Fase 1

Especificación del Lenguaje

2010

El lenguaje cuyo compilador se implementará a lo largo de las fases de la práctica tendrá las características de un lenguaje de programación de alto nivel. Su complejidad estará en relación con el número de miembros del grupo. Como mínimo deberá incorporar declaraciones de variables numéricas, instrucciones de asignación de expresiones, estructuras de control, rutinas recursivas, así como operaciones de salida para variables y ristras.

Autores

Pablo Eduardo Ojeda Vasco
Roberto Marco Sánchez

A Índice

1 Introducción

El lenguaje de programación escogido para llevar a cabo su especificación ha sido C. Los motivos que nos han llevado a ello han sido muy diversos. Entre ellos destacamos:

- Se trata de un lenguaje que no está ligado a ningún sistema operativo ni a ninguna máquina, y aunque se le llama "lenguaje de programación de sistemas" debido a su utilidad para escribir compiladores y sistemas operativos, se utiliza con igual eficacia para escribir importantes programas en diversas disciplinas.

- Es un lenguaje del 1 nivel medio que puede tomar características de los lenguajes de alto nivel y de los de bajo nivel (lenguaje ensamblador o lenguaje de máquina).

- C proporciona las construcciones fundamentales de control de flujo que se requieren en programas bien estructurados de propósito general: agrupación de proposiciones, toma de decisiones (if-else), selección de un caso entre un conjunto de ellos (switch), iteración con la condición de paro en la parte superior (While, for) o en la parte inferior (do), y la terminación prematura de ciclos (break).

- Su portabilidad, o sea que se puede llevar fácilmente de una computadora a otra, ya que las funciones automatizan la mayoría de las características de las computadoras.

- El compilador transforma grandes programas en pequeños programas objetos que se ejecutan fácilmente.

- Su uso extensivo de llamadas a funciones facilitan grandemente su entendimiento y depuración, es decir, su modularidad y eficiencia en el diseño de programas.

- Hace posible la escritura de programas fuente con cualquier procesador de palabras que maneje el formato ASCII (Word, Office, Notepad++, etc.) debido en gran parte al elevado número de operadores que incluye C. Luego se pasa el programa fuente a C para que sea compilado, y se pueda ejecutar.

Después de la compilación existe otro paso, el encadenamiento, el cual es necesario por diferentes razones:

1) El programa utilizará varias rutinas de biblioteca las cuales realizan ciertas funciones como permitir la entrada/salida de datos.

2) Cuando un programa muy grande consta de varios archivos separados, se puede no desear compilarlos todos al mismo tiempo.

2 Tipos básicos de datos

2.1 Tipos de datos

char Carácter o números pequeños -128 a 127
int Entero -2.147.483.648 a 2.147.483.647
float Real 3.4x10e-38 a 3.4x10e38
double Real con mayor o igual precisión que el float 1.7x10e-308 a 1.7x10e308
void dato descartado, tipo comodín

Ejemplos:

```
int i;     /* entero */
char c; /* carácter */
double x; /* variable real de doble precisión */
```

Modificador const

Es posible usar el modificador **const** para establecer variables no modificables una vez que se inicializan.

2.2 Literales

C dispone de formatos para constantes literales del tipo entero, real, carácter, ristra y enumerados. Estos dos últimos tipos los describiremos en su momento.
Los enteros se pueden representar en tres sistemas numéricos distintos: decimal, octal y hexadecimal. Un número está escrito en hexadecimal si comienza por "0x" o "0X". Los caracteres que representan el 10, 11, 12, 13, 14 y 15 son a, b, c, d, e y f, en minúscula o mayúscula. Los números en octal son los que comienzan por 0. Los números en decimal son el resto.

Literal	Descripción
0x10	16 en hexadecimal
0XAA	170 en hexadecimal
010	8 en octal
025	21 en octal
0x11r	Error, "r" no es un dígito válido
019	Error, 9 no es un dígito válido

Los números en coma flotante se escriben en base 10 y contienen un puto decimal o un exponente o ambos. El formato sería un número decimal con o sin signo que represente la parte entera, opcionalmente un punto y la parte decimal. Se puede establecer un exponente añadiendo la letra e, minúscula o mayúscula, y un entero que represente el exponente en base diez.

Literal	Descripción
10.1	
5e10	50000000000
1.1e-20	0.00000000000000000011

Para representar caracteres no imprimibles se usa una secuencia de escape que comienza por "\" seguido de uno o varios caracteres. Por ejemplo para representar el salto de línea (LF) se puede representar por '\n'.

Literal	Descripción
'a'..'z'	Caracteres imprimibles
'\"	Representa una comilla simple
'\\'	Representa una "\"
'\r'	Carácter de retorno de carro
'\0'	Carácter nulo

3 Operadores

3.1 Operadores aritméticos

Operadores Aritméticos			
Operador	**Descripción**	**Ejemplo a=3, b=2**	**Resultado**
+	Suma	a+b	5
-	Resta y valor negativo (unario)	a-b-b	1-2
*	Multiplicación	a*b	6
/	División	a/b	1
%	Resto	a%b	1

Los operadores anteriores se pueden usar en todos los tipos numéricos, con la excepción del módulo que sólo se puede emplear en enteros y caracteres.

Operador de incremento (++), decremento (--)

Estos operadores son monarios y realizan el incremento o decremento, respectivamente, de la variable a la que se le aplica. Además de la acción de modificar la variable devuelven el valor de la variable.

El operador de incremento o decremento puede ir delante o detrás de la variable, teniendo diferente significado. Si el operador "++" se sitúa después de la variable se denomina "postincremento", haciendo que primero se tome el valor y después se incrementa la variable. Ejemplo:

```
n = k++; /* El valor original de k se asigne a n y luego se
incrementa k */
```

Si el operador "++" se sitúa antes de la variable se denomina "preincremento" y hace que primero se incrementa la variable y después se tome el valor. Ejemplo:

```
n = ++k; /* Primero se incrementa k y luego se asigna a n */
```

3.2 Operadores relacionales

Operadores Relacionales			
Operador	**Descripción**	**Ejemplo**	**Resultado**
>	Mayor que	2 > 3	0
<	Menor que	2 < 3	1

>=	Mayor o iqual que	2 >= 3	0
<=	Menor o iqual que	2 <= 3	1
==	Igual	2 == 3	0
!=	Distinto	2 != 3	1

3.3 Operadores lógicos

Operadores Lógicos			
Operador	**Descripción**	**Ejemplo**	**Resultado**
&&	y (and)	(a > b) && c	0
||	o (or)	(c > a) || c	1
!	no (not)	!(a > c)	1

3.4 Operadores a nivel de bits

Operadores a nivel de bits			
Operador	**Descripción**	**Ejemplo (a=1, b=2, c=3)**	**Resultado**
&	y (and) bit a bit	a & b a & c	0 1
|	o (or) bit a bit	a | b	3
^	o (or) exclusivo bit a bit	a ^b	2
~ (ALT-126)	complemento a uno de bits. Invierte todos y cada uno de los bits del operando. Formato: ~operando	~a	Depende del tipo y rango de a
<<	desplazamiento a la izquierda del primer operando tantas veces como indique el segundo. Rellena de ceros los bits menos significativos.	a << b c << b	4 12
>>	desplazamiento a la derecha del primer operando tantas veces como indique el segundo. Los bits más significativos se rellenan de ceros si	c >> a 15 >> 2	1 3

	el primer operando es unsigned y si no se rellenan de unos.		

3.5 Operadores de asignación

Operadores de Asignación		
Operador	**Ejemplo**	**Equivalencia**
=	a=b=c	a=c; b=c;
+=	a+=3	a=a+3
-=	a-=3*b	a=a-(3*b)
*=	a*=2	a=a*2
/=	a/=35+b	a=a/(35+b)
%=	a%=8	a=a%8

3.6 Precedencia de operadores

L respetará la precedencia matemática habitual:

1.- ( )
2.- * , /
3.- +, -

```
* comentario */
/ comentario
```


4 Comentarios

4.1 Comentarios Multilínea

Para poner comentarios se sigue el siguiente formato:

Un comentario puede ocupar varias líneas. No se pueden poner comentarios anidados; se considera comentario todo lo incluido entre /* y */.

4.2 Comentarios a nivel de línea

Para poner comentarios se sigue el siguiente formato:

Un comentario de este tipo sólo puede ocupar una línea.

```
 = &i; /* ahora pi señala a la variable i */
pi = 3; / le asignamos un 3 a i */

nt i, *pi
```


5 Tipos de datos construidos

4.1 Punteros

Se utilizan para contener direcciones de memoria en C. Se definen con un "*" antes del nombre de la variable puntero. Para acceder al contenido de la dirección a la que apunta se usa de nuevo el "*" procediendo la variable puntero.

Operador &

El operador & es un operador unitario que devuelve la dirección de lo que sigue a continuación. Este operador se puede aplicar a una variable simple o a una expresión que nos lleve a una componente de una estructura más compleja. Como por ejemplo el elemento de un vector o un campo de una estructura. El uso de este operador permite asignar la dirección a cualquier variable a un puntero. A través del puntero se puede acceder a la variable.

Operador *

El operador "*" al ser aplicado a un puntero permite acceder a la variable a la que apunta, pudiendo tomar su valor o modificarlo. El formato es "*expresión", donde la expresión debe ser un puntero o devolver la dirección de una variable.

4.2 Vectores y matrices

Un vector es una colección de elementos del mismo tipo. En C pueden construirse matrices de cualquier dimensión. En este contexto un vector se considera una matriz unidimensional. Una matriz bidimensional es un vector de vectores, extendiéndose esta propiedad a matrices con más dimensiones.
Es necesario destacar dos propiedades de los vectores y matrices en lenguaje C:

1. El subíndice del primer elemento de cada dimensión es cero.
2. El nombre de un vector es un puntero constante a la dirección del primer elemento.

Como consecuencia de la segunda propiedad no es posible asignar directamente un vector a otro. Es como si intentásemos modificar un puntero constante.

El formato de definición de vectores y matrices es:

```
truct [NombreEstructura]
{
  … /* Campos. Definidos como las variables */
} [Definición_de_variables];
```

```
TipoDeDato NombreVector[TamañoDimension1][TamañoDimension2] …;
```

Ejemplos:

```
/* Declara un puntero a caracteres */
int *a[10]; /* define un vector de 10 enteros */
char *Ristra; /* se apunta a zona de memoria con ristra */
float c[5][7]; /* define una matriz bidimensional */
```

4.3 Ristras de caracteres

En el lenguaje C no existe un tipo de dato específico para cadenas de caracteres. Las cadenas de caracteres se representan como vectores de caracteres. Cada elemento del vector es un carácter de la ristra. Tras el último carácter de la ristra debe haber un carácter especial que marca el final de ésta, este carácter es el nulo, ASCII 0, también representado en C como '\0'.

Los literales de ristras se escriben como una secuencia de caracteres encerrados entre comillas dobles ("). Estos literales cuando están fuera de las inicializaciones representan direcciones del tipo "char *" a una zona de memoria donde el compilador almacena el literal en cuestión.

Ejemplo:

```
char *Ristra;
Ristra = "Esto es una ristra";
```

4.4 Tipos struct y union

Las estructuras en C son una colección de variables que forman una unidad. Cada una de las variables, denominadas campos, contenidas en la estructura pueden ser de distinto tipo y debe tener un identificador distinto.

Formato:

```
struct [NombreEstructura]
{
  TipoDeDato1 Campo1;
  TipoDeDato2 Campo2;
  …
} [Definición_de_variables];
```

El nombre del tipo de estructura y la definición de variables son opcionales, pero al menos debe existir una de ellas ya que, si no, la definición no tendría efecto. El nombre de la estructura sirve para usarlo en posteriores declaraciones o definiciones de variables de este tipo. La parte de "definición de variables" del formato nos permite definir variables de este tipo de forma inmediata e incluso sin necesidad de la estructura que tenga un nombre concreto.

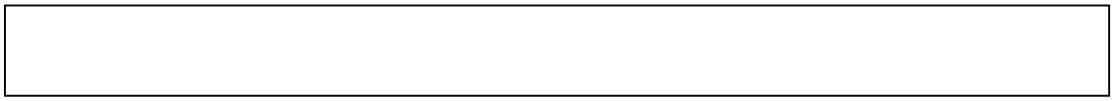
Operador "."

La forma de acceder a los campos de una variable de tipo **struct o unión** es empleando el operador ".", Variable campo.

Ejemplo:

La **unión** tiene exactamente el mismo formato y funcionalidad que struct con la única diferencia de que todos los campos comparten el mismo espacio de memoria. En un determinado momento sólo se selecciona uno.

Ejemplo:



Significa que m puede contener el entero i o el carácter c.

4.5 Definición de tipos

En C es posible definir nuevos tipos de variables a partir de construcciones con los tipos ya definidos. Para la definición de nuevos tipos se usa la palabra reservada **typedef.** El formato de uso de **typedef** es el siguiente:

```
typedef "definición de variables";
```

donde "definición de variables" es cualquier construcción válida que permita definir una o más variables derivadas de un tipo primitivo, o ya definido. El efecto es que las variables que se hubiesen definido se convierten en nuevos tipos. El nuevo tipo es como el de la variable que se hubiese definido sin el **typedef.**

6 Estructuras de Control

6.1 Inicialización de variables:

Las variables pueden inicializarse a la vez que se definen. La inicialización se produce una única vez, cuando la variable comienza su existencia.

6.2 Sentencia if:

La sentencia **if** tiene la forma:

```
if( Condición ) Instrucción1;
[else Instrucción2;]
```

Los paréntesis asociados que delimitan la condición no son opcionales. En caso de que la condición sea verdadera se ejecuta la instrucción1; en caso contrario se ejecuta, si existe, la instrucción2.

6.3 Sentencia while:

La sentencia **while** tiene la forma:

```
while (Condición) Instrucción;
```

Los paréntesis no son opcionales. Si se cumple la condición se ejecuta la instrucción y se repite el proceso.

6.4 Sentencia do-while:

La sentencia tiene la forma:

```
do
    Instrucción;
While (Condición);
```

6.5 Sentencia for:

C tiene una sentencia **for** que difiere bastante de la de otros lenguajes. Su forma es:

```
for ([inicialización]; [condición]; [expresión]) Instrucción;
```

El equivalente de esta expresión con while es:

```
inicialización;
while (condición)
{
 Instrucción;
 expresión;
}
```

Cada una de las partes del for se pueden omitir. En el caso de la condición, su omisión equivale a una condición siempre verdadera. Si se desea ejecutar más de una instrucción en una de las expresiones, se puede emplear el operador ",".

6.6 Sentencia switch:

La sentencia **switch** hace que se seleccione un salto en la ejecución dependiendo del valor de una expresión. Tiene la forma:

```
switch(Expresión)
{
 case Valor1:
 ...
 [break;]
 case Valor2:
 ...
 [break;]
 case Valor3:
 ...
 [break;]
 [default:]
 ...
}
```

La expresión entre paréntesis del **switch** debe ser entera. Su resultado se comparará con los distintos valores en los **case**. Si coincide con uno de ellos se pasará la ejecución a la instrucción siguiente al **case** con dicho valor y se seguirán ejecutando las instrucciones consecutivas hasta encontrar una instrucción **break** o alcanzar el cierra llaves del **switch**. En caso de que el resultado de la expresión no coincida con ningún valor se pasará la ejecución a la instrucción siguiente a la etiqueta **default**:, si la hubiera, y se continuará como en un **case**. Los valores en los **case** pueden ser una expresión constante. No puede haber dos **case** con el mismo valor.

7 Funciones

7.1 Definición de funciones:

Las funciones se definen indicando primero qué tipo de valor devuelven, después el nombre de la función y a continuación los parámetros que acepta encerrados entre paréntesis y separados por comas. El formato de la definición de funciones es el siguiente:

```
tipo_devuelto nombre_función(tipo1 param1,tipo2 param2, ...)
{
  ...
}
```

En C no existen procedimientos, sólo existen funciones. Una función es equivalente a un procedimiento cuando el tipo que devuelve es ***void*** . También hay que tener en cuenta que por las características de C se puede despreciar el valor resultante de cualquier expresión lo que permite que se pueda ignorar el valor devuelto por una función.
Las definiciones de funciones no pueden anidarse; todas las funciones se encuentran al mismo nivel de abstracción. No es posible definir una función local a otra. Lo que no quiere decir que una función no pueda llamarse a sí misma o a cualquier otra.

7.2 Paso de parámetros a una función:

Los parámetros en C se pasan siempre por valor; lo que quiere decir que se produce una copia del **valor** pasado al parámetro de la función, el cual no actúa sino como una variable local a la función pero con una inicialización externa.
En C no está permitido pasar por valor a una función parámetros que sean vectores, o funciones, si bien se pueden pasar punteros a éstos. El nombre de un vector, cuando se escribe sin subíndice, se interpreta como un puntero al primer elemento del vector.

7.3 Funciones sin parámetros:

Una función sin parámetros se establece poniendo como único parámetro **void**. En caso de definir una función sin parámetros estaríamos indicando que acepta cualquier número de parámetros.

7.4 Sentencia return:

La forma en que las funciones en C devuelven un valor es empleando la sentencia **return**. Esta sentencia puede aparecer en cualquier punto de la función y cuantas veces sea necesario. Su ejecución implica el inmediato abandono de la función. El valor devuelto por la función se obtiene de la expresión que se añade al **return**. En el caso de los procedimientos, funciones que devuelven **void**, también es posible emplear **return** para abandonar la función en cualquier punto.

7.5 Recursividad:

La recursividad es una técnica de programación importante. Se utiliza para realizar una llamada a una función desde la misma función. Como ejemplo útil se puede presentar el cálculo de números factoriales.

8 Entrada/Salida

8.1 Printf

Sintaxis:
int printf(const char* formato,...)

La cadena constante **formato** provee una descripción de la salida, con placeholders marcados por caracteres de escape "%", para especificar la localización relativa y el tipo de salida que la función debe producir.

- **%d}**: escribir un entero;
- **%o}**: escribir en octal;
- **%x}**: escribir en hexadecimal;
- **%u}**: escribir en entero sin signo;
- **%f}**: escribir un flotante o doble;
- **%%}**: escribir caracter '%';

La función printf retorna el número de caracteres impresos, o un valor negativo si ocurre un error.

8.2 Scanf

La función scanf() (*scan-format*, escanear con formato), en realidad representa a una familia de funciones que escanean una entrada de datos con formato y cargan el resultado en los argumentos que se pasan por referencia a dicha función o funciones:

- La función scanf() lee los datos de entrada en el **stdin** (flujo de entrada estándar).
- La función fscanf() (*file-scanf*) lee en un flujo de entrada dado, por lo general un fichero (file) abierto para lectura.
- La función sscanf() (*string-scanf*) obtiene la entrada a escanear de una cadena de caracteres dada (*string*).

Todas ellas leen octetos, los interpretan según un formato, y almacenan los resultados en sus argumentos. Cada uno cuenta con varios argumentos: por un lado, un formato de la secuencia del control (se describe más abajo), por otro, un sistema de argumentos del indicador que señala dónde la entrada convertida debe ser almacenada. El resultado es indefinido si hay escasos argumentos para dar formato. Si se agota el formato mientras que sigue habiendo las argumentos, los argumentos sobrantes son evaluados pero no procesados de ninguna otra manera.

Sintaxis:
scanf(tipo, &var);

- tipo: Tipo de dato a almacenar

- ampersand (&) se utiliza para indicar una dirección de memoria de la variable donde se almacenará el dato. Cuando se guardan de cadenas de caracteres, al tratarse de un array de tipo char, el & se omite.
- var: variable para almacenar el dato.