Programacion Imperativa

Nicolás Margenat

1Q 2021

Contents

1 C basics

1.1 #include

- #include <*nombreDelArchivo*> . Busca en un Directorio Especial: */usr/include y derivados.*
- #include "*nombeDelArchivo*" . Busca en el directorio actual de trabajo (en realidad, primero busca en el directorio actual y despues en el especial).

1.2 Palabras Reservadas en C

Las palabras reservadas en C son:

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

1.3 Funciones

Pasaje de parametros

En lenguaje C el pasaje de parametros se realiza siempre por valor (es decir, se pasa una copia de valor derecho).

Definicion - Invocacion - Declaracion

Definicion de Funcion:

```
tipoDeRetorno /* Si no ponemos nada el compilador asume int */
nombreFuncion (tipoPar par1, ..., tipoPar parN) /* Lista de parametro formal
{
    /* Proposiciones */
}
```

Invocacion de Funcion:

```
nombreFuncion(par1, ..., parN); /* Lista de parametros actuales */
nombreFuncion();
```

Declaracion - Prototipo:

```
tipoRetorno nombreFuncion (listaDeParametros);
```

1.4 Complemento a la base

El primer digito indica el signo y solo puede ser 0 o 1.
$CB(P,K)$ representacion polinomica:

$$N = (-1) * d_{k-1} * (p^{k-1}) + p^{k-2} * d_{k-2} + ... + p^1 * d_1 + p^0 * d_0$$

Ejemplo:
$CB(2,5) : 11011$

$$-5 = -1 * (1) * 2^4 + 2^3 * 1 + 2^2 * 0 + 2^1 * 1 + 2^0 * 1$$

Una regla practica para complementar en sistema binario: Invertir todos los digitos del numero y luego sumarle 1.

2 Preprocesador

2.1 Tareas del preprocesador

- Eliminacion de comentarios
- Inclusion de archivos
- Sustitucion de macros
- Compilacion condicional

El preprocesador NO analiza la sintaxis de C, solo sustituye.
!!! Las directivas pueden aparecer en *cualquier lugar de un archivo* y **tienen vigencia a partir de dicha posicion y hasta el final de la unidad de traduccion**.

2.2 Macros

Macro de sustitucion simple/Defincion de Constantes simbolicas

Sintaxis:

```
#define IDENTIFICADOR texto de Reemplazo
```

Constantes simbolicas predefinidas

```
__LINE__     Constante decimal con el nro de la linea actual
__FILE__     String que contiene el nombre del archivo que se esta compilando
__DATE__     String con la fecha de compilacion
__TIME__     String con la hora de compilacion
```

Macros con Parametros

Sintaxis:

```
#define IDENTIF(Arg1,...,ArgN) textoReemplazo
```

Compilacion condicional

Sintaxis:

```
#ifdef IDENTIFICADOR
    ...
#else       //Si se quiere usar elseif poner #else #if
    ...
#endif
```

Argumentos variables

Sintaxis:

```
#define name(fixed_args, ...)
```

Ejemplo:

```
#define imprimir(fmt, ...) printf(fmt, __VA_ARGS__)
```

2.3 Operadores con Macros

- # Unario. Convierte en cadena de texto cada aparicion de un parametro de la macro en el texto de reemplazo.
- ## Binario. Concatena los componentes lexicos a los cuales se aplica.

3 Tipos de Datos

	Tipo de dato	Espacio en Memoria		Tipo Arquitectura	Rango	
Enteros	int	short int int long int	2 bytes 4 bytes 8 bytes	Arquitectura 4 bytes	signed unsigned	[-2^31,2^31 -1] [0, 2^62]
				Arquitectura 2 bytes	signed unsigned	[-2^15,2^15 -1] [0, 2^30]
	char		1byte	Cualquiera	signed unsigned char	[-128, 127] [0, 255] [??, ??]
Reales	float		4 bytes	Cualquiera		
	double	double long double	8 bytes 16 bytes	Cualquiera		

Default:

- int = signed int
- char = NO TIENE

Abreviaciones del tipo int	
short int	short
long int	long
unsigned short int	unsigned short
unsigned long int	unsigned long

3.1 Punteros

```
tipoDeDato * nombrePuntero; // Puntero que apunta a una zona donde hay un da
void * nombrePuntero2; // Puntero generico
```

Apuntes:

- Un puntero a void NO puede ser desreferenciado.

4 Reglas de Conversion de Operandos

A operador B

- Si alguno es *long double*, se convierte al otro en *long double*.
- Si alguno es *double*, se convierte al otro en *double*.
- Si alguno es *float*, se convierte al otro en *float*.
- Se convierten los operadores tipo *char* o *short* en *int* o *unsigned int* segun sea necesario.
- Si alguno es *unsigned long int*, convertir al otro en *unsigned long int*.
- Si uno es *long int* y el otro es *unsigned int*, se convierten a *long int* o *unsigned long int* según sea necesario.
- Si alguno es *long int*, convertir al otro en *long int*.
- Si alguno es *unsigned int*, convertir al otro en *unsigned int*.

5 Constantes

Las constantes se guardan como *int*(2 o 4 bytes en memoria)

5.1 Tipos Asumidos

Tipos Asumidos		
Tipo de Constante	**Tipo Asumido (Creciente)**	
Constantes Decimales	int long unsigned long	1 2 3
Constantes Octales o Hexadecimales	int unsigned int long unsigned long	1 2 3 4

5.2 Modificacion de Tipos Asumidos

Por cuestiones de estilo, ponemos letras mayusculas para determinar el tipo de dato asumido.

Constantes de Tipo int

```
135   ------> signed int
135L  ------> signed long
135U  ------> unsigned int
135UL ------> unsigned long
```

Constantes en Punto Flotante

```
1.7F  ------> float
2.1   ------> double
16.3L ------> long double
```

5.3 Constantes Caracter

Constantes Caracter	
a	beep
b	backspace
f	Comienzo de una nueva pagina
n	newline
r	Retorno de carro
t	tabulador
0	Caracter nulo (null)
\\	Barra invertida
\'	Comilla simple
\"	Comilla doble
ooo	Caracter cuyo ASCII es el numero octal ooo
xhh	Caracter cuyo ASCII es el numero hexa hh

5.4 Constantes de enumeracion

Lista de valores *enteros constantes*
Sinaxis:

```
enum nombre { listaDeConstantes };
```

Siempre asignar el valor de la primera constante para evitar posibles errores en distintos compiladores.

6 Arreglos

Los arreglos pueden ser de cualquier de tipo de dato y la cantidad de elementos debe ser un entero.
Sintaxis:

```
tipo nombreArreglo[cantidad];
tipo nombreArreglo[cantidad] = {valor0, valor1, ..., valorCantidad-1};
tipo nombreArreglo[] = {valor0, valor1, ..., valorCantidad-1};
```

Notas:

1. Al momento de inicializar el arreglo el numero de elementos que va a tener solo puede ser un numero natural.
2. El primer indice es 0 y el ultimo es cantidad-1.
3. Si no inicializas todas las variables rellena con 0s.
4. Solo puedo conocer la cantidad de elementos de un arreglo si esta en el mismo bloque de codigo, y se puede hacer de la siguiente manera:

```
sizeof(array) / sizeof(array[0])
```

6.1 Arreglos bidimensionales

Sintaxis de Declaracion:

```
tipo matriz[filas][columnas];
tipo matriz[filas][columnas] = {
{valor00,valor01, ... ,valor0N},
{valor10,valor11, ... ,valor1N},
...
{valorN0,valorN1, ... ,valorNN} };
tipo matriz[][columnas] = {
{valor00,valor01, ... ,valor0N},
{valor10,valor11, ... ,valor1N},
...
{valorN0,valorN1, ... ,valorNN} };
```

Sintaxis de Acceso a la componente:

```
matriz[i][j]
```

Sintaxis para la Declaracion de un Parametro de tipo arreglo:
IMPRESCINDIBLE INDICAR LA CANTIDAD DE COLUMNAS:

```
tipo funcion(tipo matriz[][MAXCOL]);
tipo funcion(tipo matriz[MAXFIL][MAXCOL]);
tipo funcion(tipo datos[][M1][M2][M3]);
```

7 Punteros

Es una variable que almacena la direccion de otra variable.'
Sintaxis de declaracion:

```
tipoApuntado *nombrePuntero;
```

Equivalencia entre Arreglos y Punteros

```
 arreglo <==> &arreglo[0]
*arreglo <==>  arreglo[0]
```

7.1 Punteros como paramtros de una funcion

Cuando una funcion tiene que devolver cierta respuesta, le pasamos la direccion de una zona de memoria donde dejar dicha respuesta.
Sintaxis de definicion de parametro formal:

```
tipo nombreFunc (tipo *nombreParam, ... );
```

Sintaxis de invocacion:

```
nombreFunc(&variable);
nombreFunc(variablePunt);
```

7.2 Operadores de Punteros

Operador	Significado	Aridad
&	Se aplica a un l-value y devuelve la direccoin de memoria donde dicho l-value esta almacenado	Unario
*	Se aplica a un tipo puntero y devuelve el r-value al cual apunta. Se dice que "desreferencia" al puntero, obteniendo el valor apuntado por el.	Unario

Nota: Un puntero debe ser inicializado antes de ser desreferenciado.
NULL = constante simbolica (definida en stdio.h) utilizada para indicar que un puntero apunta a "nada".

7.3 Operaciones aritmeticas validas

```
puntero++; ++puntero; puntero--; --puntero;
puntero1 - puntero2 //(si puntero2 - puntero1 < 0)
```

Operaciones de Asignacion
puntero1 += numero
puntero1 -= numero
puntero1 = puntero2

Operaciones de Comparacion
puntero1 = = puntero2
puntero1 != puntero2
puntero1 <= puntero2
puntero1 <puntero2
puntero1 >= puntero2
puntero1 >puntero2

8 Strings

Una constante de tipo string es un arreglo de chars que no tiene nombre, por lo tanto ella misma representa la dirección de memoria de la primera componente.

¿Dónde se pueden utilizar los strings constantes?

1. Inicializando punteros a char:

```
char *texto = "Hola";
```

2. Asignando punteros a char:

```
char *texto;
texto = "Hola";
```

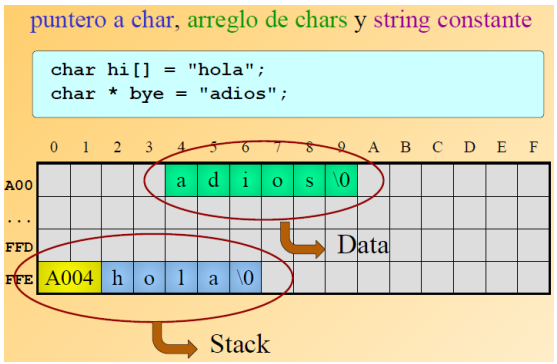
3. Como argumento de la funcion:

```
printf("Hola");
```

4. En referencias de punteros a char:

```
printf("%s", "Hola" + 2); // Imprime "la"
```

8.1 Diferencias entre puntero a char, arreglo de chars y string constante



9 Operadores

9.1 Operadores Aritmeticos

Operador	Significado	Aridad
-	Opuesto	Unario
+	Identico valor	Unario
*	Multiplicación	Binario
/	Division	Binario
+	Adicion	Binario
-	Sustraccion	Binario
%	Modulo - SOLO PARA int - Si uno de los operandos es negativo, el signo del resultado depende de la arquitectura	Binario

9.2 Operadores Relacionales

Operador	Significado	Aridad
<	Menor	Binario
<=	Menor o igual	Binario
>	Mayor	Binario
>=	Mayor o igual	Binario
==	Igual	Binario
!=	Distinto	Binario

9.3 Operadores Logicos

Operador	Significado	Aridad	Lazy?
!	Not	Unario	NO
&&	And	Binario	SI
||	Or	Binario	SI

Las comparaciones, los operadores lógicos (&&, || , !) devuelven 1 si es verdadero, 0 si es falso.

9.4 Operadores de Manipulacion de Bits

SOLO PARA ENTEROS

Operador	Significado	Aridad
$\sim$	Complemento a 1	Unario
<<	Decalaje a la Izq	Binario
>>	Decalaje a la Der	Binario
&	And a nivel bit	Binario
|	Or a nivel bit	Binario
^	Xor a nivel bit	Binario

9.5 Operadores de Incremento y Decremento

La operacion con estos operandos devuelve un valor del mismo tipo del operando.

Operador	Significado	Aridad
++	Pre/post incremento en 1	Unario
–	Pre/post incremento en 1	Unario

```
++variable = Primero modifica la variable y despues la usa
variable++ = Primero usa la variable y despues la modifica
```

9.6 Operador de Asignación

Operador	Significado	Aridad
=	Asignación	Binario

9.7 Operador Condicional

Operador	**Significado**	**Aridad**
oper1? oper2 : oper3	Se evalua oper1, si el mismo es verdadero se evalua solo oper 2, si es falso se evalua solo oper3.	Ternario

Como se lee?:

```
a = (b > c)? b : c
"b es mayor que c? Entonces devolveme b, de lo contrario c".
```

9.8 Operador Coma

Operador	**Significado**	**Aridad**
oper1, oper2	Se evalua de Izq. a Der., descartando el valor de oper1 y devolviendo el valor de oper2	Binario

9.9 Operador Sizeof

Operador	Significado	Aridad
sizeof (operando) **sizeof operando**	Numero (entero sin signo) de bytes requeridos para almacenar un objeto con el tipo de operando	Unario

El operando puede ser un tipo de dato o una expresión. Cuando se trata de una expresión NO la evalua para producir el resultado.

10 Precedencia y Asociatividad de operadores

Tabla de Precedencias		
Importancia (Creciente)	**Operadores**	**Asoc.**
1	( ) [ ] . ->	Izq. a Der.
2	! ~ ++ − + - (tipo) sizeof	Der. a Izq.
3	* / %	Izq. a Der.
4	+ - (ambos binarios)	Izq. a Der.
5	<<>>	Izq. a Der.
6	«= »=	Izq. a Der.
7	== !=	Izq. a Der.
8	&	Izq. a Der.
9	^	Izq. a Der.
10	|	Izq. a Der.
11	&&	Izq. a Der.
12	||	Izq. a Der.
13	?:	Der. a Izq.
14	= += -= *= %= &= ^= |= <<= >>=	Der. a Izq.

La precedencia y asociatividad de los operadores estan especificadas completamente, pero el *orden de evaluacion de las expresiones es indefinida.*
Excepciones: && || ?: ,

11 Entrada y salida de datos

11.1 putchar

Defínción: Escribe el carácter correspondiente al código indicado en la salida estándar.
Sintaxis:

```
putchar(valorASCII)
```

11.2 printf

Defínción: Permite generar salida estándar con formato.
Sintaxis:

```
printf("Cadena de formato", expr1, expr2, ...)
```

Indicadores de conversión		
Símbolo (%)	**Tipo de Dato**	**Conversión**
d, i	int	Número entero con signo en base 10
o	int	Número octal sin signo ni cero inicial
x, X	int	Número hexadecimal sin signo
u	int	Número entero sin signo en base 10
c	int	Carácter simple
s	char*	Cadena de caracteres
f	double	Número decimal en punto fijo
e, E	double	Número decimal en notación exponencial
g, G	double	Con exponente menor a -4 la precisión usa %e, sino usa %f

Modificadores de la conversión	
Simbolo	**Conversión**
-	Alineamiento a la izquierda
0	Rellena con ceros
Un número (Ancho)	Indica el ancho minimo del campo, completado con blancos a la izq o der segun corresponda
*****	El ancho o la precision la toma del argumento
.	Separa la cantidad de digitos de la parte entera de la precicion
+	Imprime con signo, incluso un positivo
Un número (Precisión)	floating point = nro. de digitos decimales strings = nro. max de caracteres a imprimir int = nro. minimo de digitos a imprimir

11.3 scanf

Sintaxis:

```
int scanf(const char * restrict fmt, ...);
```

Indicadores de conversión		
Símbolo (%)	**Tipo de Dato**	**Conversión**
d	int	Número entero con signo en base 10
o	int	Número octal sin signo ni cero inicial
x	int	Número hexadecimal sin signo
u	int	Número entero sin signo en base 10
c	int	Carácter simple
s	char*	Cadena de caracteres
f	double	Número decimal en punto fijo
[...]	cadenas	Letras/Nros/Simbolos (es como las regex)
[^...]	negar cadenas	Letras/Nros/Simbolos (es como las regex)

Opcionales:

1. Caracter de supresion *****
2. Numero para la amplitud de campo
3. h, l o L para amplitu del destino

11.4 Redireccionamiento

Símbolo	Funcionamiento
<	programa <archivoEntrada
>	programa >archivoSalida (pisa lo que habia en archivoSalida)
|	programa | otroPrograma
>>	programa >>archivoSalida (no pisa lo que habia en archivoSalida)

12 Heap

El heap forma parte de la memoria asignada a un proceso (programa en ejecución) por el sistema operativo. El mismo se fracciona en forma dinámica en bloques de distintos tamaños cada vez que un proceso solicita un bloque mediante la invocación a malloc.

13 Structs

Colección de datos heterogéneos y sin orden. Sus elementos se identifican con nombres

Sintaxis de declaracion de un struct:

```
struct nombreRegistro{
    tipo1 nombreCampo1;
    tipo2 nombreCampo2;
    ...
    tipoN nombreCampoN;
};
```

nombreRegistro es un nuevo tipo, NO es una variable. Por lo tanto no reserva espacio hasta que declaramos una variable de dicho tipo.

Sintaxis de declaracion de una variable de tipo struct:

```
/* ALTERNATIVA 1 */
struct nombreRegistro{
    tipo1 nombreCampo1;
    tipo2 nombreCampo2;
    ...
    tipoN nombreCampoN;
} nombreVariable;

/* ALTERNATIVA 2 */
struct nombreRegistro{
    tipo1 nombreCampo1;
    tipo2 nombreCampo2;
    ...
    tipoN nombreCampoN;
};
struct nombreRegistro nombreVariable;

/* ALTERNATIVA 3 */
/* Podemos usar typedef para simplificarnos la vida */
typedef struct nombreOptativo{
    tipo1 nombreCampo1;
    tipo2 nombreCampo2;
    ...
    tipoN nombreCampoN;
} nombreTipo;

struct nombreOptativo unaVariable; // Esta y la de abajo son equivalentes
nombreTipo otraVariable;
```

Inicializacion de una variable de tipo struct:

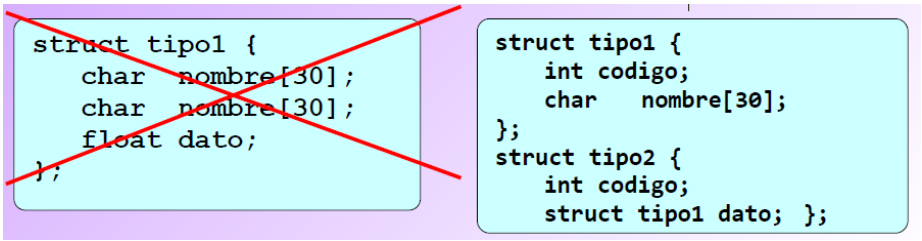
```
typedef struct {
    char aerolinea[30];
    int vuelo;
    char fecha[10];
    char hora[6];
} tipoPasaje;

/* ALTERNATIVA 1 */
tipoPasaje pasaje = {"Flybondi",905,"20240619", "20:30"};

/* ALTERNATIVA 2 */
pasaje = (tipoPasaje) {
    .aerolinea = "Flybondi",
    .fecha = "20240619",
    .vuelo = 905,
    .hora = "20:30"
};
```

Jerarquia de Campos

Al anidar estructuras se establecen jerarquías. Dos campos dentro de la misma jerarquía deben tener diferentes nombres. Dos campos de diferente jerarquía pueden coincidir en sus nombres.



14 Biblioteca Estandar

14.1 stdio.h

Standard Input/Output Library

```
int getchar(void);
int putchar(int c);
int ungetc(int c, FILE *stream);    // Devuelve un caracter al buffer
int printf(const char *fmt, ...);
void clearerr(FILE *stream);
int feof(FILE *stream);             // Devuelve 1 si era EOF
int ferror(FILE *stream);           // Devuelve 1 si era ERROR
void perror(const char *s);         // Var global para comunicar mensajes de
```

Nota: Hacer 2 ungetc seguidos NO es seguro.

14.2 stdlib.h

Standard System Library

```
int abs(int num);        // Devuelve el valor absoluto
long labs(long num);     // Idem pero con un long int
int rand(void);          // Devuelve un numero pseudo-aleatorio entre [0, RAN
int srand(unsigned int seed);   // srand se comunica con la funcion rand y e
typedef unsigned int size_t;
void * malloc(size_t size); // Devuelve el puntero a una zona de memoria que
void * calloc(size_t nobj, size_t size); // Idem malloc pero inicializa en 0
void * realloc(void * p, size_t size);
void free(void * p); // Libera la zona de memoria
```

Apuntes:

1. *rand()* tiene la secuencia armada, entonces siempre te devuelve los mismos valores pq estan ya definidos.
2. *srand()* se comunica con la funcion *rand()* y empieza con la sucesion desde otro punto.
3. *void srand(time(NULL))* para que sea aleatorio *rand()*.
4. Nunca poner *void srand(time(NULL))* adentro de loops, sino se va a generar siempre la misma secuencia (porque toma milisegundos).
5. Por cada malloc() tiene que haber un free().

14.3 ctype.h

Character Type Library
Retornan 0 (verdadero) o un valor distinto de 0 (falso)

```
int islower(int c);
int isupper(int c);
int isalpha(int c);      // Es letra
int isdigit(int c);
int isxdigit(int c);     // Es digito hexa
int isalnum(int c);      // Es letra o digito
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int iscntrl(int c);
int isgraph(int c);
int toupper(int c);
int tolower(int c);
```

Nota: Recordar que UNA FUNCION NO MODIFICA EL VALOR DE LA VARIABLE QUE LE PASAS COMO PARAMETRO

14.4 math.h

Math Library

```
double fabs(double x);        // Valor absoluto
double floor(double x);       // Mayor de los enteros menores a x
double ceil(double x);        // Menor de lo enteros mayores a x
double fmod(double x,double y);
double sqrt(double x);        // Si le pasas un valor invalido se va de rango
double pow(double x,double y);
double exp(double x);         // e^x
double log(double x);
double log10(double x);
double sin(double angulo);    /*
double cos(double angulo);    ** Usando radianes
double tan(double angulo);    */
```

Notas: Esta biblioteca no se linkedita automaticamente, hay que agregar -lm cuando se compila el codigo.

14.5 errno.h

Tiene definidas dos constantes

```
EDOM = error en el dominio de la funcion
ERANGE = error en el rango, osea fuera de rango
```

Notas: Conviene siempre inicializarla en 0 por ser una variable global.

14.6 string.h

String Library

Funcion	Significado
unsigned int strlen(const char *s)	String length
char *strcpy(char *d, const char*f)	Copies the string pointed to, by f to d.
char *strncpy(char *d, const char *f, int n) by f to d.	Copies up to n characters from the string pointed to,
char *strcat(char *d, const char*f)	Appends the string pointed to, by f to the end of the string pointed to by d.
char *strncat(char *d, const char *f, int n)	Appends the string pointed to, by f to the end of the string pointed to , by d up to n characters long.
int strcmp(const char *d, const char *f) pointed to by str2.	Compares the string pointed to, by str1 to the string
int strncmp(const char *d, const char *f, int n)	Compares at most the first n bytes of str1 and str2.
char *strchr(const char *s, char c)	Searches for the first occurrence of the character c (an unsigned char) in the string pointed to, by the argument s.
char *strrchr(const char *s, char c)	Searches for the last occurrence of the character c (an unsigned char) in the string pointed to by the argument s.
char *strpbrk(const char *s, const char *set) any character specified in set.	Finds the first character in the string s that matches
char *strstr(const char *d, const char *f)	Finds the first occurrence of the entire string f (not including the terminating null character) which appears in the string d.

14.7 strings.h

```
int strcasecmp(const char *, const char *);
int strncasecmp(const char *, const char *, size_t);
```

15 Modificadores de Variables

Nota: Las variables globales SIEMPRE se inicializan en 0.

15.1 static

Estas variables se crean afuera del stack.

Definidas fuera de una funcion

Similar a la aplicación sobre funciones, la variable es visible solo dentro del módulo donde está definida e invisible al resto de los módulos.
Ejemplo de Invocacion:

```
static int a;
```

Definidas dentro de una funcion

La variable existe durante toda la ejecución sin perder su valor entre invocación e invocación.
Ejemplo de Invocacion:

```
int
funcion(int valor)
{
    static int acum = 0;
    acum += valor;
    return acum;
}
```

15.2 extern

Se usa para utilizar una variable global. Es para avisarle al compilador que la vas a usar, si esta en el mismo archivo no hace falta.

Definidas/Declaradas fuera de una funcion

Es una variable visible desde todos los archivos que componen el programa.
Ejemplo Invocacion:

```
extern int a;
```

Declarada dentro de funciones

Es una variable definida en otra zona y referenciada desde la función donde se la declara.

15.3 const

Se utiliza para definicr una constante con el espacio de una variable (por ejemplo para definir una constate short).

16 Reglas de estilo

16.1 Nombres de Archivos

1. Los nombres de archivos deben comenzar con una letra, continuando con letras (minúsculas) o números.
2. No utilizar más de 8 caracteres para la parte prefija.
3. Sufijo .c para los archivos con código fuente en lenguaje C.
4. Sufijo .h para los archivos de encabezado.
5. Es conveniente el uso del archivo nombrado README como resumen del contenido que se encuentra en cada directorio.

16.2 Orden de las Secciones en un Archivo Fuente

1. Prólogo que indique una descripción del propósito del contenido. Debe ademas contener autor, version, referencias.
2. Inclusión de encabezamientos.
3. Definición de constantes simbólicas, macros, typedef y enumerativos, que se apliquen a todo el archivo.
4. Funciones que conforman el programa.

16.3 Comentarios

1. Los comentarios deben explicar qué se está haciendo y no cómo se lo está haciendo. Solo resulta útil explicar por qué se implementó de determinada forma un fragmento de código, cuando se analizaron varias alternativas y solo la actual resultó aceptable.
2. Los comentarios deben explicar el significado de los parámetros y las supuestas restricciones de una función.
3. Los comentarios dentro de las funciones deben estar indentados de la misma forma que el resto del código en donde se encuentran, pudiendo escribirlos en una sola línea. Si son muy cortos pueden colocarse al costado del código.
4. Cuando un comentario se refiere a un bloque debe tener alguno de los siguientes formatos:

```
/*
* Comentario
* de Bloque
*/
```

```
/*
** Comentario
** de Bloque
*/
```

16.4 Constantes de Enumeración

1. No conviene tener valores consecutivos con "baches" internos.
2. Si se usa un enumerativo, conviene que la primera constante sea un valor distinto de cero o sea un valor que indique un caso especial de la lista.
3. Las constantes enumerativas deben empezar con mayúscula o bien estar todas en mayúsculas.

16.5 Constantes Simbólicas

1. Los identificadores para las constantes simbólicas deben estar formados por letras mayúsculas.

16.6 Identificadores

1. Los identificadores no deben tener underscore ni al comienzo ni al final.
2. Evitar tener identificadores que sólo difieran en una letra, por ser mayúscula en uno y minúscula en otro.
3. Evitar identificadores que se parezcan mucho entre sí.
4. Evitar usar la l (letra ele) y el 1 (uno).
5. Evitar usar nombres de la biblioteca estándar.

16.7 Proposiciones Simples y Bloques

1. Debe haber sólo una proposición por línea, a menos que las proposiciones estén altamente relacionadas (Ej: break).
2. Aunque el lenguaje permite resumir varias acciones en una sola expresión, lo que se debe priorizar es la claridad y la mantenibilidad del código. Por ejemplo:

   ```
   a = b + c;
   d = a + e;
   ```

   En lugar de:

   ```
   d = (a = b + c) + e;
   ```

3. Las llaves de un bloque deben estar siempre en una línea separada, y cada proposición del mismo debe estar en una línea aparte, e indentada respecto de las llaves.

16.8 Proposiciones de Decisión

1. Para consultar variables "booleanas" no es necesario indicar la igualdad. Por ejemplo:

```
if (esPar != 0)   if(esPar)
```

2. Hay casos en los que se puede prescindir de la proposición if-else para asignarle a una variable un resultado booleano.

3. Para proposiciones excluyentes, usar if excluyentes. Por ejemplo:

```
if (sueldo < 300)
    printf("Sueldo inicial\n");
else if (sueldo < 900)
    printf("Sueldo medio\n");
else
    printf("Sueldo aceptable\n");
```

4. Cuando se opera con números reales hay que utilizar un rango de tolerancia para analizar igualdad o desigualdad.

16.9 Espacios y Tabulaciones

1. Usar blancos verticales y horizontales en forma generosa.

2. La indentación y el espaciado deben reflejar la estructura del bloque del código.

3. Una cadena de caracteres larga conteniendo operadores booleanos debe dividirse en líneas separadas, cortando antes de un operador booleano:

```
if (a == 5 && total < tope && tope <= MAX)
```

   La forma correcta es:

```
if ( a == 5 && total < tope
    && tope <= MAX )
```

16.10 Funciones

1. Cada funcion debe realizar una única tarea: su nombre debe ser significativo y surgir naturalmente. Si es dificil elegirle un nombre es porque no llega a hacer nada o hace demasiadas cosas.

2. Cada prototipo de funcion debe ser precedido por un prologo (comentario) que indique que hace y como usarla.

3. Utilizar programacion defensiva, o sea nunca presuponer que algo jamas va a ocurrir.

4. Someter cada funcion a pruebas de software: caja blanca y caja negra.

16.11 Macros

1. Nombres de macros en mayusculas.
2. Todas las ocurrencias de argumentos entre parentesis.
3. Si hay operaciones matematicas, encerrar el texto de la macro entre parentesis.

16.12 Arreglos

1. Para conocer la cantidad de elementos del arreglo (siempre que el arreglo este dentro del bloque donde se lo define) usar:

```
sizeof(nombreArray) / sizeof(nombreArray[0])
```

17 Algoritmos

Bubble Sort

Comparamos items consecutivos, si estan fuera de lugar los swappeamos. Al final dell arreglo hay una particion de elementos ordenados, ya que el numero mas grande (de cada iteracion) va a "bubblear" hasta el final del arreglo con cada iteracion.
Pseudocodigo:

```
for i from 1 to N
    for j from to N -1
        if a[j] > a[j + 1]
            swap(a[j], a[j + 1])
```

Complejidad: $O(n^2)$

Selection Sort

En cada iteracion seleccionamos el item mas chiquito de la particion no ordenada y lo movemos a la parte ordenada.
Pseudocodigo:

```
for(j = 0; j < n-1; j++)
{
    int iMin = j;
    for(i = j + 1; i < n; i++)
        if(a[i] < a[iMin])
            iMin = i;
    if(iMin != j)
        swap(a[j], a[iMin]);
}
```

Complejidad: $O(n^2)$

Algoritmo de Euclides

Es para encontrar el Maximo Comun Divisor entre dos numeros.
Pseudocodigo:

```
aux = num1
while(aux != 0)
{
    num1 = num2;
    num2 = aux;
    aux = num1 % num2;
}

return num2;
```