El presente trabajo no constituye (bajo ningún punto de vista) un tratado del Lenguaje C, ni pretende sustituir (de ninguna manera) la consulta de una bibliografía más adecuada . En realidad se trata de mi "ayuda memoria" o apunte en base al cual desarrollo los temas que se tratan en la cátedra . El hecho de formalizarlos en un trabajo escrito, es que pueden servir de ayuda o guía a los alumnos de la cátedra .

Algo a tener en cuenta es que fué escrito fuera del período de clases, con lo cual seguramente debe ser incompleto . Lo que con seguridad resultará provechoso para el alumno es la consulta de una bibliografía formal . En realidad estas líneas constituyen lo antes dicho, no bastan en general (yo diría que nunca), los apuntes tomados en clase junto con los apuntes de la cátedra para dominar una materia, como tampoco un libro puede abarcar los contenidos de una materia, ni una materia desarrolla los contenidos de un libro .

Una última observación (en la que desde ya incluyo a estas líneas) acerca de las guías y tutoriales que se pueden obtener en internet es que hay que tomarlas con cuidado, en lo que hace a lo conceptual del lenguaje . Muchas de estas guías contienen errores conceptuales que consideraremos inadmisibles en la evaluación de los alumnos, aunque de todos modos pueden ser valiosas para comenzar a dar los primeros pasos .

Dado que los alumnos que cursan esta materia han cursado en 1er año una materia que consta de la resolución de algoritmos empleando algún método de diagramación, y (por experiencias anteriores) rudimentos de programación en lenguaje C, no trataré al lenguaje como para principiantes, pero haremos una rápida recorrida por todo aquello que ya debían saber, junto con otros temas que rápidamente deben conocer . Bueno, manos a la obra :

La función main, funciones e identificadores :

Un programa en Lenguaje C (que en adelante denominaremos simplemente C) se almacenará en al menos un archivo (por ej.: `prog-1.c`) con un nombre (`prog-1`) y un punto (`.`) que separa la extensión del nombre del archivo (`c`), y estará compuesto por diversas funciones .

Dentro de dicho(s) archivo(s) debe haber una única función identificada por la denominación `main` . De este modo el programa más breve que se puede compilar y ejecutar en C se vería así :

```
main( )
{
}
```

El código de la función `main` será el que comienza a ejecutarse en primer lugar . Cada archivo fuente se compilará por separado, y luego se linkeará todo en conjunto para producir el código ejecutable . Las funciones, variables, tipos de datos, etc. que declaremos, los podremos nombrar según nuestro gusto, pero respetando algunas pocas reglas de sintaxis .

Un identificador debe comenzar con una letra del alfabeto inglés (no vale la letra `ñ` o `Ñ` que nos distingue, ni vocales con acento ni la `ü` con diéresis) mayúscula o minúscula . En C el carácter subguión (_) es una letra más . Hay que recordar que C es case sensitive (sensible a mayúsculas o minúsculas), con lo que los identificadores `MAIN`, `MaIn`, `MaIN`, ... podrían ser identificadores válidos .

A continuación puede continuar con cualquier carácter representativo de una letra (incluso el _), o los dígitos del `0` al `9` . Los siguiente son identificadores válidos .

x	**kxp1**	**_**	**__**	**ciclo**	**func**	**valor**	**algo**

De trodos modos, aunque identificadores como `x`, `y`, `n`, `_` sean identificadores válidos, pueden llegar a ser poco representativos de para qué se emplean, por lo que además respetaremos reglas de estilo : Los identificadores que elijamos serán representativos de su empleo .

Además, un identificador no debe coincidir con alguna de las palabras reservadas del C . La siguiente es una tabla de palabras reservadas típico de un compilador C/C++ .

Palabras Reservadas :

asm	auto	bad_cast	bad_typeid
bool	break	case	catch
char	class	const	const_cast
continue	default	delete	do
double	dynamic_cast	else	enum
except	explicit	extern	false
finally	float	for	friend
goto	if	inline	int
long	mutable	namespace	new
operator	private	protected	public
register	reinterpret_cast	return	short
signed	sizeof	static	static_cast
struct	switch	template	this
throw	true	try	type_info
typedef	typeid	typename	union
unsigned	using	virtual	void
volatile	while		

Observación : las palabras con fondo gris son agregadas por el C++ .

Además de estas, algún compilador puede contemplar otras palabras reservadas, o incluso diferir en algunas (como por ejemplo : `asm`) .

Tipos de datos y Variables :

Los tipos de datos que posee C pertenecen a dos grandes grupos (exclusivamente) :

		calificadores			
		signed	unsigned	short	long
Enteras	char	x	x		
	int	x	x	x	x
Punto flotante	float				
	double				
	long double				

A estos tipos se debe agregar el tipo de dato **`void`** o indeterminado, que se emplea para indicar que una función no devuelve nada, o para declarar punteros a algún tipo de dato .

Como se puede observar, en C no se han previsto tipos de datos alfanuméricos . Estos tipos de contenidos (carácteres o cadenas de carácteres) se pueden almacenar en variables de tipo char . Lo que en realidad se almacenan son los números de ASCII (como números enteros, con los que se puede operar) de los carácteres o cadenas de carácteres .

Ejemplos :

```
char letra;
unsigned char x;
signed char val;
```

```
int numero;
signed int y;          /* signed y;          */
unsigned int x;        /* unsigned x;        */
short int valor;       /* short valor;       */
long int cantidad;     /* long cantidad;     */
signed short int f;    /* signed short f;    */
signed long int pp;    /* signed long pp;    */
unsigned short int q;  /* unsigned short q;  */
unsigned long int ra;  /* unsigned long ra   */
```

```
float sueldo;
```

```
double valor;
```

```
long double importe;
```

Atención : no debe confundirse el tipo de dato `long double` (estándar de facto, brindado por la mayoría de los compiladores) con el calificador `long` aplicado a los `double` . Sólo es el nombre compuesto por dos palabras del tipo de dato .

Algunos compiladores para plataformas de 64 bits brindan adicionalmente el tipo de dato : `long long` que permite generar variables o constantes que se almacenan en 8 Bytes .

Los enteros `char` se almacenan en un solo Byte, por lo que son aptos para almacenar los números de ASCII de carácteres, y en arrays (o vectores de char) podemos almacenar cadenas de carácteres (emulando lo que en otros lenguajes son las variables alfanuméricas) . Para el manejo de estos array de char emulando variables alfanuméricas el C dispone de funciones de biblioteca al efecto . No se debe olvidar que con las variables char siempre se pueden efectuar operaciones aritméticas .

Constantes :

Se entiende por constantes a aquellas partes de un programa que no cambian su valor a durante la ejecución del programa . Las constantes literales son aquellas con las que se declaran dichos valores constantes . Las constantes enteras son por defecto decimales, (p.ej.: `13`) a menos que comiencen con cero (p.ej.:`015`) con lo que resultan estar expresadas en octal, o que comiencen con `0x` ó `0X` (p.ej.: `0xC`) para estar expresadas en hexadecimal . Si una constante decimal está seguida por `u` ó `U` se pone de manifiesto que es `unsigned int` . Si está seguida de `l` o `L`, se pone de manifiesto que es `long` .

Las constantes `char` se indican entre apóstrofes (o comillas simples) (p.ej.: `'d'`) .

Para indicar carácteres especiales se dispone de las siguientes secuencias de escape :

Denominación	secuencia de escape	Nro de ASCII
carácter nulo	\0	0
beep	\a	7
backspace	\b	8
*tab horizontal	\t	9
newline	\n	10
tab vertical	\v	11
formfeed	\f	12
carriage return	\r	13
contrabarra	\\	92
apóstrofe	\'	39
comillas	\"	34
número octal	\0oo	
número hexadecimal	\xhh	

Las constantes en punto flotante se expresan en decimal con signo, opcionalmente seguido por un exponente entero decimal especificado con `e` o `E` . Además puede estar seguido

por `f` , `F`, `l` ó `L` indicando que es `float` o `long double`, si no se especifica es `double` . Ejemplos :

float	double	long double	valor
2f	2.	2.L	2.0×10^{0}
2.03f	2.03	2.03L	2.03×10^{0}
12.5e-2f	12.5e-2	12.5e-2L	1.25×10^{-1}

Las constantes de cadena de cracteres se representan encerradas entre comillas . Internamente se almacena un array de enteros `char` conteniendo los números de ASCII de los carácteres, y a continuación se almacena siempre un carácter nulo (p.ej.: ″Hola″ es un arreglo de 5 bytes, en el último contiene el número 0) .

Macroreemplazos 1 (`#define` sencillos) :

Empleando la directiva para el preprocesador de C, `#define` se pueden declarar lo que algunos autores denominan como **constantes simbólicas** . En realidad el preprocesador reemplazará para el compilador en todos los lugares donde figure el identificador declarado con #define el valor definido .

El código que sigue	es equivalente a este

`#define TAM 30`
`...`
`int vector[TAM};`
`for(x = 0; x < TAM; x++)`
`...`
	
	`...`
	`int vector[30};`
	`for(x = 0; x < 30; x++)`
	`...`

La diferencia entre uno y otro, es que modificando en un solo lugar el macroreemplazo `TAM` en un solo lugar, el procesador se encarga de reemplazarlo para el compilador en todos los lugares que sea necesario . El compilador recibirá el valor `30` reemplazado como se ve en el código de la derecha .

Tipos de datos, struct y typedef :

En C el usuario puede declarar estructuras y tipos de datos que antes no existían en el lenguaje . Si el programador necesita declarar un tipo de dato que contenga, p. ej., un número de DNI, un apellido y nombres y un importe lo puede hacer de dos modos :

```
struct s_pers
{
   long dni;
   char apyn[36];
   float importe;
};
/* con s_pers se pueden
declarar variables
struct a conveniencia
del programador */

struct s_pers afi;
```

```
typedef struct s_pers
{
   long dni;
   char apyn[36];
   float importe;
} t_pers;
/* con s_pers se pueden
declarar variables
struct a conveniencia
del programador .
con t_pers se pueden
declarar variables
idénticas a las
anteriores, sin
necesidad de la palabra
clave struct */
struct s_pers afi;
/* o mejor aún : */
t_pers afi;
/* una u otra, no
ambas, preferntemente
la segunda */
/* en la declaración
typedef se puede omitir
el identificador de
estructura s_pers */
```

La declaración de tipo `typedef` además nos permite declaraciones como :

```
typedef int   BOOL;
```

Con lo que estamos definiendo un nuevo tipo de dato `BOOL` (que en el fondo es un `int` . Si junto con esta declaración declaramos :

```
#define TRUE    1   | o en su | const int TRUE  = 1;
#define FALSE   0   | defecto | const int FALSE = 0;
```

nos permite declarar variables o valores de retorno de funciones con sentido lógico que admitan los valores de `TRUE` o `FALSE` .

Tipos de dato `enum` :

Mediante las declaraciones de tipo `enum` podemos escibir en una forma mucho más compacta el ejemplo anterior, donde quedan mucho más claro y conciso los expresado en los tres renglones antes requeridos :

```
enum BOOL { FALSE, TRUE };

/* funcion que devuelve Verdad o Falso */
enum BOOL func()
{
   if( /* <alguna condición> */ )
      return TRUE;
   return FALSE;
}

void main()
{

   enum BOOL Valor;  /* declara una variable */
   Valor = TRUE;     /* asigna TRUE          */

   if(func() == Valor)
      puts("La función devolvió falso");
/*    ...    */
}
```

Se debe notar que por defecto a la primer etiqueta (`FALSE`) la declaración la inicializa con 0, a la segunda (`TRUE`) con 1, y así sucesivamente hasta la última . En el ejemplo sólo teníamos dos etiquetas, pero si para un programa en particular declararmos la enumeración de provincias, y sólo necesitamos considerar algunas de ellas que se corresponderán con los valores 0, 1, 3, 5, 6, 9, 10 y 11, podríamos tratarla del siguiente modo :

```
enum provin { CAPITAL, BSAIRES, CORDOBA = 3,
              STAFE = 5, LAPAMPA,
              ERIOS = 9, CORRTES, MISIONES };
```

Uniones :

Las uniones brindan un modo de almacenar un tipo de información u otro en el mismo espacio de almacenamiento, dependiendo de las necesidades de un programa en particular .

```
union u_persona
{
   char nombre{36};
   char inicial;
} ;
```

Este es un sencillo ejemplo que nos permitiría comprobar que una vez declarada una variable que responda a esta `union` si se completa el `nombre` con p. ej.: `"Pedro"`, la `inicial` pasa a contener el carácter `'P'` y modificando `nombre` o `inicial` se altera el otro .

Un ejemplo no tan trivial sería aquel en que necesitemos almacenar por ejemplo las dimensiones de figuras geométricas (p.ej.: círculos y rectangulos) .

```
enum t_figura { CIRCULO, RECTANGULO };
/*
 * las etiquetas del t_figura permiten
 *   identificar de qué figura se trata
*/

struct s_triang
{
   float          perim;
   float          superf;
   enum t_figura  figura;
   union
   {
      float       radio;
      struct
      {
         float    base;
         float    altura;
      }           rectan;
   }u;
};
```

La ventaja de las uniones está en que ponen de manifiesto la lógica de un programa, mostrando qué tipo de información se requiere en cada caso, y que permiten el ahorro de espacio de almacenamiento cuando se compile el programa, el espacio de almacenamiento es el de la mayor estructura declarada en la unión .

Operadores :

Nivel	Denominación	Operador
1		() [] ->:: .
2	Unarios	! ~ + - ++ -- & * sizeof new delete (cast)
3	Multiplicativos	* / %
4	Accesso a miembros	.* ->*
5	Addittivos	+ -
6	Desplazamiento	<< >>
7	Relationales	< <= > >=
8	Igualdad	== !=
9		&
10		^
11		|
12		&&
13		||
14	Condicional	?:

15	Asignación	= *= /= %= += -= &= ^= |= <<= >>=
16	Coma	,

Para más detalle ver el documento correspondiente a precedencia de operadores del C .

Control de flujo del C :

Para el C dónde se requiere una expresión esta puede ser una expresión vacía (simplemente `;`) una única expresión válida (siempre terminada con `;`) o un bloque de expresiones (encerrada entre `{}` y que a su vez este bloque de expresiones puede contener una, varias o n inguna expresión) .

Condicional `if` (el `else` es opcional) :

```
if(<condición-o-expresión no void>)
   <expresión>;
else
   <expresión>;
```

Condicional `switch` (`default` y `break` son opcionales), es una expresión condicional de alternativas múltiples :

```
switch(<condición-o-expresión no void>)
{
   case <expresión constante entera> :
      <expresión>;   /* una o varias */
      break;
   case <expresión constante entera> :
      <expresión>;   /* una o varias */
      break;
   /* ... tantos case como sean necesarios */
   default :
      <expresión>;   /* una o varias */
}
```

Ciclo while :

```
while(<condición-o-expresión no void>)
   <expresión>;
```

Ciclo for :

```
for(<expresión1>; <expresión2>; <expresión3>)
   <expresión>;
```

La `<expresión1>` se ejecutará por única vez, habitalmente se emplea para inicialización de variables, si se necesita más de una expresión se separan con (`,`) .

La `<expresión2>` puede ser cualquier expresión válida C que se evaluará como expresión booleana (falso = 0, verdad = ~falso) antes de ejecutar la `<expresión>` .

La `<expresión3>` (habitalmente llamada incremento) se ejecutará cada vez que se haya ejecutado la `<expresión>` y antes de volver a evaluar la `<expresión2>` .

Ciclo do while :

```
do
{
   <expresión>;   /* una o varias */
} while(<condición-o-expresión no void>);
```

El empleo de las llaves (`{}`) es necesario cuando hay más de una `<expresión>` .

La expresión o sentencia **`break`** produce la interrupción:de las estructuras de control `switch`, `while`, `do-`

`while` o `for` y que se continúe con la siguiente expresión fuera de la estructura de control .

La expresión o sentencia **`continue`** produce la siguiente iteración:de las estructuras de control `while`, `do-while` o `for` .

La expresión o sentencia **`goto`** produce un salto incondicional a la etiqueta indicada .

Punteros :

Un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección ′`void`′ .

Puntero a una variable :

```
void main(void)
{
   int valor = 5;
   int *punt = &valor;
   void *paux = (void *)punt;

   printf("Valor contiene %d\n", valor);
   printf("Punt apunta a  %d\n", *punt);
...printf("Paux apunta a  %d\n", *(int *)paux);

   valor = 11;          /* equivale a : */
   *punt = 11;
   *(int *)paux = 11;
}
```

Puntero a un array :

```
void main(void)
{
   char texto[] = {"Hola" };
   char *p;

   while(*p ¡= '\0')
   {
      printf("%c", *p);
      p++;
   }
}
```

Arrays de punteros :

```
void main(void)
{
   char *nombres[35] = { "Lia", "Eli", "Susi" };
   int  ciclo;
   for(ciclo = 0; ciclo < 3; ciclo++)
      puts(nombres[ciclo]);
}
```

Argumentos de `main` .

```
void main(int argc, char *argv[], char *env[])
{
   int posi;
   printf("El nombre del programa es %s\n",
          argv[0]);
   printf("%d cadenas en la línea de comando\n"
          argc);
   for(posi = 0; posi < argc; posi++)
      puts(argv[posi]);

   for(posi = 0; argv[posi]; posi++)
      puts(argv[posi]);

   puts("Las variables de entorno son :");
   while(*env)
   {
      puts(*env);
      env++;
   }
}
```

Punteros a estructuras (uso del operador ->) :

```
typedef struct
{
   long dni;
   char apyn[31];
}t_info;

void main(void)
{
   t_info inf;
   t_info *p = &inf;
   p->dni = 8765432;
   strcpy(p->apyn, "Mary");
}
```

```
typedef struct
{
   long dni;
   char apyn[31];
}t_info;

void main(void)
{
   t_info inf[30];
   t_info *p = &inf;

   p->dni = 8765432L;
   strcpy(p->apyn, "Mary");

   p++;
   p->dni = 19876543L;
   strcpy(p->apyn, "Lia");

/* ... */
}
```

Funciones (argumentos por valor) :

```
void intercambio(int, int);

void main(void)
{
   int a = 4, b = 5;
   intercambio(a, b);
   /* a y b continuan conteniendo sus valores */
}

void intercambio(int v1, int v2)
{
   int aux = v1;
   v1 = v2;
   v2 = aux;
}
```

Funciones (argumentos por referencia) :

```
void intercambio(int *, int *);

void main(void)
{
   int a = 4, b = 5;
   intercambio(&a, &b);
   /* a y b continuan conteniendo sus valores */
}

void intercambio(int *v1, int *v2)
{
   int aux = *v1;
   *v1 = *v2;
   *v2 = aux;
}
```

Prototipos de funciones y archivos include :

En los dos ejemplos anteriores se ha escrito el prototipo de la función `intercambio` .

Punteros a funciones :

En C el nombre de una función es la dirección de memoria en que comienza la función .

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

int comp1(const void *p1, const void *p2)
{
   return strcmp((char *)p1, (char *)p2);
}
int comp2(const void *p1, const void *p2)
{
   if(*(long *)p1 == *(long *)p2)
      return 0;
   else
      if(*(long *)p1 < *(long *)p2)
         return -1;
      else
         return 1;
}

void informa(const void *p,
             const void *q,
             int(*fcmp)(const void *, const void *))
{
   int cmp = fcmp(p, q);
   if(cmp == 0)
      puts("Son Iguales");
   else
      if(cmp < 0)
         puts("El primero es menor");
      else
         puts("El segundo es menor");
}
```

```
void main(void)
{
   char texto1[] = { "Algo" },
        texto2[] = { "Alga" };
   long valor1   = 1234567890L,
        valor2   = 0xabcdef12L;


   printf("Primer cadena : \"%s\"\n"
          "Segunda cadena : \"%s\"\n",
          texto1, texto2);
   informa(texto1, texto2, comp1);

   printf("Primer  numero : \"%ld\"\n"
          "Segundo numero : \"%ld\"\n",
          valor1, valor2);
   informa(&valor1, &valor2, comp2);
}
```

Funciones que devuelven un puntero :

Entre las funciones de biblioteca hay infinidad de funciones estándar que devuelven punteros .

Salvo las variables `register` (que se almacenan en los registros del procesador) se puede tomar la dirección de memoria de cualquier variable .

Nosotros podemos escribir nestras propias funciones que devuelvan punteros, con la salvedad de que **no es lícito** devolver la dirección de variables `auto` . Ejemplifiquemos con una versión nuestra de `strcpy` :

```
/*
 *   la funcion de biblioteca strcpy recibe la
 * dirección en que debe almacenar (primer
 * argumento), y la direccion desde la que debe
 * copiar (segundo argumento), y devuelve la
 * dirección que recibe como primer argumento
 *   termina su cometido tras copiar el carácter
```

```
 * nulo
 *   no tiene previsto ningun tipo de control
 * respecto de que sea o no suficiente el
 * espacio de almacenamiento reservado para el
 * destino
 *   si los espacios de almacenamiento se
 * superponen, los resultados son impredecibles
 * y normalmente catastroficos
 */
char *strcpy(const char *dest, const char *orig)
{
   char *aux = (char *)p1;
   while(*orig)
   {
      *dest = *orig;
      dest++;
      orig++;
   }
   *dest = '\0';
   return aux;
}
```

Funciones que reciben puntero a puntero :

Una variable puntero a puntero es exactamente eso : una variable que contiene la dirección de memoria de una variable puntero . Con un puntero a puntero podemos alterar la dirección a la que que apunta un puntero .

```
#include <stdio.h>

void func(int **, int *);
void main(void)
{
   int   alto  = 10,
         ancho = 25,
        *p;
   func(&p, &alto);  /* p apunta a alto */
   printf("Alto  = %d\n", *p);
   func(&p, &ancho);
   printf("Ancho = %d\n", *p);
}
void func(int **pp1, int *p2)
{
   *pp1 = *p2;
}
```

Recursividad :

La recursividad es una facilidad que poseen algunos lenguajes con lo que se facilita que una función escrita por nosotros se pueda invocar a si misma . Veamos por ejemplo una función recursiva que determina la cantidad de caracteres de un array de char .

```
#include <stdio.h>

unsigned str_len(const char *);
void main(void)
{
   char texto[] = { "Hola"};

   printf("\"%s\" tiene %u caracteres\n",
          texto,
          str_len(texto));
}

unsigned str_len(const char *p)
{
   if(*p)
      return str_len(p + 1) + 1;
   else
      return 0;
}
```