Tecnicatura Superior en Análisis de Sistemas
Programación I

"Cuestiones sintácticas básicas del Lenguaje C"
Versión 1.2 Abril 2013

Lic. Guillermo R. Cherencio

INDICE

Caracteres Válidos dentro de un programa C (además de los caracteres de control):

se pueden usar las siguientes letras:

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m
n o p q r s t u v w x y z
```

se pueden usar los siguientes dígitos:

```
0 1 2 3 4 5 6 7 8 9
```

se pueden usar los siguientes símbolos:

```
! " # % & ' ( ) * + , - . / :
; < = > ? [ \ ] ^ _ { | } ~
```

El lenguaje C es sensible a mayúsculas y minúsculas:

```
pepe != Pepe != PEPE != PEpe != PePe != PEPe != pepE ....
```

Palabras Reservadas: son palabras de uso interno del lenguaje, que el compilador reconoce y por lo tanto no podemos usarlas como identificadores de variables o constantes propias.

```
auto break case char const continue default do double else
enum extern float for goto if inline int long register
restrict return short signed sizeof static struct switch
typedef union unsigned void volatile while _Bool _Complex
_Imaginary
```

Comentarios:

```
/*
comentario multilínea
...
*/

// comentario de una sola línea
```

Sentencias (statements, proposition, proposiciones) son instrucciones *atómicas* (se ejecutan en su totalidad o no se ejecutan) que se ejecutan una detrás de la otra, de arriba hacia abajo, de izquierda a derecha y se separan por ";" (se pueden escribir más de una en un mismo "renglón"):

```
int codigo;                                 // sentencia 1
codigo = 1234;                              // sentencia 2
printf("Codigo vale %d\n",codigo);          // sentencia 3
```

Bloque de Código: Toda sentencia pertenece a un bloque de código. Un bloque de código agrupa un conjunto de sentencias y también puede contener a otros bloques de código. Un bloque de código tiene la siguiente forma:

```
{ sentencia1; sentencia2; .... sentenciaN; }
```

un bloque de código empieza con el signo "{ " y termina con el signo "}":

```
...
int main(void) {  // inicio bloque de código

     int codigo;
     codigo = 1234;
     printf("Codigo vale %d\n",codigo);

}                 // fin bloque de código
...
```

Otro ejemplo de bloque de código anidado:

```
...
int main(void) {  // inicio bloque de código

     int codigo;
     codigo = 1234;
     if ( codigo > 0 ) { // inicio bloque de código

          printf("Codigo vale %d\n",codigo);

     }                   // fin bloque de código

}                 // fin bloque de código
```

```
    ...
```

Función main():

Función `main()` no requiere indicar prototipo, puede ser implementada como:

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

Cualquier otra función que no sea main(), se debe indicar su prototipo y luego su implementación (a menos que dicha implementación se encuentre compilada dentro de una librería) para poder usarla:

```
void f_no_recibe_no_devuelve();        // prototipo

....

void f_no_recibe_no_devuelve() {       // implementación
    ....
}
```

Otro ejemplo:

```
int f_recibe_devuelve(int,char *);        // prototipo

....

int f_recibe_devuelve(int valor,char *texto) {
    // implementación
    ....
}
```

Variables y Constantes:

Identificadores (nombres) de Variables y Constantes:

- Letras y dígitos
- El 1º caracter debe ser una letra
- No puede ser una palabra reservada
- No usar "_" (subrayado) como 1º caracter
- Los primeros 31 caracteres son significativos (variable interna)
- Los primeros 6 caracteres son significativos (variable externa)
- Usar minúsculas para variables y mayúsculas para constantes

Ejemplos:

```
saldo                   // correcto
saldo1                  // correcto
1saldo                  // incorrecto!!
_saldo                  // incorrecto!! (aunque compila)
```

Tipos de datos básicos:

```
char
int
float
double
void
```

El tipo de dato "void" es un caso muy especial: indica ausencia de valor, un valor inexistente, se utiliza cuando no se es requerido un valor. Cualquier expresión se puede convertir en forma explícita (cast, casting) a "void". También es muy utilizado para declarar punteros genéricos "void *", puesto que permite que todo puntero sea convertido a algo de tipo "void *" sin pérdida de información. Favor de ver la sección "Dirección de Memoria" de una variable.

Supongamos que deseamos implementar una función que devuelve un valor entero, hace determinado trabajo y para ello necesita recibir como argumento distintas clases de punteros (apuntadores a caracteres, a números, etc.) la mejor forma de declararla sería:

```
...
int funcion_generica(void *);
...
```

Una vez que la función fue declarada de esta forma, podría ser utilizada por apuntadores de distintos tipos de datos sin necesidad de crear una función específica para cada tipo de dato, veamos este programa de ejemplo:

```
#include <stdio.h>

int funcion_generica(void *);

main() {
      int paso_entero = 1234;
      char *mensaje = "mensaje a enviar";
      long valor_long = 121212L;
      long *puntero_long;
      int retorno;
      puntero_long = &valor_long;
      retorno = funcion_generica(&paso_entero);
      retorno = funcion_generica(mensaje);
      /*
       las siguientes lineas deberian mostrar en pantalla la misma
       direccion de memoria, puesto que puntero_long y &valor_long
       en este caso, se refieren a la misma direccion de memoria!
      */
      retorno = funcion_generica(puntero_long);
      retorno = funcion_generica(&valor_long);
}

int funcion_generica(void *puntero_generico) {
   printf("direccion de memoria: %ld\n",puntero_generico);
}
```

Una forma de declarar una función que no recibe ningún argumento y devuelve un valor entero podría ser (también puede observarse este tipo de implementación en la función main()):

```
...
int funcion_sin_argumento(void);
...
```

Otra forma, también válida:

```
...
int funcion_sin_argumento();
...
```

Cualificadores aplicados a los tipos de datos básicos:

```
short
long
unsigned
signed
static
register
extern
const
```

Ejemplos:

```
int a;
unsigned int b;
short c;
short int d;
unsigned int e;
unsigned long f;
long int g;
unsigned long int h;
unsigned char cc;
```

En cuanto a los cualificadores "`static`" y "`extern`", favor de ver la sección clasificación de variables.

El cualificador "`register`" se aplica sólo a variables enteras. Es ideal para declarar contadores o variables de uso intensivo. Estas variables se almacenan en la memoria de registro del microprocesador (memoria limitada y de alta velocidad) otorgando mayor performance en la ejecución de los programas. Existe un límite (depende de la plataforma) en cuanto a la cantidad de variables de tipo "`register`" que se pueden declarar (por supuesto, esta memoria es escasa). No todos los compiladores implementan esta facilidad del lenguaje, no obstante, no provoca un error de compilación en caso de no estar implementada (simplemente será ignorado el cualificador "`register`"):

```
...
register int i,total=0;
for(i=0;i<100000;i++) {
     total+=ventas[i];
```

```
    }
    printf("Total de ventas: %d\n",ventas);
    ...
```

El cualificador "`signed`" indica que el tipo de dato guarde espacio para el bit de signo, por defecto, esto es así aunque no se indique este cualificador. Lo contrario sucede con "`unsigned`" en donde es obligatorio especificarlo cuando deseamos que el dato se opere como un número positivo, utilizando el bit de signo para representar números más grandes en vez de usarlo como indicador de signo.

El cualificador "`const`" es utilizado para indicar constantes cuyos valores no cambian luego de su asignación, favor de ver la sección constantes.

Tamaños de los tipos de datos:

- Depende de la plataforma (máquina + S.O.)
- Se puede usar la función `sizeof()` para determinar su tamaño en bytes. Ejemplo: `printf("Un int ocupa %d bytes\n",sizeof(int));`
- Los números máximos y mínimos que se pueden representar y otros datos vinculados con los tipos de datos, se pueden observar en `limits.h` y `float.h;` allí también se definen una serie de macros (CHAR_MIN, CHAR_MAX, INT_MIN, UINT_MAX, etc) que sirven para referirse a los valores mínimos y máximos para cada tipo de dato, veamos el siguiente código para que Ud. pueda comprobar el rango de valores de cada tipo de dato en su computador:

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

int main(int argc, char **argv)
{
       printf("tipo de dato\trango\t\t\t\tamaño\n");
       printf("char\t\tmin=%d max=%d\t\t%ld bytes\n",CHAR_MIN,CHAR_MAX,sizeof(char));
       printf("unsigned char\tmin=0 max=%d\t\t\t%ld bytes\n",UCHAR_MAX,sizeof(unsigned
char));
       printf("short\t\tmin=%d max=%d\t\t%ld bytes\n",SHRT_MIN,SHRT_MAX,sizeof(short));
       printf("unsigned short\tmin=0 max=%d\t\t\t%ld bytes\n",USHRT_MAX,sizeof(unsigned
short));
       printf("int\t\tmin=%d max=%d\t%ld bytes\n",INT_MIN,INT_MAX,sizeof(int));
       printf("unsigned int\tmin=0 max=%ui\t\t%ld bytes\n",UINT_MAX,sizeof(unsigned
int));
       printf("long\t\tmin=%ld max=%ld\t%ld bytes\n",LONG_MIN,LONG_MAX,sizeof(long));
       printf("unsigned long\tmin=0 max=%ld\t\t\t%ld bytes\n",ULONG_MAX,sizeof(unsigned
long));
       printf("float\t\tmin=%lf max=%lf\t %ld bytes\n",FLT_MIN,FLT_MAX,sizeof(float));
       printf("float\t\tmin=%e max=%e\t %ld bytes\n",FLT_MIN,FLT_MAX,sizeof(float));
       printf("double\t\tmin=%lf max=%lf\t %ld bytes\n",DBL_MIN,DBL_MAX,sizeof(double));
```

```
        printf("double\t\tmin=%e max=%e\t %ld bytes\n",DBL_MIN,DBL_MAX,sizeof(double));
        return 0;
}
```

La salida de este programa en linux (debian 6 amd64, cpu Intel i3) es la siguiente:

```
tipo de dato        rango                                   tamaño
char                min=-128 max=127                        1 bytes
unsigned char       min=0 max=255                           1 bytes
short               min=-32768 max=32767                    2 bytes
unsigned short      min=0 max=65535                         2 bytes
int                 min=-2147483648 max=2147483647          4 bytes
unsigned int        min=0 max=4294967295i                   4 bytes
long                min=-9223372036854775808 max=9223372036854775807         8 bytes
unsigned long       min=0 max=-1                            8 bytes
float               min=0.000000 max=340282346638528859811704183484516925440.000000     4
bytes
float               min=1.175494e-38 max=3.402823e+38       4 bytes
double              min=0.000000
max=179769313486231570814527423731704356798070567525844996598917476803157260780028538760589558632766878171540458953514382464234321326889464182768467546703537516986049910576551282076245490090389328944075868508455133942304583236903222948165808559332123348274797826204144723168738177180919299881250404026184124858368.000000     8 bytes
double              min=2.225074e-308 max=1.797693e+308     8 bytes
```

En el caso de `float` y `double`, sus valores mínimos no pueden ser impresos utilizando la máscara printf `%lf`, por ello, se agregó al programa la posibilidad de expresarlo en formato exponencial, utilizando la máscara %e . Ud debiera preguntarse: ¿Cómo es posible representar con sólo 4 bytes un número `float` tan grande? Son 32 bits, en donde, de izquierda a derecha, el primer bit se utiliza para el signo, los siguientes 8 bits representan el exponente de *e* y los siguientes 23 bits se utilizan para representar el número después del punto decimal[1].

Ordenamiento general, por tamaño, de menor a mayor:

`char  short  int  long  float  double`

Declaración de una variable: el lenguaje C es un lenguaje *fuertemente tipificado*, es decir, primero se declaran las variables y luego se usan. La declaración implica reservar espacio para la variable y asignarle un valor inicial *por defecto* (default value). El tipo de dato determina la cantidad de bytes en memoria que se necesitan para la variable y el valor por defecto de la misma. La forma de una declaración es: [<cualificador>] <tipo de dato> <identificador>;

Ejemplos (<tipo de dato> <identificador>):

[1] Para mayor información puede consultar http://en.wikipedia.org/wiki/Single-precision_floating-point_format

```
int codigo;
char letra;
float saldo;
int a,b,c,d;   // declaro las variables a, b, c, d
               // como int
```

Ejemplos (<cualificador> <tipo de dato> <identificador>):

```
unsigned int codigo;
unsigned char letra;
```

Dirección de memoria de una variable: el lenguaje C permite el acceso y manipulación de las direcciones de memoria de las variables. Una vez declarada una variable, se asignó memoría a misma y por lo tanto, se encuentra en una dirección de memoria determinada. Es posible obtener su dirección a través del signo "&". *Un puntero o apuntador es una variable que guarda en su interior la dirección de memoria de otra variable*. Esto es un concepto muy simple y potente del lenguaje C. Los punteros se declaran usando el signo "*".

Ejemplo de declaración de puntero de tipo int:

```
int *puntero;
```

Ejemplo de asignación de puntero con la dirección de otra variable:

```
int codigo = 25;
int *puntero;
puntero = &codigo;  // puntero ahora "apunta a codigo"
```

Continuando con el ejemplo anterior, se puede cambiar el valor de la variable codigo a través de su puntero:

```
printf("codigo vale %d\n",codigo);  // codigo vale 25
*puntero = 999;
printf("codigo vale %d\n,codigo);   // codigo vale 999
```

Clasificación de Variables

Variables Internas o Automáticas: son aquellas variables que han sido declaradas dentro de una función. A estas variables se les asigna y desasigna memoria automáticamente cada vez que se ejecuta la función que las declara. Si no han sido asignadas explícitamente, su cotenido es basura (indefinido).

Ejemplo:

```
...
void mi_funcion() {
     int vauto;   // variable automatica
                  // con valor indefinido
     ...
}
...
```

Otro ejemplo:

```
...
void mi_funcion() {
     int vauto = 2;  // variable automatica
                     // con valor explícito
     ...
}
...
```

Variables Externas o No Automáticas: son aquellas variables que han sido declaradas fuera de una función. A estas variables se les asigna memoria una sola vez, al comienzo del programa. Si no han sido asignadas explícitamente, su cotenido es cero. Se inicializan con una expresión constante:

```
#include <stdio.h>
int externa = 5;
int otra_externa;     // valor 0 por defecto
void otra_funcion();  // prototipo

int  main(void) {
     ...
     printf("externa vale %d\n",externa);
```

```
          printf("otra externa vale %d\n",otra_externa);
          ...
     }

     void otra_funcion() {
          printf("externa vale %d\n",externa);
          printf("otra externa vale %d\n",otra_externa);
     }
     ...
```

Cuando una variable externa va a ser utilizada en otro programa fuente (distinto al cual la declaró), se debe informar acerca de su uso utilizando el cualificador "extern"; ello sólo implica "informar acerca de su uso" y no una *re*-declaración de dichas variables o *re*-asignación de memoria (que ya fue hecho en otro programa).

Continuando con el ejemplo anterior, supongamos otro programa fuente distinto al anterior, en donde queremos hacer uso de las variables "`externa`" y "`otra_externa`", debemos indicarlo de la siguiente forma:

```
...
extern int externa;      // no implica una declaración o asignación de
extern int otra_externa; // memoria para ambas variables externas
...
void una_funcion() {
      printf("externa vale %d\n",externa);
      printf("otra externa vale %d\n",otra_externa);
}
...
```

Variables Estáticas: son aquellas variables en donde se antepone el cualificador "`static`" en su declaración. Pueden ser tanto externas como internas.

Variables Estáticas Externas: son variables de tipo "`static`" que han sido declaradas fuera de una función. Son externas y visibles dentro del programa fuente en donde han sido declaradas; no pueden usarse fuera del programa que las declara y utiliza.

Ejemplo de programa fuente "modulo.c" que implementa dos funciones que comparten el uso de dos variables estáticas ("modulo.h" contiene los prototipos de las funciones poner() y sacar()):

```
/*
Modulo que implementa las funciones poner() y sacar()
junto con el buffer estatico compartido por ambas funciones
y no accesible por los usuarios de poner() y sacar() fuera de este
archivo fuente
*/
#define BUFFER_SIZE 255

// variables estaticas externas no accesibles fuera de
// este programa fuente

static char buffer[BUFFER_SIZE];
static int posicion = -1;

#include <stdio.h>
#include "modulo.h"

int poner(char *cadena) {
/*
   copia cadena dentro de buffer y devuelve la cantidad de bytes
   que pudo copiar
*/
   int i;
   if ( posicion < 0 ) posicion = -1;
   for(i=0;posicion < BUFFER_SIZE && cadena[i] != '\0';i++) {
      buffer[++posicion] = cadena[i];
   }
   return i;
}

char *sacar(int cantidad) {
/*
copia desde buffer la cantidad de caracteres indicada y lo copia en una
nueva cadena que es el retorno de esta funcion.
*/
   char *retorno = (char *) malloc(cantidad+1);
   if ( retorno != NULL ) {
      int i;
      for(i=0;posicion >= 0 && i < cantidad;i++) {
         retorno[i] = buffer[posicion--];
      }
      retorno[i] = '\0';
   } else {
      printf("Error en asignacion de memoria!\n");
   }
   return retorno;
}
```

Ejemplo de programa fuente scope.c que hace uso de las funciones declaradas e implementadas en modulo.c :

```
/*
Uso funciones definidas en modulo.c y demuestro que no son accesibles
las variables externas estaticas declaradas y usadas dentro de modulo.c
*/

#include <stdio.h>
#include "modulo.h"

/*
extern char buffer[];      variable externas estaticas de modulo.c
extern int posicion;       no accesibles desde aqui
*/

int main(void) {

   /* las funciones poner() y sacar() pueden usarse sin problemas
      desde scope.c , sin embargo, las variables buffer y posicion
      no son "usables" desde scope.c, no son visibles, se dice
      que estan fuera de alcance. */

   printf("Puse %d caracteres\n",poner("hola q tal"));
   printf("Saque1 %s caracteres\n",sacar(4));
   printf("Saque2 %s caracteres\n",sacar(4));
   printf("Saque3 %s caracteres\n",sacar(4));
   printf("Saque4 %s caracteres\n",sacar(4));
   printf("Puse %d caracteres\n",poner("¿como estan?"));
   printf("Saque5 %s caracteres\n",sacar(25));

/*
   printf("buffer %s\n",buffer);          error en linkedicion!
   printf("posicion %d\n",posicion);      error en linkedicion!
*/

   return 0;
}
```

Para compilar y linkeditar esta aplicación formada por los programas fuentes: `modulo.c`, `scope.c`, `modulo.h` ; se puede hacer de la siguiente forma:

```
gcc -o scope.exe scope.c modulo.c
```

Variables Estáticas Internas: son variables de tipo "`static`" que han sido declaradas dentro de una función. Son internas y visibles dentro de la función en donde han sido declaradas, pero se les asigna memoria una única vez (en vez de hacerlo por cada ejecución de la función). Esto permite que "conserven"

su valor entre las distintas ejecuciones que puede tener la función que las declaró.

Continuando con el ejemplo anterior del programa fuente "modulo.c", supongamos que agregamos al mismo la siguiente función:

```
...
void cuento_ejecucion() {
   static int cuento = 0;
   cuento++;
   printf("ejecucion Nro %d\n",cuento);
}
...
```

Podríamos modificar al programa fuente "scope.c" para hacer uso de esta nueva función:

```
...
int main(void) {
   ...
   // uso variable estatica interna
   cuento_ejecucion();   // imprime: ejecucion Nro 1
   cuento_ejecucion();   // imprime: ejecucion Nro 2
   cuento_ejecucion();   // imprime: ejecucion Nro 3
   cuento_ejecucion();   // imprime: ejecucion Nro 4

   return 0;
}
```

Variables No Estáticas: son aquellas variables en donde no se usa el cualificador "`static`" en su declaración; pueden ser tanto internas como externas.

Asignación de una variable: una vez que la variable ha sido declarada, ésta ya tiene un valor por defecto (default value), luego de ello, puede asignársele un valor explícito (constante o resultante de un cálculo o expresión). La asignación puede hacerse junto con la declaración o por separado. Se utiliza el signo "=" para realizar la asignación, las expresiones a ambos lados de dicho signo deben ser compatibles, sino requerirá de una conversión para que la asignación sea satisfactoria. Es una buena práctica inicializar explícitamente las variables antes de usarlas.

Ejemplos de declaración y asignación explícita en un solo paso:

```
const char *mensaje = "mensaje no cambiable";
unsigned int codigo = 1234;
int a=1,b=2,c=3,d=4;
int otro_codigo = retorno_codigo();
```

Ejemplos de declaración y posterior asignación explícita:

```
...
unsigned int codigo;
...
codigo = 1234;
....
codigo = retorno_codigo();
...
```

Valor por defecto de una variable, en el caso de las variables automáticas tienen un valor indefinido (basura). En el caso de las variables no automáticas, su valor por defecto se asigna una sola vez. Como regla general, los valores enteros se inicializan -si corresponde- a cero y los apuntadores a NULL.

Alcance de una variable: se refiere a los lugares en dónde es válido referirse a una variable o constante. ¿Cuándo puedo usar una variable?. La regla general es: *el alcance de una variable esta restringido al código de bloque que la contiene*. Si la variable no se encuentra en ningún código de bloque, entonces su alcance es global (todo el programa o aplicación). Si se intenta el uso de una variable fuera de su alcance el programa no compila:

```
#include <stdio.h>
int global = 4;

void mi_funcion();  // prototipo o declaración de función

int main(void) {    // inicio bloque

     global = 5;    // correcto
     int interna = 2, otra_interna;
     mi_funcion();
     interna2 = 95; // incorrecto!!, variable fuera de
                    // alcance
     if ( interna > 2 ) {  // inicio bloque
```

```
          global = 10;              // correcto
          otra_interna = 100;       // correcto
     } // fin bloque
     if ( interna > 2 ) {    // inicio bloque
          int nueva_interna = 4;
     } // fin bloque
     /*
     linea siguiente incorrecta!!,
     variable nueva_interna fuera de alcance
     */
     printf("Nueva interna vale %d\n",nueva_interna);

}

void mi_funcion() {   // implemento mi_funcion(),
                      // inicio bloque

     interna = 25;    // incorrecto!!, variable fuera de
                      // alcance
     int interna2 = 33;
     global = 44;     // correcto

} // fin bloque
```

Constantes, pueden usar los literales: L, l, u, U, f, F, 0x, 0X, 0. Hacen uso del cualificador "const". Pueden ser valores constantes dentro de cualquier parte del programa, con iguales características que las variables a diferencia que su valor no puede ser cambiado luego de su asignación. También pueden aplicarse a los argumentos que reciben las funciones.

Ejemplo de constante	Significado en C
1234	`int`
1234L	`long`
1234UL	`unsigned long`
1234.0	`double`
1234e-2	`double`
037	octal `int`
0x1F	hexadecimal `int`
0x1FUL	hexadecimal `unsigned long`
037l	octal `long`

Ejemplos:

```
...
// función que no retorna nada y recibe un argumento de tipo char *
// que no será cambiado dentro de la función, su prototipo podría ser:
void funcion(const char *);
...
const unsigned char car = 'A';
const char lf = 0x0A;
const double pi = 3.141592654;
...
```

Las constantes de tipo cadena (string) o conjunto de caracteres se representan entre comillas dobles:

```
...
printf("constante entre comillas");
const char *titulo = "otra constante";
...
```

Secuencias de escape: son constantes de tipo caracter que permiten representar determinados caracteres. Estas constantes serán detectadas por el compilador y reemplazadas por el o los caracteres que representen.

Listado de las secuencias de escape y su significado:

Secuencia de escape	Significado	Secuencia de escape	Significado
\a	caracter de alarma (campana, beep)	\\	barra invertida (\ )
\b	retroceso (backspace)	\?	signo interrogación
\f	nueva pagina	\'	comillas simples
\n	nueva linea	\"	comillas dobles
\r	retorno de carro	\ooo	reemplace ooo por un número octal del caracter.
\t	tabulador horizontal	\xhh	reemplace hh por un número hexadecimal del caracter.
\v	tabulador vertical		

Suelen combinarse con constantes de tipo cadena de caracteres (strings):

```
...
printf("constante entre comillas\n");
const char *titulo = "otra\tconstante";
...
```

Operadores: son signos utilizados por el lenguaje para indicar cálculos que involucran a uno o dos operandos.

Operadores Aritméticos: permiten realizar las operaciones aritméticas elementales (+,-,*,/) y el resto de la división (%).

Operadores Aritméticos Binarios: operaciones aritméticas que involucran a dos operandos:

```
...
int a=0,b=0;
b = a + 2; // suma
b = 4 * 2; // multiplicación
b = a / 2; // division
b = 4 - 3; // resta
b = 4 % 2; // resto de la division 4 / 2
b = b + a - 1 * 2 / a;
...
```

Operadores Aritméticos Unarios: operaciones artiméticas que involucran a un solo operando.

Operador de Incremento y Decremento: permite incrementar (++) o decrementar (--) el valor de una variable numérica en una unidad. Pueden ser prefijos (primero se incrementa/decrementa y luego se resuelve la expresión) o posfijos (se resuelve la expresión y luego se incrementa/decrementa):

```
...
int a=0;
a++;        // (posfijo) a = a + 1, por lo tanto, a = 1
++a;        // (prefijo) a = a + 1, por lo tanto, a = 2

// (posfijo) imprime "a vale 2\n" y luego incrementa a, a =3
printf("a vale %d\n",a++);

// (prefijo) primero incrementa a y lueog imprime "a vale 4\n"
printf("a vale %d\n",++a);
...
```

Operadores Relacionales: operaciones cuyo resultado es un valor de verdad. A diferencia de otros lenguajes, C no posee un tipo de dato *bool* o *boolean* (para indicar un valor de verdad o falsedad), por lo tanto, toda expresión

igual a *0 (cero)* será considerada falsa y por lo tanto, toda expresión distinto de *0 (cero)* será considerada verdadera.

Expresión	Significado	Expresión	Significado
a > b	¿a mayor que b?	a == b	¿a igual que b?
a >= b	¿a mayor o igual que b?	a != b	¿a es distinto que b?
a < b	¿a menor que b?	!a	¿no es a? negación

Operadores Relacionales de Corto Circuito: operaciones lógicas *"y"* (conjunción, &&), *"o"* (disyunción, ||). Se evalúan de izquierda a derecha y en el caso de *"y"* cuando una expresión da como resultado falso no continúa evaluando el resto (corta el circuito).

Ejemplo 1, supongamos la expresión: `a > b && a > c` si a no es mayor que b y teniendo cuenta que es una operación lógica *"y"* (&&), esto implica que la segunda expresión (a > c) ni siquiera se evalúa, puesto que la primera es falsa, ¿para qué continuar? ello le permite ganar tiempo, pero también hay que estar atentos:

Ejemplo 2, supongamos la expresión: `a > b && a > get_numero(25)` si a no es mayor que b, implica que la función `get_numero(25)` nunca se ejecutó.

Algo similar sucede con un operación lógica *"o"* (||), con que sólo una expresión lógica sea verdad es suficiente y el resto no se evalúa.

Ejemplo 3, supongamos la expresión: `a > b || a > c` si a es mayor que b, entonces la expresión lógica `a > c` nunca se ejecutará.

Operadores para el manejo de bits: sólo aplicables a `char`, `short`, `int` y `long` (con o sin signo). Realiza operaciones lógicas con los bits:

Operador	Significado	Operador	Significado
&	"y" de bits	<<	desplazamiento a la izquierda
|	"o" de bits	>>	desplazamiento a la derecha
^	"o exclusivo" de bits	~	complemento a uno (op. unario)

El comportamiento de << y >> depende de la plataforma, cuando se hace un corrimiento se podrá rellenar con 1 o con 0, dependiendo de la plataforma y el signo: `c = b >> 2;  d = c >> 3;`

El complemento a 1 transforma 1 en 0 y 0 en 1.

Operadores de asignación: se utiliza el signo "=" (no confundir con "==" que sirve para comparar) que puede ser combinado con los operadores binarios +, -, *, /, %, <<, >>, &, ^, | de la siguiente forma:

<expresion1> <operador> = <expresion2>

que es equivalente a:

<expresion1> = <expresion1> <operador> <expresion2>

ejemplo: `x * = y;` que significa: `x = x * y;`

Precedencia y orden de evaluación: dada la expresión `b = b + a - 1 * 2 / a;` ¿Cómo se evalúa? ¿Qué calculo se hace primero y cuál despues? para ello se utiliza una tabla de precedencia que indica el orden en que los operadores se evalúan. Hay varios operadores con igual precedencia, en estos casos, se resuelve según su asociatividad que puede de izquierda a derecha o derecha a izquierda. A continuación se muestra la tabla de precedencia de mayor precedencia (arriba, lo que se evalúa primero) hasta la menor precedencia (abajo, lo que se evalúa último):

Operadores	**Asociatividad**
`( ) [ ] ->`	izquierda a derecha
`! ~ ++ -- + - * & (tipo) sizeof`	derecha a izquierda
`* / %`	izquierda a derecha
`+ - (binarios)`	izquierda a derecha
`<< >>`	izquierda a derecha
`< <= > >=`	izquierda a derecha
`== !=`	izquierda a derecha
`&`	izquierda a derecha
`^`	izquierda a derecha
`|`	izquierda a derecha
`&&`	izquierda a derecha
`||`	izquierda a derecha
`?:`	derecha a izquierda

= += -= *= /= %= &= ^= |= <<= >>=	derecha a izquierda
,	izquierda a derecha

Conversiones entre tipos de datos

Cuando un operador tiene operandos de tipos de datos diferentes, éstos deben convertirse a un mismo tipo de dato, que será el tipo de dato resultante de la expresión. Ejemplo:

```
...
int a=4;
double b=3.14;
int c = a + b;   // operando + con tipos de datos diferentes
...
```

Se realizan conversiones automáticas cuando "no hay pérdida de información", es decir, cuando se pasa de un tipo de dato más pequeño a uno más grande (ejemplo: de `int` a `long`, de `char` a `int`, ...). En estos casos se produce una "promoción" (promote) hacia el tipo de dato más grande (ejemplo: int + double se transforma en una expresión de tipo double). Muchas veces el compilador generará un mensaje de advertencia (*warning*) cuando se pretenda pasar de un tipo de dato más grande a uno más pequeño, sin embargo, el código compilará.

Las conversiones no automáticas o explícitas se pueden hacer de la siguiente forma:

```
(<tipo de dato>)  <expresión>
```

Ejemplo:

```
...
int a = 4,r;
double b = 3.14;
r = (int) a + b;
...
```

esto hay que hacerlo con cuidado, puesto que puede implicar pérdida de información, lo cual puede terminar en resultados no esperados. A esta acción se la suele llamar "*cast* o *casting*". También existen funciones dentro de la biblioteca `<stdlib.h>` que permiten hacer conversiones tales como: `atof()`, `atoi()`, `atol()`, etc.

Formato general de un programa C:

[<prototipo de funciones>]
[<sentencias de preprocesador (#...)>]
[<declaración de variables externas>]

<implementación de función main()>

<implementación de otras funciones>

Ejemplo:

```
#include <stdio.h>   // sentencia del preprocesador

int f_recibe_devuelve(int,char *);  // prototipo

int main(void) {    // implementacion de main()
   int f_ret;
   char *mi_texto = "hola que tal";
   f_ret = f_recibe_devuelve(4,mi_texto);
}

// implementacion de funciones

int f_recibe_devuelve(int valor,char *texto) {
   int i,largo,cambios;
   cambios=0;
   largo = strlen(texto);
   for(i=0;i < valor && i < largo;i++) {
      texto[i] = ' ';
      cambios++;
   }
   return cambios;
}
```