) en muchas maquinas como

$$0.d_1d_2....d_{15} \times 10^n$$

donde d_i es un digito y $-308 < n < 308$.

Algunos compiladores de C pueden asignar más espacio de memoria para variables del tipo *long double* que la que asignan a tipos *double*, aunque esto no ocurre necesariamente. Algunos compiladores implementan el *long double* como *double*. Una variable tipo *long float* sería equivalente a una variable *double*.

3.3. Variables tipo char

Variables tipo *char* son simplemente caracteres como letras, o signos de puntuación. Para imprimir las letras *abc* sobre la pantalla se usaría la función de entrada (o input) printf() (aunque hay otras funciones que pueden usarse, como putchar()):

printf("abc");

ó

printf(" %s","abc");

ó

```
printf(" %c %c %c",'a','b','c');
```

En la segunda forma se está usando un formato del tipo cuerda o *string*, esto es %s, que agrupa varios caracteres; y en la tercera posibilidad aparece un formato del tipo *char*, a saber %c. Nótese que en este caso, las letras que se imprimirán van entre comillas simples 'a', 'b', 'c'

3.4. printf() y scanf()

Damos aqui mas ejemplos del uso de printf() y scanf(). Considere el programa

```
#include <stdio.h>
main()
{
    int b= 35;
    float a= 0.0245;
    long float w=1.23456;
    printf("Introduzca los valores de A, w, y B \n");
    scanf(" %f %f %d",&a,&w,&b);
    printf("Ud. Introdujo A =%e, w =%f, y B =%d \n",a,w,b);
}
```

Al compilarlo aparecerá en la pantalla

```
Introduzca los valores de A, w, y B
0.0245   1.235   35
Ud. Introdujo A= 2.45e-02, w= 1.235456 y B= 35
```

Las conversiones o formatos que admiten las funciones printf() y scanf() se dan a continuación

Conversiones de printf() y scanf()	
c	como carácter
d	como entero
f	como número flotante
lf o LF	como número doble precisión
s	como cuerda
Conversiones sólo de printf()	
e	como flotante o doble en notación científica
g	e o f, el que sea mas corto

Estos son los formatos que se utilizan en la cadena de control de las funciones printf() y scanf(). Nótese que scanf, para números reales, NO admite el formato *double*, solo *float* o *long float*. Es importante tener esto en cuenta, pues si usted escribe

```
#include <stdio.h>
main()
{
    float a;
    double b;
    printf("Introduzca los valores de a y b\n");
    scanf(" %f %f",&a,&b);
    printf("Usted introdujo a = %f y b = %f\n",a,b);
}
```

si el compilador no se queja (muy probablemente no lo hará), entonces, qué

error cree que habrá?, qué cree que aparecerá en la pantalla?

4. Flujos de Control

Sirven para alterar el flujo secuencial normal en un programa. Aqui veremos algunos de estos controles de flujo.

4.1. Condicionales if() e if()-else

if y if-else proveen acciones alternativas.

while y *for* proveen mecanismos para realizar ciclos o *loops*.

Ambos requieren la evaluación de expresiones lógicas que pueden ser falsas o verdaderas. En C cualquier valor distinto de cero se considera que representa *verdadero*, y el valor cero representa lo *falso*

```
if( expresión)
        operación
```

Aquí, se evalúa primero la expresión del argumento del condicional if(), si esa expresión es verdadera (distinta de cero), entonces la *operación* que le sigue se realiza, si es falsa (igual a cero) el programa no la toma en cuenta y sigue con la siguiente instrucción.

```
if (expr)
       acción1
else
       acción2
```

Aquí, se evalúa primero la expresión del argumento del if(), si es verdadera, entonces la acción1 se lleva a cabo, si es falsa entonces es la acción2 la que se ejecuta. Así que en un *if else* una de las dos acciones siempre se ejecuta.

4.2. Ciclos o "Loops" *for* y *while*

while(expr)
    acción

Aquí, se evalúa primero la expresión del argumento del while() (que significa "mientras"), y mientras esa expresión sea verdadera la "acción" se repite. Uno debe ser muy cuidadoso en asegurarse que en el momento planeado, esa expresión será falsa para no tener un ciclo (o loop) infinito.

for(expr1;expr2;expr3)
    acción

Cuando se usa un *for*, la expresión *expr1*, generalmente se usa para inicializar variables, uno puede dejar ese espacio vacio y el compilador no se quejará. La expresión *expr2* debe ser una expresión lógica y es la que se evalúa en segundo lugar. Si es verdadera, la acción (o conjunto de acciones limitadas por { y }) se lleva a cabo, y a continuación se evalúa la expr3, y nuevamente se evalúa la veracidad de expr2, hasta que sea falsa. Cuando la expr2 es falsa, el programa sale del ciclo y continua su flujo normal. La expr3 generalmente incrementa un valor almacenado.

for(expr1;expr2;expr3)
    acción

Es equivalente al *while* siguiente

```
expr1;
while(expr2){
    acción
    expr3;
}
```

El siguiente programa ilustra el uso de los condicionales y del *for.*

```
/* calcula el maximo, el minimo, y el promedio */
#include <stdio.h>
#include <stdlib.h>
main()
{
    int i;
    double x, min, max, sum, avg;
    if( scanf(" %lf",&x) != 1) {
       printf("no se encontraron datos\n");
       exit(1);
     }
    min = max = sum = avg = x;
    for( i=2; scanf(" %lf",&x) == 1; i += 1) {
        if( x <min)
          min = x;
        else if( (x >max)
          max = x;
        sum += x;
```

```
        avg = sum/i;
    }
    printf("min =%3.5f, min= %3.5f, avg =%3.5f\n", min, max, avg);
  if( scanf(" %lf",&x) != 1) {
          printf("no se encontraron datos \n");
          exit(1);
     }
}
```

Empecemos a revisar el programa,

```
    if( scanf(" %lf",&x) != 1) {
        printf("no se encontraron datos\n");
        exit(1);
    }
```

Como mencionamos anteriormente, el tipo *long float* lf, es equivalente a double, y scanf(" %lf",&x) != 1) lee un daro que el usuario provee y lo almacena en la dirección de x, (&x se lee como "la dirección de x" & es el operador "dirección"). La función scanf() regresa un valor entero *int*, que es el número de conversiones exitosas que realiza. Si no lee nada, se imprime un mensaje y sale del programa con la función exit(). Información sobre esta función se localiza en el archivo stdlib.h (standard library), por lo que se incluyó al inicio del programa. La funció exit() toma un solo argumento tipo int, se le da un entero diferente de cero (en este caso se escribe en el código el número 1) si la salida no se considera normal, como este sería el caso cuando no lee datos.

```
  min = max = sum = avg = x;
  for( i=2; scanf(" %lf",&x) == 1; i += 1)
```

Aquí se inicializa la variable i con el valor 2. Luego se hace una prueba para ver si la expresión lógica scanf(" %lf",&x) == 1 es verdadera. Si scanf()

puede leer caracteres de la corriente de entrada standard, lo interpreta como un *long float* o *double* y coloca ese valor en la dirección de x, se ha hecho una conversión exitosa y scanf() regresa el valor 1. == es el operador lógico "*igual a*" , en este caso checa si es igual a 1, que es el entero que regresa scanf() pues lee un valor exitosamente. Un error muy común es usar = en vez de ==, cuidado!. En caso de que scanf() regrese el valor 1, el ciclo se repite, sino, se termina.

Hemos usado el simbolo " += ". Ahora explicamos su uso. La expresión general

variable OP = expresión

donde OP puede ser la suma +, resta -, multiplicación *, o división /, es equivalente a

variable = variable OP *expresión*

En el caso del ejemplo i += 1, es equivalente a i = i + 1

Si escribimos i -= 1, esto es equivalente a i = i-1, étc.

```
if( x <min)
        min = x;
else if ( x >max)
        max = x;
```

Esta construcción es un *if-else*: La acción después del else es en sí misma un condicional if(). Aquí el min y max se actualizan con cada lectura que se realiza de los datos.

Se puede crear un archivo de datos, por ejemplo "datos", compliamos el programa con la instrucción:

cc -o promedio promedio.c

y se crea el ejecutable *promedio*. Si ahora se da el comando

promedio < datos

los datos de entrada se *redireccionan* hacia el programa ejecutable *promedio*, que entonces empieza a correr.

4.3. El uso del *break*

Como hemos dicho

for(expr1; expr2; expr3)

admite tres expresiones separadas por punto y coma. Pero cada expresión puede consistir en varias expresiones separadas por comas, o bien estar vacía. Por ejemplo, en

for(suma=0.0,i=0,j=50; suma<=100.0 && j>= 30; suma += 0.5, j -=1)

se tiene que expr1 consiste de suma=0.0, i=0, j=50 cada una de estas inicializaciones están separadas por comas. La expr2 consiste de de una sola expresión condicional "suma<=100.0 && j>= 30" y expr3 tienen 2 expresiones cada una separadas por comas.

for(;;)

Esto es válido pero en el cuerpo del ciclo debera haber una instrucción que

termine con el mismo. Una manera de terminar un ciclo, como ya hemos dicho, es por medio de verificar que el segundo argumento *expr2* en el for() no se cumpla y el ciclo termina. Otra manera de terminar el ciclo "anormalmente", sería con el uso del *break*, éste interrumpe el flujo del ciclo y lo termina. Si se tiene una cadena de ciclos, *break* termina el ciclo interior (en el cual se encuentre, pero se continuan los otros).

```
while(1){
    scanf(" %f",&x);
    if(x <0.0) break;
    printf("sqrt( %f) = %f \n",x,sqrt(x));
  }
```

Como el argumento de while es 1 (que significa *verdadero*) el ciclo sería infinito, pero se interrumpe cuando se lea un número negativo (hay que tener mucho cuidado en asegurarse de que este caso, en efecto, se dé). Este es el uso típico del *break*.

5. Algunos Operadores

Hasta el momento ya hemos visto algunos operadores como la suma +, resta -, mutiplicación *, división /, asignación = , igual a == . A continuación veremos algunos más y algunas reglas importantes de los operadores.

5.1. Operadores de Incremento y Decremento

Son operadores que se usan como pre-fijos o post-fijos. Si *val* es una variable de tipo int, entonces ++val y val++ son expresiones válidas. Se aplican a variables, no a constantes o a expresiones ordinarias, así que

++i , a - -   son expresiones válidas, pero

367++ , ++(a*b+1) no lo son.

Cada expresión: ++i, i++ tiene un valor y además, cada una ocasiona que se le incremente el valor a i por 1.

++i : aumenta el valor de i por 1, y toma ese valor aumentado para la expresión

i++: esta expresión primero toma como valor, el valor de i, y luego aumenta el valor de i por 1

Observe la siguiente sección de código

```
int a, b, c=0;
a = ++c;
b = c++;
printf(" %d %d %d \n", a, b, ++c);
```

Son los números 1 1 3 los que se imprimen en la pantalla. El operador " - - " disminuye el valor de i en 1.

Ejercicio: Qué cree se imprimirá con ?

```
int a, b=0, c=0;
a = ++b  +  c++;
printf(" %d %d %d\n",a,b,c);
a = b++  +  --c;
printf(" %d %d %d\n",a,b,c);
```

5.2. Operador Modulo %

a % b da el residuo de dividir el entero “a” entre el entero “b”. Así, por ejemplo, 10 %3 tiene el valor 1.

Ejercicio. Qué hace la siguiente sección de programa ?

```
for(i=1;i<=50;i++) a[i]= 0.5*i;
for( i=1; i<=50 ; i++){
    if( i % 5 == 0) printf("\n");
    printf("a[ %d]= %f",i,a[i]);
}
```

En efecto, lo que hace primero es asignar un valor a a[1], a[2],..., a[50], y luego imprimir 4 valores de a[i] en un renglón en pantalla y después se salta renglón, asi que se imprimen cuatro valores de a[i] por renglón, esto lo hace con el condicional if(i %5 ==0) pues i %5=i si $1 \leq i \leq 4$ y el argumento del if() es falso y la acción de saltarse de renglón no se ejecuta, cuando i=5 el valor de i % 5 es cero, así que el argumento del if() es verdadero y la acción que le sigue se ejecuta, por lo que se imprime un \n cuyo resultado es saltarse de renglón. El argumento del if(), esto es i %5, vuelve a ser cero cuando i=10, 15, 20,..., étc y entonces hay salto de renglón.

5.3. Operadores Lógicos y Relacionales

Los *operadores relacionales* son: < menor que, > mayor que, <= menor o igual a, >= mayor o igual a. Tienen el significado matemático conocido.

Los *operadores de igualdad* y de *no igualdad* son: == que significa *igual a*, y != que significa *no igual a.*

El *operador lógico*: && significa “y”, el operador || significa “o”. Por ejemplo,

si x va variando de 0 a 10 en pasos de 0.1, y si se quiere imprimir algo cuando x este en el intervalo cerrado [1.0,1.5], se puede usar la siguiente sección de programa

```
for(x=0.0; x<=10.0; x += 0.1){
      if(x >= 1.0 && x <=1.5)
         printf("En x= %f, f(x)= %f \n",x,sin(x*x)/x);
      operacion1;
      operacion2;
}
```

El argumento en if(x >= 1.0 && x <=1.5) garantiza que se imprimirán los valores de x y f(x) sólo cuando x este en el interválo [1.0,1.5]. Nótese que en este caso, el loop consta de un condicional if() y de dos operaciones mas, por lo que, el ciclo completo: el if(), operación1 y operación2, debe encerrarse entre paréntesis curvos { }.

Pueden usarse varios operadores condicionales a la vez. En el ejemplo anterior, si se quieren imprimir los valores de x y f(x) cuando x esté en [1.0,1.5] ó cuando esté en [2.5,3.0], el argumento del if() podría ser:

```
(x >=1.0 && x<=1.5) || (x>= 2.5 && x<=3.0)
```

Los operadores && y || son binarios, es decir necesitan tener una expresión antes y despues de ellos: expr1 && expr2. El *operador lógico de negación* es " ! ", y es un operador "unario" pues solo antecede ó se "aplica" a una expresión: !expr lo cual trae por resultado la negación de dicha expresión. Por ejemplo != significa "no igual" o "diferente a". !5 da por resultado 0, y !(!5) es 1. También ! se puede usar en expresiones lógicas más elaboradas como !(a<b || c<d)

5.4. Operador Condicional ? :

Este operador toma tres expresiones, su forma general es:

expr1 ? expr2 : expr3

Primero se evalua expr1, si es verdadera, se evalua expr2 y ese es el valor de la expresión condicional como un todo. Si expr1 es falsa, se evalúa expr3 y ese el es valor de toda la expresión condicional

```
if(y <z)
    x=y;
else
    x=y;
```

Así se asigna a x el mínimo entre la variable “y” y la variable “z”. Es última expresión es equivalente a

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

El tipo de variable de la expresión total lo determina expr2 y expr3.

5.5. Jerarquía o Prioridad de las Operaciones

Los operadores tienen reglas de jerarqía y asociatividad. Estas reglas se siguen para determinar el orden en que las expresiones se han de ejecutar. Las expresiones dentro de paréntesis tienen prioridad y se ejecutan primero, por esta razón, es bueno incluirlos para clarificar el orden en que se ejecutan las operaciones. En la expresión

8 * 2 + 4

puesto que * tiene mayor jerarquía que + , la multiplicación se realiza primero

y después la suma, dando por resultado 20. Esta expresión es equivalente a

(8 * 2) + 4

pero si escribimos

8 * (2 + 4)

el resultado será 48, pues la suma lo encerrado en paréntesis se ejecuta primero y el resultado se multiplica por 8. Los operadores binarios + y - tienen la misma prioridad, la regla de *asociatividad* "*de izquierda a derecha*" se usa en expresiones como

5 + 1 - 3 + 6 - 2

en que aparecen operadores con igualdad de prioridad. Esta regla *de izquierda a derecha* significa que las operaciones se estarán realizando de izquierda a derecha,es decir, una expresión equivalente sería

(((5+1)-3)+6)-2

así, se realizaría la suma 5+1 primero (el paréntesis interior tiene prioridad) luego se le resta el 3 (el siguiente paréntesis sigue en prioridad) para sumarse después el 6, y finalmente restar 2. Se ha ejecutado la expresión de izquierda a derecha.

En la tabla que sigue se dan las reglas de prioridad y asociatividad de algunos operadores

Prioridad y asociatividad de operadores	
Operador	Asociatividad
() ++ - - (como postfijos)	izquierda-derecha
+ - (unitarios) ++ - - (como prefijos)	derecha-izquierda
* / %	izquierda-derecha
+ -	izquierda-derecha
= += -= *= etc	derecha-izquierda

Todos los operadores que aparezcan en una linea dada tiene igual prioridad o jerarqía entre ellos, por ejemplo *, /, y % tienen la misma prioridad, y su regla de asociatividad es como se indica en la segunda columna, en ese caso de izquierda a derecha. Pero estos operadores tiene mayor prioridad que los que se encuentran en lineas más abajo, en este caso *, /, % tienen mayor prioridad que +, - que están en la siguiente linea abajo, y que =, +=, étc, que están aún más abajo.

Cuando decimos prefijo ó postfijo, nos referimos por ejemplo a ++m, y m++, respectivamente.

Cuando decimos que un operador es binario, es que se requiere de dos cantidades con quien operar: a + b, aquí, el + es binario. Un operador unitario requiere de una sola cantidad: -a, es unitario, cambia el signo de a.

Nótese que los operadores unitarios +, -, tienen mayor prioridad que los binarios +, - Entonces, considerando la expresión

```
-a * b - c
```

el primer signo menos actúa como unitario, y el segundo como binario, por la prioridad dada en la tabla, primero se cambia el signo de a, se multiplica por b, y finalmente se le resta c. Una expresión equivalente será

```
((-a)*b)-c
```

6. Arreglos y Apuntadores

Los arreglos 1-dimensionales son un conjunto de datos que tienen un índice para distinguirlos. Los índices en C empiezan a correr desde 0. Si se declara una arreglo como flotante

```
float a[N];
```

Esta declaración aparta espacio para $a[0], a[1], ..., a[9]$ diez elementos. Es una buena practica definir el tamaño del arreglo como una constante simbólica. Para ello se debió asignar valor a N previamente. Sobrepasar el tamaño de un arreglo es un error común al programar, uno puede pensar que la declaración "float $a[10]$" significa que el último elemento del arreglo es $a[10]$, y cometer el error de sobrepasar el tamaño.

Para inicializar los arreglos que tengan valores diferentes, se escribe, por ejemplo

```
float f[5] = { 0.0, 1.0, 2.0, 3.0, 4.0 };
```

Si todos los elementos son enteros e inicialmente tienen el valor 2, se escribe

```
int a[10] = { 2 };
```

Un ejemplo del uso de arreglos:

```
#include <stdio.h>
#define N 5

main() {
    int i, j, score[N]= { 99, 87, 85, 94,70 }, sum;
    for(sum=0, i=0;i<N; ++i)
       sum += score[i];
    printf(" %5.2f es el promediobackslashn",(double) sum/N);
}
```

El programa obtiene el promedio de los datos almacenado en el arreglo

score[N]. Veamos algunos detalles del programa. En la función printf() se encuentra

(double) sum/N;

que es el operador *cast* cuyo efecto es convertir el valor entero de *sum* a un valor double. Este operador *cast* se realiza primero antes que la división, así que tendríamos una división entre un double y un entero, lo cual nos da por resultado un double. Sin este "cast" tendríamos una división entre enteros y la fracción (residuo) se descartaría, y en printf(), por el formato usado (%5.2f), el compilador marcaría un error.
Nótese que antes de la función main() se define el valor de N usando

#define N 5

Con esto, el valor de N se puede usar en cualquier parte de main() o en otras funciones usadas en el programa sin necesidad de declarar N en esas funciones ó asiganarle valor, pues ya lo tiene.

6.1. Apuntadores

Una variable se almacena con un cierto número de bytes en una localización particular ("dirección") de memoria en una máquina. Los apuntadores se usan para accesar memoria y manipular direcciones.

Un apuntador es la dirección de un objeto en memoria. Si p es un apuntador, el valor directo de p es una localización en memoria o "dirección".
Para ilustrar estas ideas, hagamos

int $a = 1, b = 2, *p;$

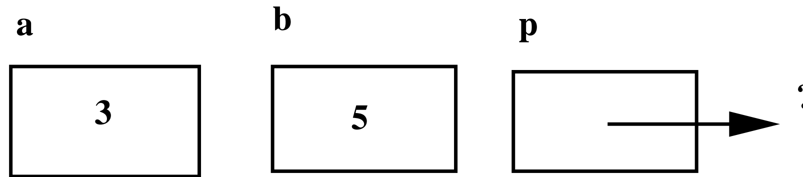

Figura 1:

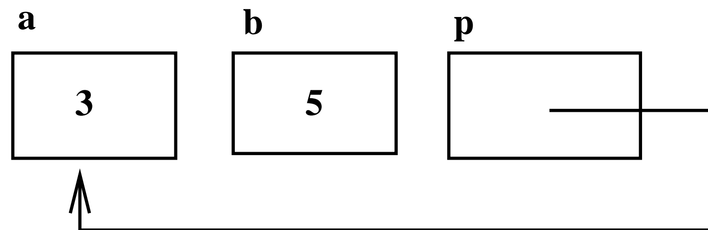

Figura 2:

Las variables a, b, p están almacenadas en memoria, y podemos pensarlas graficamente como se muestra en la figura 1.

Pensamos en p como una flecha. Pero como no se le ha asignado un valor no sabemos a que apunta. Si escribimos

```
p= &a;
```

esto lo leemos como "a p se le asigna la dirección de a", y el diagrama sería como en la figura 2.

Ahora hacemos la asignación: $b = *p$; ésta se lee "a la variable b se le asigna el valor a que apunta p", y como p apunta hacia a, entonces la expresión $b = *p$; es equivalente a $b = a$;

Un programa que ilustra la distinción $p = \&a$, y $b = *p$ es:

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

```
main()
{
    int i = 7, *p ; /* variable p de tipo entero */
    p = &i; /* p apunta a la dirección de i */
    printf(" %s %d %s %p\n"," Valor de i :", *p,"localización de i:", p);
}
```

Lo que se lee en la pantalla es:

Valor de i : 7
Localización de i : effffb24

La variable i debe declararse antes de tomar su dirección. %p es el formato preferido para el valor de un apuntador.

6.2. Relación entre Arreglos y Apuntadores

El punto central aquí es que el nombre de una arreglo por sí mismo es una dirección (en memoria). Pero a diferencia de un apuntador, que puede asignársele diferentes direcciones como valores, un arreglo es una dirección (o apuntador) que es *fijo*.

Cuando se declara un arreglo, el compilador da espacio contiguo en la memoria para contener todos lo elementos de arreglo. La dirección base del arreglo es la dirección del primer elemento $a[0]$. Si p es un apuntador, la asignación "p=a" equivale a "p = & $a[0]$" la dirección de $a[0]$. La asignación "p=$a+1$", equivale a "p=& $a[1]$" la dirección del siguiente elemento, a saber la dirección de $a[1]$.

7. Funciones

La parte de un código que describe lo que hace una función se llama *definición de la función.* No debe confundirse con la *declaración* de la función (las funciones también se declaran antes de usarse). En general, la definición de una función tiene la forma

```
tipo nombre(lista de parametros) {
declaraciones
acciones
}
```

El tipo, nombre y lista de parámetros constituyen el cabezal de la función. Y lo que esta entre paréntesis curvos constituye el *cuerpo de la definición* de la función. Por ejemplo

```
int factorial( int n) /* cabezal */
{ /* empieza el cuerpo */
    int i, product= 1;
    for( i=2; i<= n; ++i)
      product *= i;
return product;
}
```

Es una función que calcula el factorial de n. El primer int le dice al compilador que el valor que regresa la función es de tipo entero (y si es necesario lo convierte a entero). El nombre de la función es *factorial.* Solo hay un parámetro que es de tipo entero. La función se llama desde algún lugar del programa con la instrucción: factorial(4); como el factorial de 4 es 24, entonces tiene

sentido asignar a una variable entera el valor de la función

```
b = factorial(4);
```

es decir, b = 24. Si la función no regresa ningún valor, la función debe ser del tipo *void*. Si no se pone el tipo, la función será *int* por default. Los parámetros actúan como sostenedores de espacio. Si no hay parámetros, se escribe ahí *void* (que es una palabra clave y esta reservada en el sistema C).

Cualquier variable declarada en el cuerpo de la función será “*local*” a dicha función. Otras variables pueden declararse exteriores a la función, esas son variables “*globales*”.

Cuando el programa se topa con la declaración *return*, la ejecución de la función se termina y el control regresa al lugar desde donde se llamó a la función.

La declaración o *prototipo de la función* se da, por ejemplo como

```
double power(double x, double y);
```

El *prototipo de función* le dice al compilador el número y el tipo de argumentos que se van a pasar a la función, y el tipo de valor que se regresará. Pero los identificadores x, y no los usa el compilador. Por esto, también se puede escribir la declaración de la función así

```
double power(double , double );
```

Veamos el siguiente ejemplo

```
#include <stdio.h>

int main( void ) {
```

```
        int i, n;
        float max, min, x;
        float maximo(float x, float y);
        float minimo(float x, float y);
        void imprimir(int, float , float);
        printf("Introduzca el numero de valores N\n");
        scanf(" %d",&n);
        printf("Introduzca %d valores reales\n",n);
        scanf(" %f",&x);
        max = min = x;
        for(i = 2; i <= n; ++i){
           scanf(" %f",&x);
           max = maximo(max,x);
           min = minimo(max,x);
        }
        imprimir(n,max,min);
        return 0;
}

float maximo(float x, float y)
{
        if( x >y)
           return x;
        else
           return y;
}
float minimo(float x, float y)
```

```
{
    if( x <y)
        return x;
    else
        return y;
}

void imprimir(int n, float x, float y)
{
    printf("el maximo de la lista de %d números es %f y el minimo
}
```

Hasta ahora hemos escrito

main()

que es una función, por default (pues no se especifica su tipo) será del tipo int, y sin argumentos (es decir su argumento será del tipo void). Ahora, en este ejemplo, se escribió explicitamente su tipo *int* y su argumento *void*

int main(void)

por lo que, siendo del tipo int, se escribió al final return 0;

En el cuerpo de main(), se declaran las variables y las funciones que van a usarse. Y despues de main() se han escrito las definiciones de esas funciones. Si las declaraciones se hubieran escrito antes de main() las funciones ya no se declaran en main() ni en ninguna función definida posteriormente que usara a las funciones maximo(), minimo() o impirmir(). Declararlas antes de

main() es equivalente a declararlas “globalmente”

```
#include <stdio.h>
float maximo(float x, float y);
float minimo(float x, float y);
void imprimir(int, float , float);

int main( void ) {
        int i, n;
        float max, min, x;
        printf(“Introduzca el numero de valores N\n”);
        scanf(“ %d”,&n);
.........
}
.........
```

Los datos que pide el programa anterior, pueden ponerse en un archivo, digamos *datos* y se corre el programa escribiendo

```
ejecutable <datos
```

o bien el usuario los escribe en la pantall, en cualquier caso se leen con

```
for( i = 2; i <= n; ++i){
        scanf(“ %f”,&x);
        max = maximo(max,x);
        min = minimo(min,x);
}
```

En cada ciclo se lee un nuevo valor para x, entonces los valores de max y x pasan como argumentos de la función maximo() y el mayor de los dos se regresa y se asigna a max. Similarmente se pasan las variables min y x como argumentos de minimo() y se asigna a min el menor de los dos valores.

En C, los argumentos de las funciones siempre pasan sólo *su valor*. Es decir, se hace una copia del valor de cada argumento, y son estas copias las que se procesan en la función. Como consecuencia, la variables que pasan como argumentos a las funciones NO CAMBIAN en el ambiente o lugar donde se llamaron. Para ilustrar lo que decimos veamos el siguiente programa

```
#include <stdio.h>
int main( void )
{
        int a = 1;
        void change(int); /* función interna a main */
        printf("a =%d\n",a); /* se imprime 1 */
        change(a);
        printf("a =%d\n",a); /* que se imprime?*/
        return 0;
}

void change( int a)
{
   a = 100;
}
```

Cuando la variable a se pasa como argumento de la función se produce una copia de a. Es esta copia, y no la misma variable a la que pasa a la función. Así que en el ambiente en que se llama a la función (en main), la variable a NO cambia.

7.1. Funciones llamadas por Referencia

Para que una función llame variables por *referencia*, y puedan cambiar EN esa función, deben usarse apuntadores en la lista de parámetros. Y cuando se llama la función, deben pasarse direcciones de variables como argumentos.

```
#include <stdio.h>
int main(void)
{
    int i=3, j=5;
    void intercambia(int *, int *);
    intercambia(&i, &j);
    printf(" %d %d\n",i,j); /* se imprime 5 3 */
    return 0;
}

void intercambia(int *p, int *q)
{
    int tmp;
    tmp= *p;
    *p = *q;
    *q = tmp;
}
```

La función intercabia() toma dos argumentos del tipo pointer a enteros y no regresa nada. La variable tmp es local a esta función y es del tipo int (y se usa para guardar temporalmente un valor).

Con la expresión tmp= *p; a tmp se le asigna el valor al que apunta p.

Con " *p= *q" se le asigna al objeto al que apunta p, el valor al que apunta q.

Con “*q = tmp” al objeto al que apunta q se le asigna el valor de tmp.

De este modo se han cambiado en main los valores de objetos donde apunten p y q. Este ejemplo sencillo, muestra la manera típica de llamar funciones por referencia, se darán otros ejemplos mas adelante.

7.2. Arreglos como Argumentos de Funciones

En la definición de una función, un parámetro que se declara como un arreglo es realmente un apuntador. Como notación de conveniencia y claridad, el compilador permite notación de paréntesis al declarar pointers como parámetros.

```
double suma(double a[ ], int n)
{ /* n es el tamaño de a[ ] */
    int i;
    double sum= 0.0;
    for(i=0;i<n; i++) sum += a[i];
    return sum;
}
```

La declaración “double a[ ]” es equivalente a double “*a” Lo único que hace la función suma es sumar los valores de a[i].

8. Arreglos 2-Dimensionales

Aunque los elementos de un arreglo se almacenan en espacios contiguos uno tras de otro, es conveniente pensar en un arreglo 2-dimensional como una colección rectangular de elementos con filas y columnas.

Si declaramos

float $a[2][4]$;

pensamos que el arreglo se acomoda como

$a[0][0]a[0][1]a[0][2]a[0][4]$
$a[1][0]a[1][1]a[1][2]a[1][4]$

La dirección base del arreglo es $\&a[0][0]$, no a. Empezando en la dirección base, el compilador asigna espacio contiguo para 8 float's.

8.1. Declaración de Parámetros, e Inicialización

Cuando un arreglo multidimensional es un parámetro en una definición de función, todos los tamaños, exceptuando el primero, deben especificarse, de modo que el compilador pueda almacenar al arreglo correctamente. Considere la declaración

float $a[2][4]$;

Esto aparta espacio para una matriz 2×4. La siguiente función realiza la suma de todos lo elementos de la matriz y regresa ese valor.

```
float suma(float a[ ][4])
{
    int i,j;
    float sum;
    for(sum=0.0, i=0; i<2 ; i++)
      for(j=0;j<4;j++) sum += a[i][j];
    return sum;
}
```

Se presentan tres formas en que pueden inicializarse los arreglos de dos dimensiones al declararse

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

8.2. El Uso Apropiado de Arreglos 2-Dimensionales

Suponga que se declara una matriz 3×3

```
float a[3][3];
```

y quiere usarse una función que calcule su determinante. Si la matriz ya recibió valores, en algún momento quisieramos calcular el determinante utilizando

```
det = determinate(a);
```

cuya definición se vería como:

```
float determinante(float a[][3])
{
    Cuerpo de la funcion
}
```

Se mencionó que es necesario especificar el 3 para que el compilador trabaje bien. Así que nuestra función trabajará sólo para matrices 3×3. Si quere-

mos calcular el determinante de una matriz 4×4, se requiere escribir una nueva función, esto no es aceptable, queremos una función que calcule determinantes para una matriz de tamaño arbitrario. A continuación, se explica como usar arreglos 2-dimensionales adecuadamente para este fin.

Si uno empieza con un apuntador que apunta a un apuntador que apunta a un float, se puede construir una matriz de cualquier tamaño que uno quiera, y pasarlas a funciones diseñadas para trabajar con matrices de tamaño arbitrario. Aquí estan las instrucciones para crear el espacio para una matriz:

```
int i, j, n;
float **a, det, tr;
............ /* se obtiene n de algún lado */
a = calloc(n, sizeof(float *));
```

El tamaño de a no se necesita saber a priori, ni se necesita definir como constante. Una vez conocido el tamaño deseado, se usa la función calloc() de la librería standard (stdlib.h) para crear espacio para la matriz. Se llama a esta función de la siguiente manera:

```
calloc(n, sizeof( type))
```

que crea espacio para un arreglo de n elementos del tipo especificado (en este caso un apuntador que apunta a un float). Así que

```
a = calloc(n, sizeof( float *));
```

crea espacio para a, un arreglo de tamaño n, de apuntadores a float.

La expresión a[i][j] se usa para accesar el elemento de la i-ésima fila y j-ésima columna del arreglo. Ahora se puede pasar como argumento la matriz a en algunas funciones que ud. haya definido previamente, digamos

```
det = determinante(a,n);
tr = traza(a,n);
```

La función *traza* es sencilla de definir y aquí la escribimos

```
float traza(float **a, int n)
{
    int i;
    float sum;
    for(sum=0.0,i=0;i<n;i++) sum += a[i][i];
    return sum;
}
```

Espacio asignado por calloc() (que en inglés es contiguous allocation) no regresa al sistema cuando uno sale de la función, debe usarse free() explícitamente para regresar ese espacio. En este caso

```
free(a);
```

libera ese espacio al sistema. Como ejemplo, damos esta sección de programa

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

    float suma(int *a, int n)
{
    int i;
    float add;
```

```
    for(add= 0.0, i=0; i<n; i++)
       add += a[i];
    return add;
}

    int main(void)
{
    int n, i;
    float *a;
    printf("Escriba el tamaño del arreglo: \n");
    scanf("
    a=calloc(n, sizeof(int));
    for(i=0;i<n; i++)
        a[i] = 0.5*i;
    printf("la suma de lo a[i] es
    free(a);
}
```

En el libro Numerical Recipes in "C", hay un archivo *nrutil.c* que tiene ya las instrucciones para asignar dinámicamente espacio para arreglos, y también para liberarlo, así como para iniciar el conteo de arreglos desde cualquier índice que deseen (también puede usarse la función calloc() ó malloc()). Pueden utilizar *nrutil.c* si lo desean. Para ello deben incluir ese archivo en el preámbulo de main() , pero *nrutil.c* debe entonces estar en el mismo directorio del programa que uno escribe.

```
#include <stdio.h>
#include "nrutil.c"
```

```
int main(void)
{
    float *b, **a;
    int n, m;
    n= 20;
    m= 10;
    b=vector(1,n);
    a=matrix(1,n,1,m);
    ..........
    free_vector(b,1,n);
    free_matrix(a,1,n,1,m);
}
```

El vector b y la matriz a, deben declararse como un apuntador (*b) y un apuntador a un apuntador (**a). La instrucción "b=vector(1,n)" así como "a=matrix(1,n,1,m)" están ya en *nrutil.c*, y dicen que el vector b es del tipo float y sus indices corren de 1 a n. La matriz a es del tipo float y es de tamaño $n \times m$. Al final se libera la memoria con la instrucción "free_vector(b,1,n)" y "free_matrix(a,1,n,1,m)" Si se requiere doble precisión, debe escribirse "b= dvector(1,n)" y liberarse con "free_dvector(1,n)", similarmente para matrices.

9. Guardando Datos en un Archivo

Para guardar los datos que un programa genera en un archivo de datos, uno debe usar un apuntador a un archivo, así se da una dirección para el archivo. Después la función fopen() y fclose() cuya función prototipo está en stdio.h , se usan para abrir y cerrar el archivo cuando uno lo quiera. Considere el ejemplo

```
/** Este programa guarda datos para graficar la funcion **/
/** que usted defina, en este caso la función es seno( x) exp(-x) **/

#include <stdio.h>
#include <math.h>
#define XMIN 0.0 /** se grafica de XMIN a XMAX **/
#define XMAX 5.0
#define DELTA 0.1 /** se escribe un dato en pasos DELTA en x **/

int main(void)
{
    float x
    float funcion( float );
    FILE *fp;
    fp= fopen("file.dat","w");
    for(x=XMIN; x<= XMAX; x += DELTA)
        fprintf(fp,"\ %e\t\ %e\n", x,funcion(x));
    fclose(fp);
    return 0;
}

float funcion( float x)
{
    return (sin(x)*exp(-x));
}
```

La función fopen("file.dat","w"); toma dos cadenas de argumentos. La primera le da nombre al archivo *fp*, en este caso se le da el nombre file.dat,

y la segunda indica el modo en que se usará dicho archivo. Los modos más comunes son *r* para leer (en cuyo caso, el archivo file.dat debe ya existir), y *w* para escribir, aunque hay mas modos y combinaciones.

El loop se usa para avanzar el valor de la x de XMIN a XMAX con un paso DELTA, y fprintf() es una función que almacena x y f(x) con un formato de notación científica %e. La función fprintf() necesita saber en que archivo (su dirección) almacenará datos, por eso se escribe *fp* como argumento de fprintf(fp,"..."); . Luego hay dos cadenas, una donde van los formatos, y otra que dice *qué se escribe* ahí. La expresión *backslash*t da espacio entre las cantidades x y f(x) que se registran en el archivo que llamamos file.dat . No se requiere la terminación ".dat" algunos acostumbran esa terminaciń como referencia para saber que es un archivo de datos.

En el ejemplo usamos unas funciones matematicas, por eso se incluyo *math.h* .

10. Funciones Matemáticas

C tiene funciones matemáticas cuyos prototipos están en math.h Algunas de estas funciones son

sqrt() pow() exp() log() log10() fabs() sin() cos() tan() asin() acos() atan() sinh() cosh() tanh()

La función abs() está diseñada para valores enteros, no para números reales, tenga esto en cuenta, y no confunda *fabs()* con *abs()*

Exceptuando pow(), todas las funciones arriba enlistadas, toman un solo argumento de tipo double y regresan uno de tipo double. Sólo pow() toma dos argumentos de tipo double, regresa uno double, así pow(2.0, 2.5) eleva el 2.0 a la potencia 2.5

10.1. Compilando cuando se usan funciones matemáticas

Cuando se usan funciones matemáticas, se requiere incluir el archivo *math.h*, en cuyo caso, al compilar, es menester escribir al final -lm , es decir

```
cc -o file file.c -lm
```

con esto se compila el programa file.c que tiene funciones matematicas, creándose el ejecutable *file*

11. Uso de Assertions

Suponga que la función f(a,b) se diseña para generar valores positivos, y queremos asegurarnos que en efecto son positivos, entonces usamos:

```
c=f(a,b);
assert(c>0);
```

Si $c > 0$ no se cumple, assert() escribe un mensaje y aborta. assert() tiene sus prototipos en assert.h así que debe incluirse este archivo en el preámbulo.

Si supone que el valor de a debe ser 1 ó -1, y que b debe estar en el intervalo [5,10] escribimos:

```
assert(a ==1 || a == -1);
assert(b >= 5 && b <= 10);
```

Si $|a| \neq 1$, assert() escribe un mensaje y aborta. Después checa si $b\epsilon[5, 10]$ si no es el caso, aborta. assert() es fácil de usar y agrega robustez al código.

12. Algunos Ejercicios

I. Escriba un programa que pida un número entero y escriba todos los números primos menores que él. (Sugerencia: use el operador %)

II. Escriba un código que calcule e imprima los números de Fibonacci: $f_0 = 0, f_1 = 1, f_{i+1} = f_i + f_{i-1}$, y juntamente con cada número su respectivo cociente de Fibonacci: $q_i = f_i / f_{i-1}$ con i=2,3,4,...,N haga N=45 por ejemplo. A qué valor converge q_j?

III. Sea a un número real positivo, considere la sucesión:

$x_0 = 1, x_{i+1} = \frac{x_i + a/x_i}{2}$

con $i = 0, 1, 2, ...$ Puede mostrarse que $x_i \to sqrt(a)$ cuando $i \to \infty$. Escriba un programa que lea valores de a, y use el algoritmo para calcular la raíz de a. Detenga la iteración cuando $|a - x_i * x_i| < EPS$, ud. dél valor de la precisión EPS.