Contenidos

CONTENIDOS 1

INTRODUCCIÓN ..... ¡ERROR! MARCADOR NO DEFINIDO.

OBJETIVO ..... **¡ERROR! MARCADOR NO DEFINIDO.**
RELACIÓN CON EL PLAN DE ESTUDIOS 1995 ..... **¡ERROR! MARCADOR NO DEFINIDO.**
ORGANIZACIÓN Y CONTENIDOS 3
ESTÁNDARES, LENGUAJE Y BIBLIOTECA 3

1. LENGUAJE 4

1.1. SOBRE LA SINTAXIS 4
1.2. SOBRE LAS GRAMÁTICAS 4
1.3. SOBRE LA SINTAXIS DEL ANSI C 5
1.4. GRAMÁTICA LÉXICA 9
1.4.1. Elementos Léxicos 9
1.4.2. Palabras Reservadas 9
1.4.3. Identificadores 9
1.4.4. Constantes 10
1.4.5. Constantes Cadena 12
1.4.6. Punctuators – Caracteres de Puntuación 12
1.4.7. Nombre de Encabezados 13
1.4.8. Números de Preprocesador 13
1.5. GRAMÁTICA DE ESTRUCTURA DE FRASES 13
1.5.1. Expresiones 13
1.5.2. Declaraciones 16
1.5.3. Sentencias 18
1.5.4. Definiciones Externas 18
1.6. GRAMÁTICA DEL PREPROCESADOR 19

2. BIBLIOTECA 21

2.1. DEFINICIONES COMUNES <STDDEF.H> 22
2.2. MANEJO DE CARACTERES <CTYPE.H> 22
2.3. MANEJO DE CADENAS <STRING.H> 23
2.3.1. Concatenación 23
2.3.2. Copia 23
2.3.3. Búsqueda y Comparación 23
2.3.4. Manejo de Memoria 24
2.4. UTILIDADES GENERALES <STDLIB.H> 24
2.4.1. Tips y Macros 24
2.4.2. Conversión 25
2.4.3. Administración de Memoria 25
2.4.4. Números Pseudo-Aleatorios 25
2.4.5. Comunicación con el Entorno 26
2.4.6. Búsqueda y Ordenamiento 26
2.5. ENTRADA / SALIDA <STDIO.H> 27
2.5.1. Tipos 27
2.5.2. Macros 27

2.5.3. Operaciones sobre Archivos 27
2.5.4. Acceso 28
2.5.5. Entrada / Salida Formateada 28
2.5.6. Entrada / Salida de a Caracteres 29
2.5.7. Entrada / Salida de a Cadenas 29
2.5.8. Entrada / Salida de a Bloques 29
2.5.9. Posicionamiento 30
2.5.10. Manejo de Errores 30
2.6. OTROS 31
2.6.1. Hora y Fecha <time.h> 31
2.6.2. Matemática 31
2.7. LOS FORMATOS 32
2.7.1. Funciones printf, sprintf, fprintf 32
2.7.2. Funciones scanf, sscanf, fscanf 34

3. IMPLEMENTACIONES 35

3.1. TIPOS DE DATOS PRIMITIVOS – TAMAÑO Y RANGO 35
3.2. FUNCIONES NO ESTÁNDAR 36
3.2.1. Implementación Borland – Funciones sobre la Consola 36

4. BIBLIOGRAFÍA ..... ¡ERROR! MARCADOR NO DEFINIDO.

Organización y Contenidos

El módulo está organizado en tres secciones bien diferenciadas.

- Lenguaje. Presenta la descripción del Lenguaje C, previa explicación de los conceptos *sintaxis* y *gramática* de lenguajes de programación y de la notación *BNF*.
- Biblioteca. Reúne los componentes más importantes y utilizados de la Biblioteca Standard, agrupados por funcionalidad. Por cada función se presenta su prototipo y una breve sinopsis, junto con los tipos, objetos y macros declarados en los diferentes encabezados que facilitan el uso de dichas funciones.
- Implementaciones. Presenta ejemplos de implementación de tipos de datos primitivos y funciones no estándar de uso frecuente.

Estándares, Lenguaje y Biblioteca

El estándar que norma al *Lenguaje de Programación C* es el publicado en el documento **ANSI X3.159-1989**, en 1989 y ratificado internacionalmente por el documento **ISO/IEC 9898:1990** en 1990. Este lenguaje es conocido indistintamente como **ANSI C**, **ISO C**, **C89** ó **C90**. Actualmente se encuentra publicada en **ISO/IEC 9898:1990** una segunda versión, el **C99** pero existen pocas implementaciones comerciales del mismo a diferencia del muy extendido C90. El curso, y por lo tanto el presente módulo, giran entorno al C90.

Es importante destacar que el estándar norma dos aspectos importantes y bien diferenciados del Lenguaje de Programación C, el *Lenguaje* en sí mismo y la *Biblioteca*. A manera de ejemplo y en pocas palabras, el Lenguaje se refiere a asuntos como la forma correcta de escribir (*sintaxis*) una *sentencia de selección* y establecer el significado (*semántica*) de ciertas construcciones. Por otro lado, la sección Biblioteca del estándar se encarga, entre otras cosas, de definir la especificación de funciones (e.g. funciones de entrada / salida, de manejo de memoria, etc.) de uso común en todas las plataformas. Estas funciones deben estar disponibles en todas las implementaciones y serán estas las funciones que utilizaremos a lo largo del curso.

1. Lenguaje

1.1. Sobre la Sintaxis

Un Lenguaje de Programación está compuesto por un conjunto de *Lenguajes Regulares* y otro conjunto de *Lenguajes Independientes del Contexto*.

Los *componentes léxicos (tokens)* (identificadores, números enteros, números reales, caracteres constantes, cadenas constantes, operadores y caracteres de puntuación) constituyen diferentes Lenguajes Regulares. Algunos de estos lenguajes son *finitos*, como los operadores y los caracteres de puntuación, y otros son *infinitos*, como sucede con los identificadores o los números reales. En ambos casos, estos lenguajes pueden ser *generados* por *Gramáticas Regulares* y *descriptos* mediante *Expresiones Regulares*.

En cambio, las expresiones y las sentencias de un Lenguaje de Programación son, en general, *Lenguajes Independientes del Contexto*. Como tales, estos lenguajes no pueden ser generados por Gramáticas Regulares sino que requieren *Gramáticas Independientes del Contexto*.

Una gramática formal *no solo genera* un lenguaje formal, sino que también *se puede utilizar para describir* la sintaxis del lenguaje generado. Como veremos en la descripción sintáctica del ANSI C, aún muchos componentes léxicos son descriptos mediante Gramáticas Independientes del Contexto, aunque, en este caso, la estructura de las producciones es muy similar a la de las Gramáticas Regulares, como se observará en el siguiente análisis.

Nota: Los no-terminales se representan mediante nombres o frases encerrados entre corchetes angulares (< y >), como, por ejemplo, `<identificador>`; por otro lado, los terminales se resaltan en "**`negritas`**".

En las descripciones gramaticales que siguen, hay varios no-terminales con producciones del tipo v -> v | vw, en las que tanto v como w son no-terminales, pero donde w actúa como "simplificación" de un grupo de terminales. Un ejemplo de esta situación es la producción:

```
<identificador> -> <identificador> <dígito>
```

Por lo tanto, ese tipo de producción es "quasi-lineal a izquierda" y, por ser recursiva, es la producción base en la generación del lenguaje regular `w+`.

1.2. Sobre las Gramáticas

Las gramáticas formales utilizadas para definir la sintaxis de los Lenguajes de Programación pertenecen a una categoría llamada *Gramáticas Independientes del Contexto (GICs).*

La característica fundamental de una GIC es que todas sus producciones son de la forma: variable -> secuencia de variables y/o terminales, es decir:

```
variable -> (variable + terminal)*
```

Ejemplos:

```
S -> a   corresponde a variable -> terminal
S -> Ra  corresponde a variable -> variable terminal
R -> aa  corresponde a variable -> terminal terminal
T -> QPZ corresponde a variable -> variable variable variable
```

Los Lenguajes Formales más importantes son los lenguajes infinitos. Consecuentemente, las GICs más importantes son aquellas que generan lenguajes infinitos.

En la descripción sintáctica del ANSI C encontraremos algunas GICs que describen lenguajes finitos y otras GICs que describen lenguajes infinitos.

1.3. Sobre la Sintaxis del ANSI C

Analizaré -con todos ustedes- aspectos importantes de la sintaxis del ANSI C, ampliando lo que escribí en las dos secciones anteriores, Sobre la Sintaxis y Sobre la Gramática.

La sintaxis de un Lenguaje de Programación debe describirse con precisión, utilizando una notación sin ambigüedades. La notación que utilizamos en este módulo, comúnmente llamada **BNF extendida**, deriva de la usada en la representación de las reglas o producciones de una GIC.

Ejemplo 1

Describamos una estructura sintáctica de dos maneras:

- Mediante un lenguaje natural, normalmente ambiguo;
- Mediante una notación especial no-ambigua.

Si queremos especificar (o definir) informalmente la sintaxis de la **sentencia WHILE** en **Pascal**, utilizando el lenguaje castellano, podemos decir:

```
Una sentencia WHILE comienza con la palabra reservada WHILE,
     que es seguida de una expresión Booleana,
     que es seguida de la palabra reservada DO,
     que es seguida de una sentencia.
```

Si, en cambio, utilizamos una notación BNF como la que es utilizada en este módulo, la sintaxis de la **sentencia WHILE** en Pascal la especificamos de esta manera:

```
<sentencia-WHILE> -> WHILE <expresión Booleana> DO <sentencia>
```

1. ¿Comprende cabalmente esta última descripción?
2. ¿Cuál especificación le parece mejor y por qué?

Ejemplo 2

Como escribe David Watt en su libro "*Programming Language Syntax and Semantics*" (1991, Prentice Hall): La siguiente es la especificación informal de la sintaxis de **identificadores**, parafraseada de un manual de un viejo lenguaje de programación:

```
Un identificador es una secuencia de letras,
     posiblemente con la inclusión de subrayados (guiones bajos) en medio.
```

A partir de esta especificación, es evidente que `PI`, `CANTIDAD` y `CANTIDAD_TOTAL` son identificadores correctos. También es muy claro que `PI34`, `78` y `CANTIDAD_` no son identificadores válidos.

3. ¿Está de acuerdo?

Pero, si nos atenemos a esa definición: 1) ¿Es una única letra, como X, un identificador válido?; 2) ¿la frase "subrayados en medio" significa que puede haber varios consecutivos o que deben estar separados o que ...?

4. ¿Puede responder con seguridad a estas preguntas?

5. ¿Se le ocurre alguna otra *ambigüedad* en la especificación informal que estamos analizando?

Esas imprecisiones o ambigüedades son casi inevitables en una especificación informal, como la que se realiza utilizando un lenguaje natural.

Ahora, supongamos que usamos una notación BNF para describir estos identificadores, y lo hacemos de la siguiente manera (asumiendo que ya hemos definido <letra>):

```
<identificador> ->
     <letra> | <identificador> <letra> | <identificador> _ <letra>
```

Aclarando que el símbolo '|' significa 'o', las líneas anteriores involucran tres reglas que actúan en la definición de nuestro ya famoso "identificador".

Observe que la primera regla nos dice que un identificador puede ser una letra (por ejemplo, X), mientras que las otras dos reglas son *recursivas* (porque `<identificador>` aparece en ambos lados de la regla). Estas reglas recursivas permiten generar, muy simplemente, un número infinito de identificadores mediante la aplicación sucesiva de esas reglas.

Ejemplo 3

```
<identificador> ->
     <identificador> <letra>
     <identificador> A
     <identificador> <letra> A
     <identificador> BA
     <identificador> _ <letra> BA
     <identificador> _XBA
     <letra> _XBA
     R_XBA
```

6. ¿Comprende cómo generar cualquier identificador?

7. De acuerdo a la descripción formal en BNF, ¿puede un identificador comenzar con un subrayado?

8. ¿Es **A__B** un identificador válido? ¿Lo puede generar?

9. ¿Puede generar el identificador **A_B_C**?

10. ¿Podría describir en castellano, sin ambigüedades, el conjunto de identificadores válidos en este antiguo Lenguaje de Programación?

Luego de leídas las dos secciones previas, pasamos a iniciar el análisis de la "Sintaxis del ANSI C", cuya especificación en BNF comienza con la siguiente sección, "Gramática Léxica".

En la tercera subsección, “Identificadores”, se especifica cómo son los identificadores en ANSI C. La notación utilizada ya no debe presentar dificultades. Sin embargo, aparece algo nuevo: la frase “*uno de*”.

Esta la primera incorporación novedosa que hacemos a la notación BNF que usamos hasta ahora. La empleamos para mejorar la legibilidad cuando existen varios ‘o’, es decir: por ejemplo, sin el uso de “*uno de*”, <dígito> debería especificarse así:

<dígito> -> **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

y esta enumeración no es tan legible como la que ocurre con el uso de “*uno de*”.

11. De acuerdo a la especificación de la página 4, ¿es **___123** un identificador válido en ANSI C?

En la sección *Identificador* se especifica cómo son los identificadores en ANSI C. En la notación utilizada, llamada *BNF Extendida*, se utilizan algunos símbolos especiales que colaboran en la definición o especificación de la correspondiente construcción.

Por ejemplo, para definir la sintaxis de <identificador> se utilizan ciertos símbolos especiales:

<, >, ->, |, *uno de*

Cada uno de ellos tiene una aplicación especial y se denomina *metasímbolo*.

Seguimos adelante; encontramos la sección *Constante Real*. En esta sección se define, obviamente, la sintaxis correcta de cualquier constante real correcta en ANSI C.

Comienza diciendo que una <constante real> adopta dos formas posibles. En esta especificación aparece un nuevo metasímbolo: el representado por el "**?**". Este metasímbolo significa que el elemento que lo precede puede figurar o no.

Por ejemplo, “<parte exponente>**?**” significa que <parte exponente> puede figurar o no en la construcción de una constante real.

Entonces, la primera línea de esta especificación:

```
<constante real> -> <constante fracción> <parte exponente>? <sufijo real>?
```

se lee de esta manera: "una *constante real* en ANSI C es una *constante fracción*, seguida *eventualmente* de una *parte exponente*, seguida *eventualmente* de un *sufijo real*.

Por lo tanto, esta notación compacta permite definir cuatro tipos diferentes de constantes reales:

```
<constante real> -> <constante fracción> <parte exponente> <sufijo real>
<constante real> -> <constante fracción> <sufijo real>
<constante real> -> <constante fracción> <parte exponente>
<constante real> -> <constante fracción>
```

Para completar la definición de una constante real, debemos analizar la definición de `<constante fracción>`, la de `<parte exponente>` y la de `<sufijo real>`.

12. Termine el análisis de la especificación sintáctica de la `<constante real>` en ANSI C, escriba 2 ejemplos de cada caso posible y escriba 5 "constantes reales" erróneas.

1.4. Gramática Léxica

1.4.1. Elementos Léxicos

```
<token> ->
     <palabra reservada> |
     <identificador> |
     <constante> |
     <literal de cadena> |
     <punctuator>
<token de preprocesamiento> ->
     <nombre de encabezado> |
     <identificador> |
     <número de preprocesador>|
     <constante carácter> |
     <literal de cadena> |
     <punctuator> |
     cada uno de los caracteres no-espacio-blanco que no sea uno de los anteriores
```

1.4.2. Palabras Reservadas

```
<palabra reservada> -> una de
     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
```

1.4.3. Identificadores

```
<identificador> ->
     <no dígito> |
     <identificador> <no dígito> |
     <identificador> <dígito>
<no dígito> -> uno de
     _ 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
<dígito> -> uno de
     0 1 2 3 4 5 6 7 8 9
```

- Toda implementación debe distinguir, como mínimo, los primeros 31 caracteres de un identificador que actúa como nombre de variable, de función, de constante o de tipo.

- Dado que los identificadores constituyen un Lenguaje Regular, podemos describirlos mediante la definición regular que figura en el Ejemplo 6 del libro "Autómatas Finitos y Expresiones Regulares", página 113:

```
<letra> = [a-zA-Z] (cualquier letra minúscula o mayúscula del alfabeto reducido)
<dígito> = [0-9]
<subrayado> = _
<primer carácter> = <letra> | <subrayado>
<otro carácter> = <letra> | <dígito> | <subrayado>
<identificador> = <primer carácter> <otro carácter>*
```

1.4.4. Constantes

```
<constante> ->
     <constante entera> |
     <constante real> |
     <constante carácter> |
     <constante enumeración>
```

- En general, en computación las constantes enteras *no* son un subconjunto de las constantes reales.

Constante Entera

```
<constante entera> ->
     <constante decimal> <sufijo entero>? |
     <constante octal> <sufijo entero>? |
     <constante hexadecimal> <sufijo entero>?
<constante decimal> ->
     <dígito no cero> |
     <constante decimal> <dígito>
<dígito no cero> -> uno de
     1 2 3 4 5 6 7 8 9
<dígito> -> uno de
     0 1 2 3 4 5 6 7 8 9
<constante octal> ->
     0 |
     <constante octal> <dígito octal>
<dígito octal> -> uno de
     0 1 2 3 4 5 6 7
<constante hexadecimal> ->
     0x <dígito hexadecimal> |
     0X <dígito hexadecimal> |
     <constante hexadecimal> <dígito hexadecimal>
<dígito hexadecimal> -> uno de
     0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
<sufijo entero> ->
     <sufijo "unsigned"> <sufijo "long">? |
     <sufijo "long"> <sufijo "unsigned">?
<sufijo "unsigned"> -> uno de
     u U
<sufijo "long"> -> uno de
     l L
```

- El tipo de una constante entera depende de su valor y será representada como primero corresponda, según la siguiente lista: **int**, **unsigned int**, **long**, **unsigned long**.

- El lenguaje de "Las constantes enteras en ANSI C" es regular; por lo tanto, podemos describirlo a través de una definición regular como la que figura en el Ejemplo 7 del libro "Autómatas Finitos y Expresiones Regulares", página 113:

```
<sufijo U> = u | U
<sufijo L> = l | L
<sufijo entero> =
     <sufijo U> |
     <sufijo L> |
     <sufijo U> <sufijo L> |
     <sufijo L> <sufijo U>
```

```
<dígito decimal> = [0-9]
<dígito decimal no nulo> = [1-9]
<dígito hexadecimal> = [0-9a-fA-F]
<dígito octal> = [0-7]
<prefijo hexadecimal> = 0x | 0X
<constante decimal> = <dígito decimal no nulo> <dígito decimal>*
<constante hexadecimal> = <prefijo hexadecimal> <dígito hexadecimal>+
<constante octal> = 0 <dígito octal>*
<constante incompleta> =
     <constante decimal> |
     <constante hexadecimal> |
     <constante octal>
<constante entera> = <constante incompleta> <sufijo entero>?
```

Constante Real

```
<constante real> ->
     <constante fracción> <parte exponente>? <sufijo real>? |
     <secuencia dígitos> <parte exponente> <sufijo real>?
<constante fracción> ->
     <secuencia dígitos>? . <secuencia dígitos> |
     <secuencia dígitos> .
<parte exponente> ->
     e <signo>? <secuencia dígitos> |
     E <signo>? <secuencia dígitos>
<signo> -> uno de + -
<secuencia dígitos> ->
     <dígito> |
     <secuencia dígitos> <dígito>
<dígito> -> uno de 0 1 2 3 4 5 6 7 8 9
<sufijo real> -> uno de f F l L
```

- Si no tiene sufijo, la constante real es **double**.

- En inglés esta constante es conocida como <floating-point-constant>, <constante de punto flotante>.

Constante Carácter

```
<constante carácter> ->
     '<carácter-c>' |
     '<secuencia de escape>'
<carácter-c> -> cualquiera excepto
     ' \
<secuencia de escape> ->
     <secuencia de escape simple> |
     <secuencia de escape octal> |
     <secuencia de escape hexadecimal>
<secuencia de escape simple> -> uno de
     \' \" \? \\ \a \b \f \n \r \t \v
<secuencia de escape octal> ->
     \<dígito octal> |
     \<dígito octal> <dígito octal> |
     \<dígito octal> <dígito octal> <dígito octal>
<dígito octal> -> uno de
     0 1 2 3 4 5 6 7
```

```
<secuencia de escape hexadecimal> ->
    \x<dígito hexadecimal> |
    \x <dígito hexadecimal> <dígito hexadecimal>
\x admite únicamente la x minúscula.
<dígito hexadecimal> -> uno de
    0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
```

Constante Enumeración

```
<constante enumeración> ->
    <identificador>
```

1.4.5. Constantes Cadena

```
<constante cadena> ->
    "<secuencia caracteres-s>"
<secuencia caracteres-s> ->
    <carácter-s> |
    <secuencia caracteres-s> <carácter-s>
<carácter-s> ->
    cualquiera excepto " \ |
    <secuencia de escape>
<secuencia de escape> ->
    <secuencia de escape simple> |
    <secuencia de escape octal> |
    <secuencia de escape hexadecimal>
<secuencia de escape simple> -> uno de
    \' \" \? \\ \a \b \f \n \r \t \v
<secuencia de escape octal> ->
    \ <dígito octal> |
    \ <dígito octal> <dígito octal> |
    \ <dígito octal> <dígito octal> <dígito octal>
<dígito octal> -> uno de
    0 1 2 3 4 5 6 7
<secuencia de escape hexadecimal> ->
    \x <dígito hexadecimal> |
    \x <dígito hexadecimal> <dígito hexadecimal
\x admite únicamente la x minúscula.
<dígito hexadecimal> -> uno de
    0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
```

- En inglés, estas constantes son referidas como <string literals>, <literales de cadena>.
- Notar que no están agrupadas con el resto de las constantes.

1.4.6. Punctuators – Caracteres de Puntuación

```
punctuator -> uno de
    [ ] ( ) { } . ->
    ++ -- & * + - ~ !
    / % << >> < > <= >= == != ^ | && ||
    = *= /= %= += -= <<= >>= &= ^= |=
    ? : ; ... , # ##
```

- La mayoría cumple el papel de operador, ver sección "*Precedencia y Asociatividad de los 45 Operadores*".

1.4.7. Nombre de Encabezados

```
<nombre de encabezado> ->
     < <secuencia de caracteres h> > |
     " <secuencia de caracteres q> "
<secuencia de caracteres h> ->
     <carácter h> |
     <secuencia de caracteres h> <carácter h>
<carácter h> ->
     cualquier miembro del conjunto de caracteres fuente excepto el carácter nueva-
     línea y el carácter >
<secuencia de caracteres q> ->
     <carácter q> |
     <secuencia de caracteres q> <carácter q>
<carácter q> ->
     cualquier miembro del conjunto de caracteres fuente excepto el carácter nueva-
     línea y el carácter "
```

1.4.8. Números de Preprocesador

```
<número de preprocesador> ->
     <dígito> |
     . <dígito> |
     <número de preprocesador> <dígito> |
     <número de preprocesador>  <identificador no dígito> |
     <número de preprocesador> e <sign> |
     <número de preprocesador> E <sign>|
     <número de preprocesador> p <sign>|
     <número de preprocesador> P <sign>|
     <número de preprocesador> .
```

1.5. Gramática de Estructura de Frases

1.5.1. Expresiones

```
<expresión> ->
     <expresión de asignación> |
     <expresión> , <expresión de asignación>
<expresión de asignación> ->
     <expresión condicional> |
     <expresión unaria> <operador asignación> <expresión de asignación>
<expresión condicional> ->
     <expresión O lógico> |
     <expresión O lógico> ? <expresión> : <expresión condicional>
<operador asignación> -> uno de
     = *= /= %= += -= <<= >>= &= ^= |=
<expresión O lógico> ->
     <expresión Y lógico> |
     <expresión O lógico> || <expresión Y lógico>
<expresión Y lógico> ->
     <expresión O inclusivo> |
     <expresión Y lógico> && <expresión O inclusivo>
<expresión O inclusivo> ->
     <expresión O excluyente> |
```

```
        <expresión O inclusivo> | <expresión O excluyente>
    <expresión O excluyente> ->
        <expresión Y> |
        <expresión O excluyente> ^ <expresión Y>
    <expresión Y> ->
        <expresión de igualdad> |
        <expresión Y> & <expresión de igualdad>
    <expresión de igualdad> ->
        <expresión relacional> |
        <expresión de igualdad> == <expresión relacional> |
        <expresión de igualdad> != <expresión relacional>
    <expresión relacional> ->
        <expresión de corrimiento> |
        <expresión relacional> < <expresión de corrimiento> |
        <expresión relacional> > <expresión de corrimiento> |
        <expresión relacional> <= <expresión de corrimiento> |
        <expresión relacional> >= <expresión de corrimiento>
    <expresión de corrimiento> ->
        <expresión aditiva> |
        <expresión de corrimiento> << <expresión aditiva> |
        <expresión de corrimiento> >> <expresión aditiva>
    <expresión aditiva> ->
        <expresión multiplicativa> |
        <expresión aditiva> + <expresión multiplicativa> |
        <expresión aditiva> - <expresión multiplicativa>
    <expresión multiplicativa> ->
        <expresión de conversión> |
        <expresión multiplicativa> * <expresión de conversión> |
        <expresión multiplicativa> / <expresión de conversión> |
        <expresión multiplicativa> % <expresión de conversión>
    <expresión de conversión> ->
        <expresión unaria> |
        (<nombre de tipo>) <expresión de conversión>
    <expresión unaria> ->
        <expresión sufijo> |
        ++ <expresión unaria> |
        -- <expresión unaria> |
        <operador unario> <expresión de conversión> |
        sizeof <expresión unaria> |
        sizeof (<nombre de tipo>)
    <nombre de tipo> está descripto más adelante, en la sección Declaraciones.
    <operador unario> -> uno de & * + - ~ !
    <expresión sufijo> ->
        <expresión primaria> |
        <expresión sufijo> [<expresión>] | /* arreglo */
        <expresión sufijo> (<lista de argumentos>?) | /* invocación */
        <expresión sufijo> . <identificador> |
        <expresión sufijo> -> <identificador> |
        <expresión sufijo> ++ |
        <expresión sufijo> --
    <lista de argumentos> ->
        <expresión de asignación> |
        <lista de argumentos> , <expresión de asignación>
    <expresión primaria> ->
        <identificador> |
        <constante> |
        <constante cadena> |
        (<expresión>)
```

Expresiones Constantes

```
<expresión constante> -> <expresión condicional>
```

- Las expresiones constantes pueden ser evaluadas durante la traducción en lugar de durante la ejecución.

Precedencia y Asociatividad de los 45 Operadores

ID Asociatividad Izquierda-Derecha, DI Asociatividad Derecha-Izquierda

1	ID	operadores de acceso	`( )`		invocación a función
			`[ ]`		subíndice de arreglo
			`.`		acceso a struct y a union
			`->`		acceso a struct y a union
2	DI	operadores unarios (operan	`+`	`-`	signos positivo y negativo
		sobre un solo operando)	`~`		complemento por bit
			`!`		NOT lógico
			`&`		dirección de
			`*`		"indirección"
			`++`		pre-incremento
			`--`		pre-decremento
			`sizeof`		tamaño de
			(*tipo*)		conversión explícita
3	ID	operadores multiplicativos	`*`		multiplicación
			`/`		cociente
			`%`		módulo o resto
4	ID	operadores aditivos	`+`	`-`	suma y resta
5	ID	operadores de desplazamiento	`<<`		desplazamiento de bits a izquierda
			`>>`		desplazamiento de bits a derecha
6	ID	operadores relacionales	`<`	`>`	
			`<=`	`>=`	
7	ID	operadores de igualdad	`==`	`!=`	igual a y distinto de
8	ID	operadores binarios por bit	`&`		AND
9	ID		`^`		OR exclusivo
10	ID		`|`		OR
11	ID	operadores binarios lógicos	`&&`		AND
12	ID		`||`		OR
13	DI	operador condicional	`? :`		(único que opera sobre 3 operandos)
14	DI	operadores de asignación	`=`		
			`*=`	`/=`	`%=` `+=` `–=`
			`<<=`	`>>=`	`&=` `^=` `|=`
15	ID	operador concatenación expresiones	`,`		"coma"

- Los operadores **`&&`** , **`||`** y **`,`** "coma" son los únicos que garantizan que los operandos sean evaluados en un orden determinado (de izquierda a derecha).

- El operador condicional (**`? :`**) evalúa solo un operando, entre el 2do. y el 3ro., según corresponda.

1.5.2. Declaraciones

- Una declaración especifica la interpretación y los atributos de un conjunto de identificadores.

- Si una declaración provoca reserva de memoria, se la llama *definición*.

```
<declaración> ->
     <especificadores de declaración> <lista de declaradores>?
<especificadores de declaración> ->
     <especificador de clase de almacenamiento> <especificadores de declaración>? |
     <especificador de tipo> <especificadores de declaración>? |
     <calificador de tipo> <especificadores de declaración>?
<lista de declaradores> ->
     <declarador> |
     <lista de declaradores> , <declarador>
<declarador> ->
     <decla> |
     <decla> = <inicializador>
<inicializador> ->
     <expresión de asignación> | /* Inicialización de tipos escalares */
     {<lista de inicializadores>} | /* Inicialización de tipos estructurados */
     {<lista de inicializadores> , }
<lista de inicializadores> ->
     <inicializador> |
     <lista de inicializadores> , <inicializador>
<especificador de clase de almacenamiento> -> uno de
     typedef static auto register extern
```

- No más de un especificador de clase de almacenamiento> puede haber en una declaración

```
<especificador de tipo> -> uno de
     void char short int long float double signed unsigned
     <especificador de "struct" o "union">
     <especificador de "enum">
     <nombre de "typedef">
<calificador de tipo> -> const | volatile
<especificador de "struct" o "union"> ->
     <"struct" o "union"> <identificador>? {<lista de declaraciones "struct">} |
     <"struct" o "union"> <identificador>
<"struct" o "union"> -> struct | union
<lista de declaraciones "struct"> ->
     <declaración "struct"> |
     <lista de declaraciones "struct"> <declaración "struct">
<declaración "struct"> ->
     <lista de calificadores> <declaradores "struct"> ;
<lista de calificadores> ->
     <especificador de tipo> <lista de calificadores>? |
     <calificador de tipo> <lista de calificadores>?
<declaradores "struct"> ->
     <decla "struct"> |
     <declaradores "struct"> , <decla "struct">
<decla "struct"> ->
     <decla> |
     <decla>? : <expresión constante>
<decla> -> <puntero>? <declarador directo>
```

```
<puntero> ->
     * <lista calificadores tipos>? |
     * <lista calificadores tipos>? <puntero>
<lista calificadores tipos> ->
     <calificador de tipo> |
     <lista calificadores tipos> <calificador de tipo>
<declarador directo> ->
     <identificador> |
     ( <decla> ) |
     <declarador directo> [ <expresión constante>? ] |
     <declarador directo> ( <lista tipos parámetros> ) /* Declarador nuevo estilo */
     <declarador directo> ( <lista de identificadores>? ) /* Declarador estilo
     obsoleto */
<lista tipos parámetros> ->
     <lista de parámetros> |
     <lista de parámetros> , . . .
<lista de parámetros> ->
     <declaración de parámetro> |
     <lista de parámetros> , <declaración de parámetro>
<declaración de parámetro> ->
     <especificadores de declaración> <decla> | /* Parámetros "nombrados" */
     <especificadores de declaración> <declarador abstracto>? /* Parámetros
     "anónimos" */
<lista de identificadores> ->
     <identificador> |
     <lista de identificadores> , <identificador>
<especificador de "enum"> ->
     enum <identificador>? { <lista de enumeradores> } |
     enum <identificador>
<lista de enumeradores> ->
     <enumerador> | <lista de enumeradores> , <enumerador>
<enumerador> ->
     <constante de enumeración> |
     <constante de enumeración> = <expresión constante>
<constante de enumeración> -> <identificador>
<nombre de "typedef"> -> <identificador>
<nombre de tipo> -> <lista de calificadores> <declarador abstracto>?
<declarador abstracto> ->
     <puntero> |
     <puntero>? <declarador abstracto directo>
<declarador abstracto directo> ->
     ( <declarador abstracto> ) |
     <declarador abstracto directo>? [ <expresión constante>? ] |
     <declarador abstracto directo>? ( <lista tipos parámetros>? )
```

- Ejemplos de <nombre de tipo>:

```
int * [3] /* vector de 3 punteros a int */
int (*) [3] /* puntero a un vector de 3 ints */
int (*) ( void ) /* puntero a una función sin parámetros y que retorna un
     int */
int ( *[ ] ) ( unsigned, . . . ) /* vector de un número no especificado de
     punteros a funciones, cada una de las cuales tiene un  parámetro
     unsigned int más un número no especificado de otros parámetros, y
     retorna un int */
```

1.5.3. Sentencias

```
<sentencia> ->
     <sentencia expresión> |
     <sentencia compuesta> |
     <sentencia de selección> |
     <sentencia de iteración> |
     <sentencia etiquetada> |
     <sentencia de salto>
<sentencia expresión> ->
     <expresión>? ;
<sentencia compuesta> ->
     {<lista de declaraciones>? <lista de sentencias>?}
<lista de declaraciones> ->
     <declaración> |
     <lista de declaraciones> <declaración>
<lista de sentencias> ->
     <sentencia> |
     <lista de sentencias> <sentencia>
```

- La sentencia compuesta también se denomina *bloque*.

```
<sentencia de selección> ->
     if (<expresión>) <sentencia> |
     if (<expresión>) <sentencia> else <sentencia> |
     switch (<expresión>) <sentencia>
La expresión e controla un switch debe ser de tipo entero.
<sentencia de iteración> ->
     while (<expresión>) <sentencia> |
     do <sentencia> while (<expresión>) ;  |
     for (<expresión>? ; <expresión>? ; <expresión>?) <sentencia>
<sentencia etiquetada> ->
     case <expresión constante> : <sentencia> |
     default : <sentencia> |
     <identificador> : <sentencia>
Las sentencias case y default se utilizan solo dentro de una sentencia switch.
<sentencia de salto> ->
     continue ; |
     break ; |
     return <expresión>? ; |
     goto <identificador> ;
```

- La sentencia **continue** solo debe aparecer dentro del cuerpo de un ciclo. La sentencia **break** solo debe aparecer dentro de un **switch** o en el cuerpo de un ciclo. La sentencia **return** con una expresión no puede aparecer en una función **void**.

1.5.4. Definiciones Externas

```
<unidad de traducción> ->
     <declaración externa> |
     <unidad de traducción> <declaración externa>
<declaración externa> ->
     <definición de función> |
     <declaración>
```

- La unidad de texto de programa luego del preprocesamiento es una *unidad de traducción*, la cual consiste en una secuencia de declaraciones externas.

- Las *declaraciones externas* son llamadas así porque aparece fuera de cualquier función. Los términos *alcance de archivo* y *alcance externo* son sinónimos.

- Si la declaración de un identificador para un *objeto* tiene *alcance de archivo* y un *inicializador*, la declaración es una definición externa para el identificador.

```
<definición de función> ->
     <especificadores de declaración>? <decla> <lista de declaraciones>? <sentencia
     compuesta>
```

1.6. Gramática del Preprocesador

```
<archivo de preprocesamiento> ->
     <grupo>?
<grupo> ->
     <parte de grupo> |
     <grupo parte de grupo>
<parte de grupo> ->
     <sección if> |
     <línea de control> |
     <línea de texto> |
     # <no directiva>
<sección if> ->
     <grupo if> <grupos elif>? <grupo else>? <línea endif>
<grupo if> ->
     # if <expresión constante> <nueva línea> <grupo>? |
     # ifdef <identificador> <nueva línea> <grupo>? |
     # ifndef <identificador> <nueva línea>< grupo>?
<grupos elif> ->
     <grupo elif> |
     <grupos elif> <grupo elif>
<grupo elif> ->
     # elif <expresión constante> <nueva línea> <grupo>?
<grupo else> ->
     # else <nueva línea> grupo>?
<línea endif> ->
     # endif <nueva línea>
<línea de control> ->
     # include <tokens pp> <nueva línea> |
     # define <identificador> <lista de reemplazos> <nueva línea> |
     # define <identificador> <parizq> <lista de identificadores>? ) <lista de
     reemplazos> <nueva línea> |
     # define <identificador> <parizq> ... ) <lista de reemplazos> <nueva línea> |
     # define <identificador> <parizq> <lista de identificadores> , ... ) <lista de
     reemplazos> <nueva línea> |
     # undef <identificador> <nueva línea> |
     # line <tokens pp> <nueva línea> |
     # error <tokens pp>? <nueva línea> |
     # pragma <tokens pp> <nueva línea> |
     # <nueva línea>
<línea de texto> ->
     <tokens pp>? <nueva línea>
<no directiva> ->
     <tokens pp> <nueva línea>
<parizq> ->
     un carácter ( no inmediatamente precedido por un espacio blanco
```

```
<lista de reemplazos> ->
     <tokens pp>?
<tokens pp> ->
     <token de preprocesamiento> |
     <tokens pp> <token de preprocesamiento>
<nueva línea> -> el carácter nueva línea
```

Las expresiones constantes de **if** y **elif** pueden estar formadas por los operadores comunes y/o por los siguientes operadores de preprocesamiento:

```
defined ( <identificador> )
defined <identificador>
#
##
```

2. Biblioteca

A continuación se presentan los componentes más importantes y utilizados de la biblioteca estándar agrupados por funcionalidad y archivo *header* (encabezado). Las clases de componentes son *tipos*, *macros* y *funciones*. De las últimas, se indica su *prototipo*, sinopsis y valor de retorno según el estado de terminación de la misma.

El *estilo de codificación* utilizado para los tipos punteros concatena el tipo base junto con el carácter *****. Las declaraciones

```
const char *s;
const char* s;
```

son ambas sintácticamente correctas y equivalentes, la segunda es la utilizada en este texto.

Definición de Algunos Conceptos

Bit

Unidad de almacenamiento de datos, del *ambiente de ejecución*, suficientemente grande para guardar un objeto que puede tener uno de dos valores.

Byte

Unidad *direccionable* de almacenamiento de datos suficientemente grande para almacenar cualquier miembro del *conjunto básico de caracteres* del ambiente de ejecución. Es posible expresar unívocamente la dirección de cada byte individual de un objeto. Un byte está compuesto por un secuencia contigua de bits.

Carácter – General

Miembro de un conjunto de elementos usado para la organización, control o representación de datos.

Carácter – C

Representación de bits que entra en un byte.

Flujo ó Corriente (Stream)

Las *entradas* y *salidas*, ya sea desde o hacia *dispositivos físicos* (e.g. terminales) o *archivos* en dispositivos de almacenamiento estructurado, se corresponden con *flujos lógicos de datos*. Cada flujo esta asociado a un *archivo externo*, el cual puede tener varios flujos asociados. Existen dos tipos de flujos: *flujos de texto* y *flujos binarios*.

Flujos de Texto

Secuencia ordenada de caracteres organizados de a *líneas*. Cada línea está compuesta por cero o más caracteres, más el carácter terminador *nueva línea*; la implementación define si la última línea posee este terminador o no. No es necesario que haya una correspondencia uno a uno entre los caracteres del flujo y los del archivo externo; pero los caracteres imprimibles (función `isprint`), el carácter *tabulado*

horizontal y el carácter *nueva línea* escritos al flujo previamente deben ser iguales a los leídos desde el flujo posteriormente.

Flujos Binarios

Secuencia ordenada de caracteres que puede registrar los datos transparentemente. Los caracteres en el flujo son los mismos que en el archivo externo. Bajo una misma implementación, los caracteres escritos al flujo previamente deben ser iguales a los leídos desde el flujo posteriormente.

Objeto

Región, en el ambiente de ejecución, para el almacenamiento de datos. Los contenidos de dicha región pueden representar valores.

Valor

Significado preciso de los contenidos de un objeto cuando estos son interpretados como poseedores de un tipo específico.

2.1. Definiciones Comunes <stddef.h>

Define, entre otros elementos, el tipo `size_t` y la macro `NULL`. Ambas son definidas, también en otros encabezados, como en `<stdio.h>`.

`size_t`
Tipo entero sin signo que retorna el operador `sizeof`. Generalmente `unsigned int`.

`NULL`
Macro que se expande a un puntero nulo, definido por la implementación. Generalmente el entero cero `0` ó `0L`, ó la expresión constante `(void*)0` .

2.2. Manejo de Caracteres <ctype.h>

`int isalnum (int);`
Determina si el carácter dado `isalpha` o `isdigit` Retorna (ok ? ≠0 : 0).

`int isalpha (int);`
Determina si el carácter dado es una letra (entre las 26 minúsculas y mayúsculas del alfabeto inglés). Retorna (ok ? ≠0 : 0).

`int isdigit (int);`
Determina si el carácter dado es un dígito decimal (entre '0' y '9'). Retorna (ok ? ≠0 : 0).

`int islower (int);`
Determina si el carácter dado es una letra minúscula (entre las 26 del alfabeto inglés). Retorna (ok ? ≠0 : 0)

`int isprint (int);`
Determina si el carácter dado es imprimible, incluye al espacio. Retorna (ok ? ≠0 : 0)

`int isspace (int);`
Determina si el carácter dado es alguno de estos: espacio (`' '`), `'\n'`, `'\t'`, `'\r'`, `'\f'`, `'\v'`. Retorna (ok ? ≠0 : 0)

`int isupper (int);`
Determina si el carácter dado es una letra mayúscula (entre las 26 del alfabeto inglés). Retorna (ok ? ≠0 : 0)

`int isxdigit (int);`
Determina si el carácter dado es un dígito hexadecimal (`'0'`..`'9'`, `'a'`..`'f'` o `'A'`..`'F'`). Retorna (ok ? ≠0 : 0)

`int tolower (int c);`
Si `c` es una letra mayúscula (entre las 26 del alfabeto inglés), la convierte a minúscula. Retorna (mayúscula ? minúscula : `c`)

`int toupper (int c);`
Si `c` es una letra minúscula (entre las 26 del alfabeto inglés), la convierte a mayúscula. Retorna (minúscula ? mayúscula : `c`)

2.3. Manejo de Cadenas <string.h>

Define el tipo `size_t` y la macro `NULL`, ver *Definiciones Comunes*.

`unsigned strlen (const char*);`
Cuenta los caracteres que forman la cadena dada hasta el 1er carácter `'\0'`, excluido. Retorna (longitud de la cadena).

2.3.1. Concatenación

`char* strcat (char* s, const char* t);`
Concatena la cadena `t` a la cadena `s` sobre `s`. Retorna (`s`).

`char* strncat (char* s, const char* t, size_t n);`
Concatena hasta `n` caracteres de `t`, previos al carácter nulo, a la cadena `s`; agrega siempre un `'\0'`. Retorna (`s`).

2.3.2. Copia

`char* strncpy (char* s, const char* t, size_t n);`
Copia hasta `n` caracteres de `t` en `s`; si la longitud de la cadena `t` es < `n`, agrega caracteres nulos en `s` hasta completar `n` caracteres en total; atención: no agrega automáticamente el carácter nulo. Retorna (`s`).

`char* strcpy (char* s, const char* t);`
Copia la cadena `t` en `s` (es la asignación entre cadenas). Retorna (`s`).

2.3.3. Búsqueda y Comparación

`char* strchr (const char* s, int c);`
Ubica la 1ra. aparición de `c` (convertido a `char`) en la cadena `s`; el `'\0'` es considerado como parte de la cadena. Retorna (ok ? puntero al carácter localizado : `NULL`)

`char* strstr (const char* s, const char* t);`
Ubica la 1ra. ocurrencia de la cadena t (excluyendo al '\0') en la cadena s. Retorna (ok ? puntero a la subcadena localizada : `NULL`).

`int strcmp (const char*, const char*);`
Compara "lexicográficamente" ambas cadenas. Retorna (0 si las cadenas son iguales; < 0 si la 1ra. es "menor" que la 2da.; > 0 si la 1ra. es "mayor" que la 2da.)

`int strncmp (const char* s, const char* t, size_t n);`
Compara hasta `n` caracteres de `s` y de `t`. Retorna (como `strcmp`).

`char* strtok (char*, const char*);`
Separa en "tokens" a la cadena dada como 1er. argumento; altera la cadena original; el 1er. argumento es la cadena que contiene a los "tokens"; el 2do. argumento es una cadena con caracteres separadores de "tokens". Retorna (ok ? puntero al 1er. carácter del "token" detectado : `NULL`).

2.3.4. Manejo de Memoria

`void* memchr(const void* s, int c, size_t n);`
Localiza la primer ocurrencia de `c` (convertido a un `unsigned char`) en los `n` iniciales caracteres (cada uno interpretado como `unsigned char`) del objeto apuntado por `s`. Retorna (ok ? puntero al carácter localizado : `NULL`).

`int memcmp (const void* p, const void* q, unsigned n);`
Compara los primeros `n` bytes del objeto apuntado por `p` con los del objeto apuntado por `q`. Retorna (0 si son iguales; < 0 si el 1ero. es "menor" que el 2do.; > 0 si el 1ero. es "mayor" que el 2do.)

`void* memcpy (void* p, const void* q, unsigned n);`
Copia `n` bytes del objeto apuntado por `q` en el objeto apuntado por `p`; si la copia tiene lugar entre objetos que se superponen, el resultado es indefinido. Retorna (`p`).

`void* memmove (void* p, const void* q, unsigned n);`
Igual que `memcpy`, pero actúa correctamente si los objetos se superponen. Retorna (`p`).

`void* memset (void* p, int c, unsigned n);`
Inicializa los primeros `n` bytes del objeto apuntado por `p` con el valor de `c` (convertido a `unsigned char`). Retorna (`p`).

2.4. Utilidades Generales <stdlib.h>

2.4.1. Tips y Macros

`size_t`
`NULL`
Ver *Definiciones Comunes*.

`EXIT_FAILURE`
`EXIT_SUCCESS`
Macros que se expanden a expresiones constantes enteras que pueden ser utilizadas como argumentos de `exit` ó valores de retorno de `main` para retornar al entorno de ejecución un estado de terminación no exitosa o exitosa, respectivamente.

RAND_MAX
Macro que se expande a una expresión constante entera que es el máximo valor retornado por la función **rand**, como mínimo su valor debe ser 32767.

2.4.2. Conversión

```
double atof (const char*);
```
Convierte una cadena que representa un real **double** a número **double**. Retorna (número obtenido, no necesariamente correcto).

```
int atoi (const char*);
```
Convierte una cadena que representa un entero **int** a número **int**. Retorna (número obtenido, no necesariamente correcto) .

```
long atol (const char*);
```
Convierte una cadena que representa un entero **long** a número **long**. Retorna (número obtenido, no necesariamente correcto).

```
double strtod (const char* p, char** end);
```
Convierte como **atof** y, si el 2do. argumento no es **NULL**, un puntero al primer carácter no convertible es colocado en el objeto apuntado por **end**. Retorna como **atof**.

```
long strtol (const char* p, char** end, int base);
```
Similar a **atol** pero para cualquier base entre 2 y 36; si la base es 0, admite la representación decimal, hexadecimal u octal; ver **strtod** por el parámetro **end**. Retorna como **atol**.

```
unsigned long strtoul (const char* p, char** end, int base);
```
Igual que **strtol** pero convierte a **unsigned long**. Retorna como **atol**, pero **unsigned long**.

2.4.3. Administración de Memoria

```
void* malloc (size_t t);
```
Reserva espacio en memoria para almacenar un objeto de tamaño **t**. Retorna (ok ? puntero al espacio reservado : **NULL**)

```
void* calloc (size_t n, size_t t);
```
Reserva espacio en memoria para almacenar un objeto de **n** elementos, cada uno de tamaño **t**. El espacio es inicializado con todos sus bits en cero. Retorna (ok ? puntero al espacio reservado : **NULL**)

```
void free (void* p);
```
Libera el espacio de memoria apuntado por **p**. No retorna valor.

```
void* realloc (void* p, size_t t);
```
Reubica el objeto apuntado por **p** en un nuevo espacio de memoria de tamaño **t** bytes. Retorna (ok ? puntero al posible nuevo espacio : **NULL**).

2.4.4. Números Pseudo-Aleatorios

```
int rand (void);
```
Determina un entero pseudo-aleatorio entre 0 y **RAND_MAX**. Retorna (entero pseudo-aleatorio).

```
void srand (unsigned x);
```
Inicia una secuencia de números pseudo-aleatorios, utilizando a **x** como semilla. No retorna valor.

2.4.5. Comunicación con el Entorno

```
void exit (int estado);
```

Produce una terminación normal del programa. Todos los flujos con *buffers* con datos no escritos son escritos, y todos los flujos asociados a archivos son cerrados. Si el valor de **estado** es **EXIT_SUCCESS** se informa al ambiente de ejecución que el programa terminó exitosamente, si es **EXIT_FAILURE** se informa lo contrario. Equivalente a la sentencia **return estado**; desde la llamada inicial de **main**. Esta función *no retorna a su función llamante*.

```
void abort (void);
```

Produce una terminación anormal del programa. Se informa al ambiente de ejecución que se produjo una terminación no exitosa. Esta función *no retorna a su función llamante*.

```
int system (const char* lineadecomando);
```

Si **lineadecomando** es **NULL**, informa si el sistema posee un procesador de comandos. Si **lineadecomando** no es **NULL**, se lo pasa al procesador de comandos para que lo ejecute. Retorna (**lineacomando** ? valor definido por la implementación, generalmente el nivel de error del programa ejecutado : (sistema posee procesador de comandos ? ≠0 : 0)).

2.4.6. Búsqueda y Ordenamiento

```
void* bsearch (
     const void* k,
     const void* b,
     unsigned n,
     unsigned t,
     int (*fc) (const void*, const void*)
);
```

Realiza una búsqueda binaria del objeto ***k** en un arreglo apuntado por **b**, de **n** elementos, cada uno de tamaño **t** bytes, ordenado ascendentemente. La función de comparación **fc** debe retornar un entero < 0, 0 o > 0 según la ubicación de ***k** con respecto al elemento del arreglo con el cual se compara. Retorna (encontrado ? puntero al objeto : **NULL**).

```
void qsort (
     const void* b,
     unsigned n,
     unsigned t,
     int (*fc) (const void*, const void*)
);
```

Ordena ascendentemente un arreglo apuntado por **b**, de **n** elementos de tamaño **t** cada uno; la función de comparación **fc** debe retornar un entero < 0, 0 o > 0 según su 1er. argumento sea, respectivamente, menor, igual o mayor que el 2do. No retorna valor.

2.5. Entrada / Salida <stdio.h>

2.5.1. Tipos

`size_t`
Ver *Definiciones Comunes*.

`FILE`
Registra toda la información necesitada para controlar un *flujo*, incluyendo su *indicador de posición en el archivo*, puntero asociado a un *buffer* (si se utiliza), un *indicador de error* que registra sin un error de lectura/escritura ha ocurrido, y un *indicador de fin de archivo* que registra si el fin del archivo ha sido alcanzado.

`fpos_t`
Posibilita registrar la información que especifica unívocamente cada posición dentro de un archivo.

2.5.2. Macros

`NULL`
Ver *Definiciones Comunes*.

`EOF`
Expresión constante entera con tipo `int` y valor negativo que es retornada por varias funciones para indicar *fin de archivo*; es decir, no hay mas datos entrantes que puedan ser leídos desde un *flujo*, esta situación puede ser porque se llegó al fin del archivo o porque ocurrió algún error. Contrastar con `feof` y `ferror`.

`SEEK_CUR`
`SEEK_END`
`SEEK_SET`
Argumentos para la función `fseek`.

`stderr`
`stdin`
`stdout`
Expresiones del tipo `FILE*` que apuntan a objetos asociados con los flujos estándar de error, entrada y salida respectivamente.

2.5.3. Operaciones sobre Archivos

`int remove(const char* nombrearchivo);`
Elimina al archivo cuyo nombre es el apuntado por `nombrearchivo`. Retorna (ok ? 0 : ≠0)

`int rename(const char* viejo, const char* nuevo);`
Renombra al archivo cuyo nombre es la cadena apuntada por `viejo` con el nombre dado por la cadena apuntada por `nuevo`. Retorna (ok ? 0 : ≠0).

2.5.4. Acceso

```
FILE* fopen (
     const char* nombrearchivo,
     const char* modo
);
```

Abre el archivo cuyo nombre es la cadena apuntada por **nombrearchivo** asociando un flujo con este según el **modo** de apertura. Retorna (ok ? puntero al objeto que controla el flujo : **NULL**).

```
FILE* freopen(
     const char* nombrearchivo,
     const char* modo,
     FILE* flujo
);
```

Abre el archivo cuyo nombre es la cadena apuntada por **nombrearchivo** y lo asocia con el flujo apuntado por **flujo**. La cadena apuntada por **modo** cumple la misma función que en **fopen**. Uso más común es para el redireccionamiento de **stderr**, **stdin** y **stdout** ya que estos son del tipo **FILE*** pero no necesariamente *lvalues* utilizables junto con **fopen**. Retorna (ok ? flujo : **NULL**).

```
int fflush (FILE* flujo);
```

Escribe todos los datos que aún se encuentran en el buffer del flujo apuntado por **flujo**. Su uso es imprescindible si se mezcla **scanf** con **gets** o **scanf** con **getchar**, si se usan varios **fgets**, etc. Retorna (ok ? 0 : **EOF**).

```
int fclose (FILE* flujo);
```

Vacía el *buffer* del flujo apuntado por **flujo** y cierra el archivo asociado. Retorna (ok ? 0 : **EOF**)

2.5.5. Entrada / Salida Formateada

Flujos en General

```
int fprintf (FILE* f, const char* s, ...);
```

Escritura formateada en un archivo ASCII. Retorna (ok ? cantidad de caracteres escritos : < 0).

```
int fscanf (FILE* f, const char*, ...);
```

Lectura formateada desde un archivo ASCII. Retorna (cantidad de campos almacenados) o retorna (**EOF** si detecta fin de archivo).

Flujos stdin y stdout

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

Lectura formateada desde **stdin**. Retorna (ok ? cantidad de ítems almacenados : **EOF**).

```
int printf (const char*, ...);
```

Escritura formateada sobre **stdout**. Retorna (ok ? cantidad de caracteres transmitidos : < 0).

Cadenas

```
int sprintf (char* s, const char*, ...);
```

Escritura formateada en memoria, construyendo la cadena **s**. Retorna (cantidad de caracteres escritos).

```
int sscanf (const char* s, const char*, ...);
```

Lectura formateada desde una cadena **s**. Retorna (ok ? cantidad de datos almacenados : **EOF**).

2.5.6. Entrada / Salida de a Caracteres

`int fgetc (FILE*); ó`
`int getc (FILE*);`
Lee un carácter (de un archivo ASCII) o un byte (de un archivo binario). Retorna (ok ? carácter/byte leído : **EOF**).

`int getchar (void);`
Lectura por carácter desde **stdin**. Retorna (ok ? próximo carácter del buffer : **EOF**).

`int fputc (int c, FILE* f); ó`
`int putc (int c, FILE* f);`
Escribe un carácter (en un archivo ASCII) o un byte (en un archivo binario). Retorna (ok ? **c** : **EOF**).

`int putchar (int);`
Eescritura por carácter sobre stdout. Retorna (ok ? carácter transmitido : **EOF**).

`int ungetc (int c, FILE* f);`
"Devuelve" el carácter o byte **c** para una próxima lectura. Retorna (ok ? **c** : **EOF**).

2.5.7. Entrada / Salida de a Cadenas

`char* fgets (char* s, int n, FILE* f);`
Lee, desde el flujo apuntado **f**, una secuencia de a lo sumo **n**-1 caracteres y la almacena en el objeto apuntado por **s**. No se leen más caracteres luego del carácter nueva línea o del fin del archivo. Un carácter nulo es escrito inmediatamente después del último carácter almacenado; de esta forma, s queda apuntando a una cadena. Importante su uso con **stdin**. Si leyó correctamente, **s** apunta a los caracteres leídos y retorna **s**. Sí leyó sólo el fin del archivo, el objeto apuntado por **s** no es modificado y retorna **NULL**. Si hubo un error, contenido del objeto es indeterminado y retorna **NULL**. Retorna (ok ? **s** : **NULL**).

`char* gets (char* s);`
Lectura por cadena desde **stdin**; es mejor usar **fgets**() con **stdin** . Retorna (ok ? **s** : **NULL**).

`int fputs (const char* s, FILE* f);`
Escribe la cadena apuntada por **s** en el flujo **f**. Retorna (ok ? último carácter escrito : **EOF**).

`int puts (const char* s);`
Escribe la cadena apuntada por **s** en **stdout**. Retorna (ok ? ≥ 0 : **EOF**).

2.5.8. Entrada / Salida de a Bloques

`unsigned fread (void* p, unsigned t, unsigned n, FILE* f);`
Lee hasta **n** bloques contiguos de **t** bytes cada uno desde el flujo **f** y los almacena en el objeto apuntado por **p**. Retorna (ok ? **n** : < **n**).

`unsigned fwrite (void* p, unsigned t, unsigned n, FILE* f);`
Escribe n bloques de t bytes cada uno, siendo el primero el apuntado por p y los siguientes, sus contiguos, en el flujo apuntado por f. Retorna (ok ? **n** : < **n**).

2.5.9. Posicionamiento

```
int fseek (
     FILE* flujo,
     long desplazamiento,
     int desde
);
```

Ubica el *indicador de posición de archivo* del flujo binario apuntado por **flujo**, **desplazamiento** caracteres a partir de **desde**. **desde** puede ser **SEEK_SET**, **SEEK_CUR** ó **SEEK_END**, comienzo, posición actual y final del archivo respectivamente. Para flujos de texto, **desplazamiento** deber ser cero o un valor retornado por **ftell** y **desde** debe ser **SEEK_SET**. En caso de éxito los efectos de **ungetc** son deshechos, el *indicador de fin de archivo* es desactivado y la próxima operación puede ser de lectura o escritura. Retorna (ok ? 0 : ≠ 0).

```
int fsetpos (FILE* flujo, const fpos_t* posicion);
```

Ubica el *indicador de posición de archivo* (y otros estados) del flujo apuntado por **flujo** según el valor del objeto apuntado por **posicion**, el cual debe ser un valor obtenido por una llamada exitosa a **fgetpos**. En caso de éxito los efectos de **ungetc** son deshechos, el *indicador de fin de archivo* es desactivado y la próxima operación puede ser de lectura o escritura. Retorna (ok ? 0 : ≠ 0).

```
int fgetpos (FILE* flujo, fpos_t* posicion);
```

Almacena el *indicador de posición de archivo* (y otros estados) del flujo apuntado por **flujo** en el objeto apuntado por **posicion**, cuyo valor tiene significado sólo para la función **fsetpos** para el restablecimiento del *indicador de posición de archivo* al momento de la llamada a **fgetpos**. Retorna (ok ? 0 : ≠ 0).

```
long ftell (FILE* flujo);
```

Obtiene el valor actual del *indicador de posición de archivo* para el flujo apuntado por **flujo**. Para flujos binarios es el número de caracteres (bytes ó posición) desde el comienzo del archivo. Para flujos de texto la valor retornado es sólo útil como argumento de **fseek** para reubicar el indicador al momento del llamado a **ftell**. Retorna (ok ? indicador de posición de archivo : –1L).

```
void rewind(FILE *stream);
```

Establece el indicador de posición de archivo del flujo apuntado por **flujo** al principio del archivo. Semánticamente equivalente a **(void)fseek(stream, 0L, SEEK_SET)**, salvo que el indicador de error del flujo es desactivado. No retorna valor.

2.5.10. Manejo de Errores

```
int feof (FILE* flujo);
```

Chequea el *indicador de fin de archivo* del flujo apuntado por **flujo**. Contrastar con la macro **EOF** (Contrastar con la macro **EOF** retornada por algunas funciones). Retorna (*indicador de fin de archivo* activado ? ≠ 0 : 0).

```
int ferror (FILE* flujo);
```

Chequea el *indicador de error* del flujo apuntado por **flujo** (Contrastar con la macro **EOF** retornada por algunas funciones). Retorna (*indicador de error* activado ? ≠ 0 : 0).

```
void clearerr(FILE* flujo);
```

Desactiva los indicadores de fin de archivo y error del flujo apuntado por flujo. No retorna valor.

```
void perror(const char* s);
```

Escribe en el flujo estándar de error (**stderr**) la cadena apuntada por **s**, seguida de dos puntos (**:**), un espacio, un mensaje de error apropiado y por último un carácter nueva línea (**\n**). El mensaje de error está en función a la expresión **errno**. No retorna valor.

2.6. Otros

2.6.1. Hora y Fecha <time.h>

`NULL`
`size_t`
Ver *Definiciones Comunes*.

`time_t`
`clock_t`
Tipos aritméticos capaces de representar el tiempo. Generalmente `long`.

`CLOCKS_PER_SEC`
Macro que expande a una expresión constante de tipo `clock_t` que es el número por segundos del valor retornado por la función `clock`.

`clock_t clock(void);`
Determina el tiempo de procesador utilizado desde un punto relacionado con la invocación del programa. Para conocer el valor en segundos, dividir por `CLOCKS_PER_SEC`. Retorna (ok ? el tiempo transcurrido: `(clock_t)(-1)`.

`char* ctime (time_t* t);`
Convierte el tiempo de `*t` a fecha y hora en una cadena con formato fijo. Ejemplo: Mon Sep 17 04:31:52 1973\n\0. Retorna (cadena con fecha y hora).

`time_t time (time_t* t);`
Determina el tiempo transcurrido en segundos desde la hora 0 de una fecha base; por ejemplo: desde el 1/1/70. Retorna (tiempo transcurrido). Si `t` no es `NULL`, también es asignado a `*t`.

2.6.2. Matemática

`int abs(int i);`
`long int labs(long int i);`
`<stdlib.h>` Calcula el valore del entero `i`. Retorna (valor absoluto de `i`).

`double ceil (double x);`
`<math.h>` Calcula el entero más próximo, no menor que `x`. Retorna (entero calculado, expresado como `double`).

`double floor (double x);`
`<math.h>` Calcula el entero más próximo, no mayor que `x`. Retorna (entero calculado, expresado como `double`).

`double pow (double x, double z);`
`<math.h>` Calcula x^z; hay error de dominio si $x < 0$ y z no es un valor entero, o si x es 0 y $z \neq 0$. Retorna (ok ? x^z : error de dominio o de rango).

`double sqrt (double x);`
`<math.h>` Calcula la raíz cuadrada no negativa de `x`. Retorna ($x \geq 0.0$? raíz cuadrada : error de dominio).

2.7. Los Formatos

2.7.1. Funciones printf, sprintf, fprintf

- Antes de interpretarse la cadena de formatos, todo argumento `float` es convertido a `double`, y todo argumento `char` o `short` es convertido a `int`.

- Si la cantidad de argumentos es menor que la cantidad de formatos, el comportamiento es indefinido. Si la cantidad de argumentos es mayor que la cantidad de formatos, los argumentos sobrantes son evaluados como siempre pero son ignorados. El retorno de la función se produce cuando llega al final de la cadena de formatos.

Cada *especificación de conversión* se realiza mediante la siguiente codificación:

```
% [banderas] [ancho] [precisión] especificador
```

Banderas

Opcional, modifican el significado de la especificación de conversión.

– (ejemplo: %–30s) justifica la conversión a izquierda y rellena con espacios a derecha (si es necesario)

0 (ejemplo: %04x) rellena con ceros a izquierda (después del signo o de un prefijo)

+ (ejemplo: %+5d) si el número convertido es positivo, genera un signo **+** como primer carácter.

<espacio>
(ejemplo: % 5d) un espacio genera un espacio si el número convertido es positivo

(ejemplo: %#x) altera el comportamiento de ciertas conversiones: El especificador **x** produce **0x** como prefijo y el especificador **X** produce el prefijo **0X** Las conversiones para números reales generan un **.** (punto decimal) aún si el número es entero.

Ancho

Opcional, especifica la cantidad mínima de caracteres a ser generados por la conversión.

Es un entero decimal sin signo (ejemplo: %10c).

- Si se escribe un * (ejemplo: %*d), entonces toma al próximo argumento int como valor del ancho; si este valor es negativo, contribuye con una bandera –.

- Si la conversión produce menos caracteres que la cantidad indicada por ancho, habrá relleno; en ausencia de las banderas – o 0, rellena con blancos a izquierda.

Precisión

Opcional, controla la cantidad de caracteres generados por ciertas conversiones.

Se escribe como . (punto) seguido de un entero decimal sin signo.

Un . sólo especifica una precisión cero.

Si se escribe `.*`, el valor del próximo argumento `int` es tomado como precisión; si este valor es negativo, se considera que la precisión es cero.

La precisión (ejemplos: %.10e, %.*s) especifica:

- si se convierte un entero => la cantidad mínima de dígitos a generar
- para los especificadores `e`, `E` o `f` => la cantidad de dígitos a derecha del punto decimal
- para los especificadores `g` o `G` => la cantidad máxima de dígitos significativos a generar
- para el especificador `s` => la cantidad máxima de caracteres a generar

Especificador

Obligatorio, determina cómo interpreta y convierte al correspondiente argumento.

Sea p el valor del campo precisión; entonces:

`c` convierte el argumento `int` a `unsigned char` para generar un carácter

`d` convierte el argumento `int` a una secuencia de al menos `p` dígitos decimales; por omisión, el valor de `p` es 1.

`hd` convierte el argumento `int` a `short` y luego actúa como `d`.

`ld` convierte el argumento `long` igual que `d`.

`i`, `hi`, `li` igual que `d`, `hd`, `ld` respectivamente.

`u` convierte el argumento `unsigned int` y genera una secuencia, sin signo, de un mínimo de `p` dígitos decimales; por omisión, el valor de `p` es 1.

`hu` convierte el argumento `int` a `unsigned short` y luego actúa igual que `u`.

`lu` convierte el argumento `long` a `unsigned long` y luego actúa igual que `u`.

`x`,`X` convierte el argumento `int` a `unsigned int` y luego genera una secuencia, sin signo, de un mínimo de `p` dígitos hexadecimales. Los dígitos hexadecimales con valores `10` a `15` se representan con las letras `a` a `f` o las letras `A` a `F`, respectivamente. Por omisión, el valor de `p` es 1.

`hx,hX` convierte el argumento `int` a `unsigned short` y luego actúa igual que `x` o `X`, según corresponda.

`lx,lX` convierte el argumento `long` a `unsigned long` y luego actúa igual que `x` o `X`, según corresponda.

`o` convierte el argumento `int` a `unsigned int` y luego genera una secuencia sin signo de al menos `p` dígitos octales. Por omisión, el valor de `p` es 1.

`ho` convierte el argumento `int` a `unsigned short` y luego actúa igual que `o`.

`lo` convierte el argumento `long` igual que `o`.

`e,E` convierte el argumento double a una secuencia de la forma `d.ddde±dd` o `d.dddE±dd` (notación punto flotante); ejemplo: `0.123456e+04` o `0.123456E+04`. Cada `d` significa dígito decimal; el exponente está formado por signo y dos dígitos decimales como mínimo; por omisión, la precisión es 6. Si `p` es `0` y no figura la bandera `#`, el punto decimal (.) es omitido.

`Le, LE` convierte el argumento `long double` igual que `e` o `E`, respectivamente.

`F` convierte el argumento `double` a una secuencia de la forma `d.dddddd` (notación punto fijo). Por omisión, la precisión es `6`. Si `p` es `0` y no figura la bandera `#`, el punto decimal (.) es omitido.

`Lf` convierte el argumento `long double` igual que `f`.

`P` convierte el argumento de tipo `void*` a una secuencia de caracteres definida por la implementación, como, por ejemplo, la representación hexadecimal de una dirección de memoria.

`S` genera los caracteres de la cadena apuntada por el argumento, que debe ser de tipo `char*`. Si se especifica una precisión, entonces genera `p` caracteres como máximo.

`%` no hay conversión; genera el carácter `%`.

2.7.2. Funciones scanf, sscanf, fscanf

Cada *especificación de conversión* se realiza mediante la siguiente codificación:

```
% [*] [ancho] especificador
```

*** (Asterisco)** Opcional, indica "supresión de asignación" del campo "scaneado". Ejemplo: %*s indica "saltear" una secuencia de caracteres que no contiene blancos.

Ancho Opcional, especifica la cantidad máxima de caracteres a ser convertidos para su asignación.

Especificador Obligatorio, determina cómo interpreta y convierte al dato que deberá ser almacenado.

`C` almacena un carácter. No "saltea" espacios en blanco.

`D` convierte el entero (base 10) ingresado y lo almacena en una variable `int`.

`Hd` como `d` pero almacena en una variable `short`.

`Ld` como `d` pero almacena en una variable `long`.

`u`,`hu`,`lu`
como `d`, `hd`, `ld` respectivamente pero el entero ingresado es sin signo y es almacenado en una variable `unsigned`.

`o`,`ho`,`lo`
como `d`, `hd`, `ld` respectivamente pero el entero ingresado es interpretado en base 8 y lo almacena en una variable `unsigned`.

`x`(`X`),`hx`(`hX`),`lx`(`lX`)
como `d, hd, ld` respectivamente pero el entero ingresado es interpretado en base 16 y almacenado en una variable `unsigned`.

`i`,`hi`,`li`
igual que `d`, `hd`, `ld` respectivamente, pero interpretan también enteros *octales* (que comienzan con `0`) y enteros *hexadecimales* (que comienzan con `0x` o `0X`).

`e`,`E`,`f`,`g`,`G`
interpreta el dato como "número real" y lo almacena en una variable `float`.

`le`,`lE`,`lf`,`lg`,`lG`
como el especificador `e` pero lo almacena en una variable `double`.

`Le`,`LE`,`Lf`,`Lg`,`LG`
como el especificador `e` pero lo almacena en una variable `long double`.

`S` almacena una secuencia de caracteres (sin blancos) en una zona de memoria que representa un arreglo de `char`. Siempre agrega el carácter `'\0'` como centinela.

3. Implementaciones

3.1. Tipos de Datos Primitivos – Tamaño y Rango

A continuación se muestra el tamaño en bytes y rango de los tipos de datos primitivo para algunas implementaciones comunes. Se indica la arquitectura (Procesador + Sistema Operativo) sobre la cual cada implementación corre, la misma define si se utilizan palabras de dos o cuatro bytes (16 o 32 bits).

		Implementaciones					
Nombre del Tipo	Otros Nombres	**Borland Turbo C++ 3.0** *16 bits*		**Borland C++ 4.02** *32 bits*		**Microsoft Visual Studio .NET** *32 bits*	
		Bytes	Rango	Bytes	Rango	Bytes	Rango
char	signed char	1	-128 .. 127 [-2^7 .. (2^7-1)] (ASCII Standard)				
unsigned char	–	1	0 .. 255 [0 .. (2^8-1)] (ASCII Extendido)				
short	short int, signed short, signed short int	2	–32,768 .. 32,767 [-2^{15} .. $(2^{15}-1)$]				
unsigned short	unsigned short int	2	0 .. 65,535 [0 .. $(2^{16}-1)$]				
int	Signed, signed int	2	–32,768 .. 32,767 [-2^{15} .. $(2^{15}-1)$]	4	–2,147,483,648 .. 2,147,483,647 [-2^{31} .. $(2^{31}-1)$]		
unsigned int	unsigned	2	0 .. 65,535 [0 .. $(2^{16}-1)$]	4	0 .. 4,294,967,295 [0.. $(2^{32}1)$]		
long	long int, signed long, signed long int	4	–2,147,483,648 .. 2,147,483,647				
unsigned long	unsigned long int	4	0 .. 4,294,967,295				
enum	–	*igual a int*					
float	–	4	3.4 x 10^{-38} .. 3.4 x 10^{38} (7 dígitos de precisión)				
double	–	8	1.7 x 10^{-308} .. 1.7 x 10^{308} (15 dígitos)				
long double	–	10	3.4 x 10^{-4932} .. 1.1 x 10^{4932} (19 dígitos)			*igual a double*	

- Borland C++ 4.02 pude correr en arquitecturas de 32 ó 16 bits, en el segundo caso los tamaños y rangos serán idénticos a los de Borland Turbo C++ 3.0.

3.2. Funciones no Estándar

3.2.1. Implementación Borland – Funciones sobre la Consola

`void clreol (void);`
"Borra" la línea a parteir de la posición en que se encuentra el cursor; éste no se desplaza. No retorna valor.

`void clrscr (void);`
"Borra" la pantalla y posiciona el cursor en el extremo superior izquierdo. No retorna valor.

`int getch (void);`
Lectura de carácter sin buffer y sin eco. Retorna (carácter leído).

`int getche (void);`
Lectura de carácter sin buffer pero con eco. Retorna (carácter leído).

`void gotoxy (int x, int y);`
Posiciona el cursor en la columna `x`, fila `y` de la pantalla. El origen es el extremo superior izquierdo, con fila `1` y columna `1`. No retorna valor.

```
/* Programa que utiliza una Maquina de Turing  */
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#define NUMESTADOS 10
#define NUMCOLS 7
#define ACEPTADA 1
#define RECHAZADA 0
#define D 1
#define I 2
int columna(int c);
int verifica(char *s);
int mturing(char *cadena);
/*************************************************************/
/* Elemento de la Matriz de transiciones de la maq. de Turing */
typedef struct {
 int marca;
 int moverA;
 int estadoSiguiente;
} elementoMatriz;

/*****************************************************************/
/* Funcion Principal que lee una cadena de la linea de comandos
   y decide si pertenece al lenguaje, utilizando la maq. de Turing */
int main(int argc, char *argv[]){
  if ( argc == 1 )
  {
   printf("Debe ingresar una Cadena en Linea de Comandos\n");
   return EXIT_FAILURE;
  }

  if ( argc != 2 )
  {
   printf("Cantidad de Argumentos Incorrecta\n");
   return EXIT_FAILURE;
  }

  if ( !verifica(argv[1]) )  {
   printf("Los Caracteres de la Cadena No pertenecen al Alfabeto\n");
   return EXIT_FAILURE;
  }

  if ( mturing(argv[1]) ) printf("La Cadena ingresada pertenece al Lenguaje\n");
  else printf("La Cadena ingresada No pertenece al Lenguaje\n");
  return 0;
}
```

```
/*****************************************/
/* Funcion que implementa la maq. de Turing */

int mturing(char *cadena)
{
 static elementoMatriz tabla[NUMESTADOS][NUMCOLS] =
{      a              b          c          X           Y           Z           fdt
0{{ 'X', D, 1 }, { 0,  0, 9 },{ 0, 0, 9 }, {0, 0, 9 }, {'Y', D, 6 },{ 0, 0, 9 }, { 0, 0, 9 } },

1{{ 'a', D, 1 },{ 'Y', D, 2 },{ 0, 0, 9 }, { 0, 0, 9 },{'Y', D, 4 },{ 0, 0, 9 }, { 0, 0, 9 } },

2{{  0, 0, 9 }, { 'b', D, 2 },{ 'Z', I, 3 },{ 0, 0, 9 },{ 0, 0, 9 }, {'Z', D, 5 }, {0, 0, 9 } },

3{{ 'a', I, 3}, { 'b', I, 3 },{ 0, 0, 9 },  {'X', D, 0},{'Y', I, 3 },{ 'Z', I, 3 },{0, 0, 9 } },

4{{ 0, 0, 9 }, { 'Y', D, 2 }, { 0, 0, 9 }, { 0, 0, 9 }, { 'Y', D, 4 }, { 0, 0, 9 },{0, 0, 9 } },

5{{ 0, 0, 9 }, { 0, 0, 9 }, { 'Z', I, 3 }, { 0, 0, 9 }, { 0, 0, 9 }, { 'Z', D, 5 },{0, 0, 9 } },

6{{ 0, 0, 9 }, { 0, 0, 9 }, { 0, 0, 9 }, { 0, 0, 9 }, { 'Y', D, 6 }, { 'Z', D, 7 },{0, 0, 9 } },

7{{ 0, 0, 9 }, { 0, 0, 9 }, { 0, 0, 9 }, { 0, 0, 9 }, { 0, 0, 9 }, { 'Z', D, 7 }, { 0, I, 8 } },

8+{{ 0, 0, 9 }, { 0, 0, 9 }, { 0, 0, 9 }, { 0, 0, 9 }, { 0, 0, 9 }, { 0, 0, 9 }, { 0, 0, 9 }},

9{{ 0, 0, 9 }, { 0, 0, 9 }, { 0, 0, 9 }, { 0, 0, 9 }, { 0, 0, 9 }, { 0, 0, 9 }, { 0, 0, 9 } } };

 int estadoActual = 0;
 int moverA = D;
 char *punteroCar = cadena;
 int caracter = *punteroCar;
 while ( caracter ) {
  *punteroCar = tabla[estadoActual][columna(caracter)].marca;
  moverA = tabla[estadoActual][columna(caracter)].moverA;
  estadoActual = tabla[estadoActual][columna(caracter)].estadoSiguiente;

  if ( moverA == D ) ++punteroCar;
  if ( moverA == I ) --punteroCar;
  caracter = *punteroCar;
 }
 estadoActual = tabla[estadoActual][columna(caracter)].estadoSiguiente;

 if ( estadoActual == 8 ) return ACEPTADA;
 else return RECHAZADA;
}
/*****************************************************************/
/* Funcion que retorna la columna de la matriz que corresponde al caracter leido */

int columna(int c)
{
 switch ( c )
 {
  case 'a' : return 0;
  case 'b' : return 1;
  case 'c' : return 2;
  case 'X' : return 3;
  case 'Y' : return 4;
  case 'Z' : return 5;
  case '\0' : return 6;
 }
}

/*****************************************************************/
/* Funcion que verifica si los caracteres de la cadena pertenecen al alfabeto */

int verifica(char *s)
```

```
{
 size_t n = strlen(s);
 if ( s[n - 1] == '\n' ) s[n - 1] = '\0';

 for ( ; *s; ++s )
  if ( !(*s == 'a' || *s == 'b'
 || *s == 'c' ) ) return 0;

 return 1;
}
```

```
/* Compilador del Lenguaje Micro (Fischer) */

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

#define NUMESTADOS 15
#define NUMCOLS 13
#define TAMLEX 32+1
#define TAMNOM 20+1

/*****************************************************************/
/******************Declaraciones Globales*************************/
FILE * in;
typedef enum
{
 INICIO, FIN, LEER, ESCRIBIR, ID, CONSTANTE, PARENIZQUIERDO, PARENDERECHO, PUNTOYCOMA,
COMA, ASIGNACION, SUMA, RESTA, FDT, ERRORLEXICO
} TOKEN;
typedef struct
{
 char identifi[TAMLEX];
 TOKEN t;   /* t=0, 1, 2, 3 Palabra Reservada, t=ID=4 Identificador */
} RegTS;

RegTS TS[1000] = { {"inicio", INICIO}, {"fin", FIN}, {"leer", LEER}, {"escribir",
ESCRIBIR}, {"$", 99} };

typedef struct
{
 TOKEN clase;
 char nombre[TAMLEX];
 int valor;
} REG_EXPRESION;

char buffer[TAMLEX];
TOKEN tokenActual;
int flagToken = 0;

/*****************************************************************/
/**********************Prototipos de Funciones********************/

TOKEN scanner();
int columna(int c);
int estadoFinal(int e);

void Objetivo(void);
void Programa(void);
void ListaSentencias(void);
void Sentencia(void);
void ListaIdentificadores(void);
void Identificador(REG_EXPRESION * presul);
void ListaExpresiones(void);
void Expresion(REG_EXPRESION * presul);
void Primaria(REG_EXPRESION * presul);
void OperadorAditivo(char * presul);

REG_EXPRESION ProcesarCte(void);
REG_EXPRESION ProcesarId(void);
char * ProcesarOp(void);
void Leer(REG_EXPRESION in);
```

```
void Escribir(REG_EXPRESION out);
REG_EXPRESION GenInfijo(REG_EXPRESION e1, char * op, REG_EXPRESION e2);

void Match(TOKEN t);
TOKEN ProximoToken();
void ErrorLexico();
void ErrorSintactico();
void Generar(char * co, char * a, char * b, char * c);
char * Extraer(REG_EXPRESION * preg);
int Buscar(char * id, RegTS * TS, TOKEN * t);
void Colocar(char * id, RegTS * TS);
void Chequear(char * s);
void Comenzar(void);
void Terminar(void);
void Asignar(REG_EXPRESION izq, REG_EXPRESION der);

/**************************Programa Principal***********************/

int main(int argc, char * argv[])
{
 TOKEN tok;
 char nomArchi[TAMNOM];
 int l;

/************************Se abre el Archivo Fuente*****************/

 if ( argc == 1 )
 {
  printf("Debe ingresar el nombre del archivo fuente (en lenguaje Micro) en la linea de
comandos\n");
  return -1;
 }

 if ( argc != 2 )
 {
  printf("Numero incorrecto de argumentos\n");
  return -1;
 }

 strcpy(nomArchi, argv[1]);
 l = strlen(nomArchi);
 if ( l > TAMNOM )
 {
  printf("Nombre incorrecto del Archivo Fuente\n");
  return -1;
 }

 if ( nomArchi[l-1] != 'm' || nomArchi[l-2] != '.' )
 {
  printf("Nombre incorrecto del Archivo Fuente\n");
  return -1;
 }

 if ( (in = fopen(nomArchi, "r") ) == NULL )
 {
  printf("No se pudo abrir archivo fuente\n");
  return -1;
 }

/************************Inicio Compilacion*************************/
```

```
 Objetivo();

/*************************Se cierra el Archivo Fuente*****************/

 fclose(in);

 return 0;
}

/**************Procedimientos de Analisis Sintactico (PAS) ************/

void Objetivo(void)
{
 /* <objetivo> -> <programa> FDT #terminar */

 Programa();
 Match(FDT);
 Terminar();
}

void Programa(void)
{
 /* <programa> -> #comenzar INICIO <listaSentencias> FIN */

 Comenzar();
 Match(INICIO);
 ListaSentencias();

 Match(FIN);
}

void ListaSentencias(void)
{
 /* <listaSentencias> -> <sentencia> {<sentencia>} */

 Sentencia();

 while ( 1 )
 {
  switch ( ProximoToken() )
  {
   case ID : case LEER : case ESCRIBIR :
   Sentencia();
   break;
   default : return;
  }
 }
}

void Sentencia(void)
{
 TOKEN tok = ProximoToken();
 REG_EXPRESION izq, der;

 switch ( tok )
 {
  case ID : /* <sentencia> -> ID := <expresion> #asignar ; */
   Identificador(&izq);
   Match(ASIGNACION);
   Expresion(&der);
   Asignar(izq, der);
```

```
   Match(PUNTOYCOMA);
   break;
  case LEER :     /* <sentencia> -> LEER ( <listaIdentificadores> ) */
   Match(LEER);
   Match(PARENIZQUIERDO);
   ListaIdentificadores();
   Match(PARENDERECHO);
   Match(PUNTOYCOMA);
   break;
  case ESCRIBIR :/* <sentencia> -> ESCRIBIR ( <listaExpresiones> ) */
   Match(ESCRIBIR);
   Match(PARENIZQUIERDO);
   ListaExpresiones();
   Match(PARENDERECHO);
   Match(PUNTOYCOMA);
   break;
  default : return;
 }
}

void ListaIdentificadores(void)
{
 /* <listaIdentificadores> -> <identificador> #leer_id {COMA <identificador> #leer_id} */

 TOKEN t;
 REG_EXPRESION reg;

 Identificador(&reg);
 Leer(reg);

 for ( t = ProximoToken(); t == COMA; t = ProximoToken() )
 {
  Match(COMA);
  Identificador(&reg);
  Leer(reg);
 }
}

void Identificador(REG_EXPRESION * presul)
{
 /* <identificador> -> ID #procesar_id */

 Match(ID);
 *presul = ProcesarId();
}

void ListaExpresiones(void)
{
 /* <listaExpresiones> -> <expresion> #escribir_exp {COMA <expresion> #escribir_exp} */

 TOKEN t;
 REG_EXPRESION reg;

 Expresion(&reg);
 Escribir(reg);

 for ( t = ProximoToken(); t == COMA; t = ProximoToken() )
 {
  Match(COMA);
  Expresion(&reg);
  Escribir(reg);
```

```
 }
}

void Expresion(REG_EXPRESION * presul)
{
 /* <expresion> -> <primaria> { <operadorAditivo> <primaria> #gen_infijo } */

 REG_EXPRESION operandoIzq, operandoDer;
 char op[TAMLEX];
 TOKEN t;

 Primaria(&operandoIzq);

 for ( t = ProximoToken(); t == SUMA || t == RESTA; t = ProximoToken() )
 {
  OperadorAditivo(op);
  Primaria(&operandoDer);
  operandoIzq = GenInfijo(operandoIzq, op, operandoDer);
 }
 *presul = operandoIzq;
}

void Primaria(REG_EXPRESION * presul)
{
 TOKEN tok = ProximoToken();

 switch ( tok )
 {
  case ID : /* <primaria> -> <identificador> */
   Identificador(presul);
   break;
  case CONSTANTE :      /* <primaria> -> CONSTANTE #procesar_cte */
   Match(CONSTANTE);
   *presul = ProcesarCte();
   break;
  case PARENIZQUIERDO : /* <primaria> -> PARENIZQUIERDO <expresion> PARENDERECHO */
   Match(PARENIZQUIERDO);
   Expresion(presul);
   Match(PARENDERECHO);
   break;
  default : return;
 }
}

void OperadorAditivo(char * presul)
{
 /* <operadorAditivo> -> SUMA #procesar_op | RESTA #procesar_op */

 TOKEN t = ProximoToken();

 if ( t == SUMA || t == RESTA )
 {
  Match(t);
  strcpy(presul, ProcesarOp());
 }
 else
  ErrorSintactico(t);
}

/*****************************************************************/
/********************Rutinas Semanticas*****************************/
```

```
REG_EXPRESION ProcesarCte(void)
{
 /* Convierte cadena que representa numero a numero entero y construye un registro
semantico */

 REG_EXPRESION reg;

 reg.clase = CONSTANTE;
 strcpy(reg.nombre, buffer);
 sscanf(buffer, "%d", &reg.valor);

 return reg;
}

REG_EXPRESION ProcesarId(void)
{
 /* Declara ID y construye el correspondiente registro semantico */

 REG_EXPRESION reg;

 Chequear(buffer);
 reg.clase = ID;
 strcpy(reg.nombre, buffer);

 return reg;
}

char * ProcesarOp(void)
{
 /* Declara OP y construye el correspondiente registro semantico */

 return buffer;
}

void Leer(REG_EXPRESION in)
{
 /* Genera la instruccion para leer */

 Generar("Read", in.nombre, "Entera", "");
}

void Escribir(REG_EXPRESION out)
{
 /* Genera la instruccion para escribir */

 Generar("Write", Extraer(&out), "Entera", "");
}

REG_EXPRESION GenInfijo(REG_EXPRESION e1, char * op, REG_EXPRESION e2)
{
 /* Genera la instruccion para una operacion infija y construye un registro
  semantico con el resultado */

 REG_EXPRESION reg;
 static unsigned int numTemp = 1;
 char cadTemp[TAMLEX] ="Temp&";
 char cadNum[TAMLEX];
 char cadOp[TAMLEX];

 if ( op[0] == '-' ) strcpy(cadOp, "Restar");
```

```
 if ( op[0] == '+' ) strcpy(cadOp, "Sumar");

 sprintf(cadNum, "%d", numTemp);
 numTemp++;
 strcat(cadTemp, cadNum);

 if ( e1.clase == ID) Chequear(Extraer(&e1));
 if ( e2.clase == ID) Chequear(Extraer(&e2));
 Chequear(cadTemp);
 Generar(cadOp, Extraer(&e1), Extraer(&e2), cadTemp);

 strcpy(reg.nombre, cadTemp);

 return reg;
}

/*****************************************************************/
/**************Funciones Auxiliares*******************************/

void Match(TOKEN t)
{
 if ( !(t == ProximoToken()) ) ErrorSintactico();
 flagToken = 0;
}

TOKEN ProximoToken()
{
 if ( !flagToken )
 {
  tokenActual = scanner();
  if ( tokenActual == ERRORLEXICO ) ErrorLexico();
  flagToken = 1;
  if ( tokenActual == ID )
  {
   Buscar(buffer, TS, &tokenActual);
  }
 }

 return tokenActual;
}

void ErrorLexico()
{
 printf("Error Lexico\n");
}

void ErrorSintactico()
{
 printf("Error Sintactico\n");
}

void Generar(char * co, char * a, char * b, char * c)
{
 /* Produce la salida de la instruccion para la MV por stdout */

 printf("%s %s%c%s%c%s\n", co, a, ',', b, ',', c);
}

char * Extraer(REG_EXPRESION * preg)
{
 /* Retorna la cadena del registro semantico */
```

```
 return preg->nombre;
}

int Buscar(char * id, RegTS * TS, TOKEN * t)
{
 /* Determina si un identificador esta en la TS */

 int i = 0;

 while ( strcmp("$", TS[i].identifi) )
 {
  if ( !strcmp(id, TS[i].identifi) )
  {
   *t = TS[i].t;
   return 1;
  }
  i++;
 }
 return 0;
}

void Colocar(char * id, RegTS * TS)
{
 /* Agrega un identificador a la TS */

 int i = 4;

 while ( strcmp("$", TS[i].identifi) ) i++;

 if ( i < 999 )
 {
  strcpy(TS[i].identifi, id );
  TS[i].t = ID;
  strcpy(TS[++i].identifi, "$" );
 }
}

void Chequear(char * s)
{
 /* Si la cadena No esta en la Tabla de Simbolos la agrega,
    y si es el nombre de una variable genera la instruccion */

 TOKEN t;

 if ( !Buscar(s, TS, &t) )
 {
  Colocar(s, TS);
  Generar("Declara", s, "Entera", "");
 }
}

void Comenzar(void)
{
 /* Inicializaciones Semanticas */
}

void Terminar(void)
{
 /* Genera la instruccion para terminar la ejecucion del programa */
```

```
 Generar("Detiene", "", "", "");
}

void Asignar(REG_EXPRESION izq, REG_EXPRESION der)
{
 /* Genera la instruccion para la asignacion */

 Generar("Almacena", Extraer(&der), izq.nombre, "");
}

/*************************Scanner**********************************/
TOKEN scanner(){
 int tabla[NUMESTADOS][NUMCOLS] = {
{  1,  3,  5,  6,  7,  8,  9, 10, 11, 14, 13,  0, 14 },
{  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2 },
{ 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
{  4,  3,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4 },
{ 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
{ 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
{ 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
{ 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
{ 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
{ 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
{ 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
{ 14, 14, 14, 14, 14, 14, 14, 14, 14, 12, 14, 14, 14 },
{ 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
{ 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
{ 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 } };
 int car;
 int col;
 int estado = 0;

 int i = 0;

 do
 {
  car = fgetc(in);
  col = columna(car);
  estado = tabla[estado][col];

  if ( col != 11 )
  {
   buffer[i] = car;
   i++;
  }
 }
 while ( !estadoFinal(estado) && !(estado == 14) );

 buffer[i] = '\0';

 switch ( estado )
 {
  case 2 : if ( col != 11 )
          {
           ungetc(car, in);
           buffer[i-1] = '\0';
          }
          return ID;
  case 4 : if ( col != 11 )
          {
           ungetc(car, in);
```

```
          buffer[i-1] = '\0';
         }
           return CONSTANTE;
  case 5 : return SUMA;
  case 6 : return RESTA;
  case 7 : return PARENIZQUIERDO;
  case 8 : return PARENDERECHO;
  case 9 : return COMA;
  case 10 : return PUNTOYCOMA;
  case 12 : return ASIGNACION;
  case 13 : return FDT;
  case 14 : return ERRORLEXICO;
 }

 return 0;
}

int estadoFinal(int e)
{
 if ( e == 0 || e == 1 || e == 3 || e == 11 || e == 14 ) return 0;
 return 1;
}

int columna(int c)
{
 if ( isalpha(c) ) return 0;
 if ( isdigit(c) ) return 1;
 if ( c == '+' ) return 2;
 if ( c == '-' ) return 3;
 if ( c == '(' ) return 4;
 if ( c == ')' ) return 5;
 if ( c == ',' ) return 6;
 if ( c == ';' ) return 7;
 if ( c == ':' ) return 8;
 if ( c == '=' ) return 9;
 if ( c == EOF ) return 10;
 if ( isspace(c) ) return 11;

 return 12;
}
```

Bibliotecas en ANSI C

Objetivos

- Presentar el **concepto de biblioteca**.
- Presentar los pasos necesarios para la **creación de una biblioteca** con ANSI C.
- Presentar los pasos para la **construcción de biblioteca con BCC32**.
- Presentar los pasos para **compilar** (y linkeditar) **con BCC32 programas fuente que utilizan bibliotecas a parte de la Standarad**.

Los objetivos se cumplen mediante la presentación de un caso de estudio.

Objetivos de la Asignatura que este Artículo Trata

El programa sintético de Estructura de datos y algoritmos indica que los objetivos que la asignatura debe cumplir son "Introducir al alumno en el estudio de las estructuras de datos y el dominio de un lenguaje procedural"

Este paper ayuda a cumplir el objetivo **"dominio de un lenguaje procedural"** y se encuentra dentro de la categoría **"Herramientas Software para la Construcción de Software"**.

Introducción

En este artículo se introduce el concepto de biblioteca y dos casos de estudio que presentan aplicaciones prácticas del concepto. El diseño de la solución para cada caso de estudio se modela con UML.

El proceso para la creación de una biblioteca es genérico y puede ser reproducido en diferentes entornos de desarrollo ANSI C, teniendo en cuentas las características de cada uno. En este artículo se ejemplifica mediante el compilador de línea de comando "***C++ Compiler 5.5 with Command Line Tools***", también conocido como ***BCC32***, disponible en **http://www.borland.com/ y ftp://ftpd.borland.com/download/bcppbuilder/freecommandLinetools.exe**

Conceptos

Bibliotecas

Una biblioteca es una colección de herramientas para el programador. En una biblioteca se encuentran subprogramas (funciones para ANSI C), tipos de datos, constantes, enumeraciones y otros identificadores. Las bibliotecas permiten la modularización, el desacoplamiento y la centralización de funcionalidad común.

Una biblioteca tiene una parte **privada**, la **implementación**, y otra **pública**, la **interfaz**. La biblioteca **encapsula** la implementación, y la interfaz debe estar diseñada para que **oculte información** sobre el diseño de la implementación. ANSI C provee una biblioteca Standard, y varios archivos header que hacen de interfaz. Cada archivo header representa una *agrupación funcional* (e.g. string.h para el manejo de cadenas y stdio.h para la entrada y salida).

Las bibliotecas se utilizan también para el desarrollo de **Tipos de Datos Abstractos** (TADs).

Una biblioteca puede contener la implementación de un TAD y venir acompañada por su interfaz; o bien una misma biblioteca puede contener varios TADs y estar acompañada por diferentes interfaces, una por cada TAD. Las bibliotecas son utilizadas por programa u otras bibliotecas.a

Archivos Header

Los archivos header (encabezados) son archivos con código fuente ANSI C que deben ser compartidas por varios programas. En su gran mayoría, son declaraciones y macros; no contienen *definiciones de funciones*, sino *declaraciones de funciones* (i.e. *prototipos*). Los archivos header, por convención, tienen como extensión **".h"**.

No se debe confundir el concepto de archivo header con biblioteca. Por ejemplo, la biblioteca Standard posee varios archivos header, en los cuales header hay declaraciones para utilizar las funciones que están precompiladas (i.e. código objeto) en la biblioteca Standard.

Funciones Públicas

Una función pública de una biblioteca es una función que puede ser invocada por diferentes programas u otras bibliotecas.

En ANSI C, por defecto, las funciones son públicas; pueden ser invocadas desde otros programas o bibliotecas (i.e. unidades de traducción).

Leer la sección **4.6 Static Variables** del capítulo **4. Functions and Program Structure** de **[K&R1988]**.

Funciones Privadas (static)

Una función privada de una biblioteca es una función que el programa o biblioteca llamante no necesita conocer para poder hacer uso de la biblioteca.

En general son funciones helper ("auxiliares") que ayudan al diseño de la implementación de la biblioteca. Por ejemplo, la función pública **`Alumno *ImprimirAlumno(int legajo)`** puede, internamente, invocar (i.e. hacer uso) de la función privada **`BuscarAlumno`** para abstraer la tarea de la búsqueda del alumno a imprimir.

En ANSI C, las funciones privadas se implementan precediendo el prototipo y la definición de la función con la palabra clave **`static`**. Las funciones static sólo pueden ser invocadas desde la unidad de traducción (i.e. archivo fuente) dónde están definidas, no pueden ser invocadas por otras programas o bibliotecas. El prototipo de una funcione static se encuentran al comienzo del archivo fuente donde está definida, junto a otras declaraciones externas; sus prototipos no se escriben en los archivos header.

Leer la sección **4.6 Static Variables** del capítulo **4. Functions and Program Structure** de **[K&R1988]**.

Directivas al Preprocesador

Permiten dirigir las acciones del preprocesador. Otorgan funcionalidades de compilación condicional y de substitución.

La directiva **`#include`** incorpora el contenido completo del archivo indicado. Existen dos sintaxis, la primera es particular para los *headers Standard* (e.g. **`#include <time.h>`**), y la segunda es para otros *archivos fuente no Standard* (e.g. **`#include "stack.h"`**).

La directiva **`#define`** introduce un identificador, opcionalmente asociándolo con una porción de código fuente.

La expresión **`defined`** `<identificador>` se reemplaza por *cero* o por *uno* en función de si el identificador ya fue definido o no.

Las directivas de preprocesador **#ifndef** <expresión constante> y **#endif** le indican al preprocesador que procese el bloque que encierran si y solo si la expresión constante es *cero* (falsa), permiten la compilación condicional.

Las siguientes líneas son equivalentes

#if ! defined <identificador>

#ifndef <identificador>

La siguiente estructura sintáctica evita que el contenido de un archivo *header* sea procesado más de una vez.

#ifndef <identificador>

#define <identificador>

Líneas de código fuente del archivo header.

#endif

Leer de **[K&R1988]** las siguientes secciones del capítulo **A.12 Preprocessing** · **A.12.3 Macro Definition and Expansion**

· **A.12.4 File Inclusion**

· **A.12.5 Conditional Compilation**

Caso de Estudio – Saludos

Este caso presenta dependencias más complejas entre bibliotecas y un programa. La dependencias están dadas por la relación de invocación y de inclusión. Una de las bibliotecas utiliza una función privada.

El programa hace uso de dos bibliotecas y también de la Biblioteca Standard. Una biblioteca llama a la otra, y la última invoca a una función de la Standard.

La única y simple acción del programa es la emitir por stdin un saludo una determinada cantidad de veces.

Construcción

Archivos Fuentes

Biblioteca Saludar

Saludar.h

```
#ifndef SaludarHeaderIncluded
#define SaludarHeaderIncluded
/* Saludar.h
*/
void Saludar( const char* unNombre );
#endif
```

Saludar.c

```
/* Saludar.c
*/
#include <stdio.h> /* printf */
#include "Saludar.h" /* Saludar */
/* Prototipo de función privada */
static void SaludarEnCastellano( void );
static void SaludarEnCastellano( void ){
printf("Hola");
```

```
return;
}
void Saludar( const char* unNombre ){
SaludarEnCastellano();
printf(", %s\n", unNombre);
return;
}
```

Biblioteca SaludarMuchasVeces

SaludarMuchasVeces.h

```
#ifndef SaludarMuchasVecesHeaderIncluded
#define SaludarMuchasVecesHeaderIncluded
/* SaludarMuchasVeces.h
*/
void SaludarMuchasVeces( const char* unNombre, int cuantasVeces );
#endif
```

SaludarMuchasVeces.c

```
/* SaludarMuchasVeces.c
*/
#include "SaludarMuchasVeces.h" /* SaludarMuchasVeces */
#include "Saludar.h" /* Saludar */
void SaludarMuchasVeces( const char* unNombre, int cuantasVeces ){
int i;
for( i = 0; i < cuantasVeces; i++)
Saludar( unNombre );
return;
}
```

Aplicación

Aplicación.c

```
/* Aplicacion.c */
#include <stdio.h> /* puts */
#include <stdlib.h> /* EXIT_SUCCESS */
#include "Saludar.h" /* Saludar */
#include "SaludarMuchasVeces.h" /* SaludarMuchasVeces */
int main ( void ){
/* de biblioteca Saludar */
Saludar("Mundo");
puts("");
/* de biblioteca Saludar */
/* pero no se puede invocar porque es static */
/* SaludarEnCastellano("Mundo"); */
/* puts(""); */
/* de biblioteca SaludarMuchasVeces */
SaludarMuchasVeces("Mundo", 4);
```

```
puts("");
return EXIT_SUCCESS;
}
```

Conclusión

Las bibliotecas demuestran ser una excelente forma de implementar la **modularización** y para la construcción de **TADs**.

Se debe prestar especial cuidado al diseño de las interfaces que exponen las bibliotecas para cumplir con determinada funcionalidad, se debe aplicar el **ocultamiento de información**. Mediante la **abstracción** se debe evitar que el cliente de la biblioteca necesite conocer los detalles de implementación para hacer uso de la biblioteca. ANSI C permite encapsulamiento de componentes de las bibliotecas mediante la palabra clave **static**.

Bibliografía

K&R1988] "The C Programming Language, 2nd Edition", B. W. Kernighan & D. M. Ritchie, 1988, Prentice-Hall, ISBN 0-13-110362-8

Documentación Borland C++ Compiler 5.5 with Command Line Tools

OMG-Unified Modeling Language, v1.5, http://www.omg.org .—

Normativas y Guías para construcción de TADs

TAD

Tipos de datos definidos por

- Conjunto de valores
- Conjunto de operaciones

Pueden ser

- Fundamentales, básicos, primitivos, predefinidos.
- Derivados
- Definidos por el usuario o TAD.

ABSTRACCION

Capacidad para encapsular y aislar la información, del diseño y ejecución.
Es un proceso mental, mas sencillo que lo que modela para ser útil. El mapa es una abstracción del camino, la palabra león no ruge. Es una herramienta fundamental para tratar la complejidad.

ABSTRACCION EN EL SOFTWARE

- Instrucciones binarias
- Nemotécnicos de los lenguajes ensambladores
- Agrupamiento de instrucciones primitivas formando macroinstrucciones
- Encapsular y aislar en procedimientos, funciones, módulos, TAD y objetos.

TAD

- Lo crea el usuario y los puede manipular como los creados por el sistema.
- Para su creación se utilizan los módulos, para construirlos se debe poder
    - Dar una definición precisa del tipo
    - Hacer disponible de un conjunto de operaciones utilizables para manipular instancias del tipo.
    - Proteger los datos asociados de modo que solo se pueda trabajar con las operaciones definidas.
    - Hacer instancias múltiples del tipo.

OBJETOS

- TAD al que se añaden innovaciones para la reutilización
    - Conceptos
        - Abstracción
        - Encapsulamiento
        - Herencia
        - Polimorfismo
        - Persistencia
    - Entidades básicas
        - Objeto
        - Instancia
        - Clases
        - Mensajes
        - Métodos
        - Propiedades
        - Herencia – Jerarquía

MODULARIZACION

- Parte publica
    - Primitivas de acceso
    - Descripción de las propiedades de los datos
- Parte privada
    - Atributos
    - Representación

ABSTRACCION EN LENGUAJES DE PROGRAMACION

- Abstracción de datos
    - Representación: elección de la estructura del dato
    - Operaciones: Elección del algoritmo

- Abstracción de control
  - Abstracción procedimental
  - Parámetros

VENTAJAS DE LOS TADS

- Permite mejor conceptualización y modelización
- Mejora el rendimiento
- Separa implementación de especificación
- Permite agregados y mejoras sin afectar la parte publica

ESPECIFICACION DE UN TAD

- Informal
  - Detallar los valores del tipo y las operaciones que se pueden vincular con esos valores
  - TAD NombreTipo(valores y descripción)
- Formal
  - Axiomas que describen el compòrtamieno
  - TAD NombreTipo (valores que pueden tomar los datpos)
  - Sintaxis
    - Operación (Tipo Argumento) → Tipo resultado
    - ConjuntoVacio → Conjunto
    - Agregar(Conjunto, Elemento) → Conjunto
    - Pertenece(Conjunto, Elemento) → Booleano
    - ----------
  - Semantica
    - Operación(Valores particulares) → Expresión resultado
    - Union (ConjuntoVacio, ConjuntoVacio) → ConjuntoVacio
    - Union(ConjuntoVacio, Agregar(Conjunto, Elemento) → Agregar (conjunto, Elemento)

E*specificación*

Guía para la estructuración de la especificación de los valores

Descripción, n-upla, descripción de cada miembro de la n-upla, tipo de dato al que pertenece cada miembro de la n-upla, restricciones.

Guía para la estructuración de la especificación de las operaciones

· f : A × B × C → D × E (*Título de la operación*)

· Clasificación de la operación.

· Breve descripción.

· f : M × K × K → S × M (*Refinación de los conjuntos*)

· f (m_1, k_1, k_2) = (s, m_2) (*Identificación de los datos y resultados*)

· Precondiciones y Poscondiciones.

· Dominio e Imagen.

· Semántica descripta en lenguaje matemático con la ayuda de axiomas o de lenguaje natural.

· Ejemplo(s).

Normativas para las implementaciones

Archivos Encabezado

El contenido de los archivos encabezado debe estar ″rodeado″ por las siguientes directivas al preprocesador:

```
#ifndef _INCLUIR_TAD_
```

```
#define _INCLUIR_TAD_
/* Contenido del archivo encabezado. */
#endif
```

Esto permite la compilación condicional, evitando incluir más de una vez el contenido del archivo encabezado. Reemplazar *TAD* por un nombre del TAD que se está incluyendo.

Convención de Nomenclatura para los Identificadores

Estilos de "Capitalización"

PascalCase

Todas las palabras juntas, sin espacios. Todas las palabras comienzan con mayúsculas y siguen enminúsculas.

Ejemplos:

EstaEsUnaFraseEnPascalCase

Pila

AgregarElemento

camelCase

Todas las palabras juntas, sin espacios. Todas las palabras comienzan con mayúsculas y siguen en minúsculas, excepto la primera que está en minúsculas.

Ejemplos:

estaEsUnaFraseEnCamelCase

unaPila

laLongitud

UPPERCASE

Todas las palabras juntas, sin espacios. Todas las palabras en mayúsculas. En general se separan las palabras con undescores.

Ejemplos:

ESTAESUNAFRASEENUPPERCASE

LIMITE_SUPERIOR

MAXIMO

Underscores (Guiones Bajos)

· No usar con camelCase ni con PascalCase,

· Sí usar con UPPER_CASE.

· No usar delante de identificadores.

Identificadores para diferentes elementos

Nombres de tipo (typedef)

En camelCase, sustantivo. SistemaPlanetario, Planeta y NumeroAstronomico.

Funciones

En PascalCase, deben comenzar con un verbo en infinitivo. Ejemplos: OrdenarArreglo(),

SepararUsuarioDeDominio(), Planeta_SetNombre().

Funciones públicas de cada TAD

Los identificadores de las funciones públicas que implementan las operaciones se prefijan con el nombre

del TAD seguido de un *underscore* ("_"). SistemaPlanetario_,.

Variables Locales y Parámetros

En camelCase, sustantivo, en plural para arreglos.

Variables Globales.

En PascalCase, sustantivo, en plural para arreglos.

Enumeraciones.

Su typedef y sus elementos en PascalCase y en singular.

Constantes Simbólicas (#define).

En UPPER_CASE_CON_UNDERSCORES.

Cadenas

Implementadas sin tamaño máximo (malloc y free).

Funciones privadas

Definidas como static. No forman parte de la especificación.

Operaciones de Destrucción

No forman parte de la especificación. Permiten liberar los recursos tomados durante la creación.

Operaciones de Creación

Las operaciones de creación serán implementadas como funciones que retornan un puntero a un objeto del tipo del TAD. Deberán usar malloc para obtener memoria para ese objeto. Si no hay memoria disponible, retornarán NULL. Por ejemplo:

```
NumeroAstronomico *NumeroAstronomico_CrearDesdeCadena(
const char *unaCedena );
```

Valores de los TADs como parámetros

Un valor de un TAD será pasado como parámetro mediante un puntero a ese valor. Si el parámetro es del tipo entrada/salida no tendrá el calificador const, si es de entrada sí.

Por ejemplo:

```
SistemaPlanetario* SistemaPlanetario_AgregarPlaneta(
SistemaPlanetario* unSistemaPlanetario, /*inout*/
const Planeta* unPlaneta /*in*/
);
int NumeroAstronomico_EsOverflow(
const NumeroAstronomico* unNumeroAstronomico /*in*/
);
```

Operaciones de Modificación (Mutación, Setter)

Estas operaciones seguirán el modelo impuesto por funciones como strcat de ANSI C.

La función strcat recibe dos parámetros: Primero, la cadena que será modificada concatenándole al final otra cadena, y segundo, la cadena que será concatenada al final de la primera. La función retorna el puntero a la primera cadena.

```
char *strcat(char *s1, const char *s2);
```

Este modelo permite evitar construcciones del tipo:

```
char s1[4+1]="ab", *s2="cd";
strcat(s1, s2);
puts(s1);
```

al ser reemplazadas por:

```
char s1[4+1]="ab", *s2="cd";
puts( strcat(s1, s2) );
```

Análogamente

```
Planeta* Planeta_SetNombre(Planeta* unPlaneta, const char* unNombre);
```

permite reemplazar:

```
Planeta* unPlaneta=Planeta_Crear(…);
Planeta_SetNombre(unPlaneta, "Krypton");
puts( Planeta_GetNombre(unPlaneta) );
```

por:

```
Planeta* unPlaneta=Planeta_Crear(…);
puts( Planeta_GetNombre( Planeta_SetNombre(unPlaneta, "Krypton") ) );
```

Nombres de archivos

· Cada TAD se implementará en una biblioteca. El código fuente de la implementación estará en un archivo *TAD*.c, la declaración de la parte pública en el archivo encabezado *TAD*.h, y se generará la biblioteca *TAD*.lib.

· El código fuente del programa de aplicación que prueba el TAD será *TADAplicacion*.c, y se generará *TADAplicacion*.exe.

· Sólo se entregarán los códigos fuentes: *TAD*.c, *TAD*.h y *TADAplicacion*.c.

· No se aceptarán *TAD*.lib y *TADAplicacion*.exe.

Nombres de los archivos de los TADs de este TP:

· SistemaPlanetario.c, SistemaPlanetario.h, SistemaPlanetario.lib, SistemaPlanetarioAplicacion.c y SistemaPlanetarioAplicacion.exe

· Planeta.c, Planeta.h, Planeta.lib, PlanetaAplicacion.c y PlanetaAplicacion.exe.

· NumeroAstronomico.c, NumeroAstronomico.h, NumeroAstronomico.lib, NumeroAstronomicoAplicacion.c y NumeroAstronomicoAplicacion.exe.

Guía de secuencia de actividades para la generación de los TADs

A continuación se presenta una guía de una posible secuencia de actividades para la construcción de TADs.

En base a una *correcta especificación*, se diseña un *correcto programa de prueba*, luego se implementa el TAD. Si la implementación realizada pasa el programa de prueba, la *implementación es correcta.*

Se debe considerar que *el proceso es en general iterativo*, en el sentido de *la especificación siempre es la entrada para la implementación* y que debe estar *completamente definida*, pero que *hay veces que la implementación retroalimenta a la especificación para mejorarla* y volver a comenzar el proceso.

1. Comprensión del contexto y del problema que el TAD ayuda a solucionar.
2. Diseño de la Especificación.
3. Diseño de los casos de prueba a nivel especificación.
4. Implementación.
   - 4.1. Diseño de prototipos.
   - 4.2. Codificación de prototipos.
   - 4.3. Diseño programa de prueba (aplicación)
   - 4.4. Diseño y codificación de implementación de valores.
   - 4.5. Codificación de funciones públicas y privadas (static).
   - 4.6. Construcción de biblioteca.
   - 4.7. Ejecución de programa de prueba.
   - 4.8. ¿Hubo algún error? Entonces volver a 4.4

Presentación

Forma

- El trabajo debe presentarse en **hojas A4 abrochadas en la esquina superior izquierda**.
- En el **encabezado de cada hoja** debe figurar el **título del trabajo**, el **título de entrega**, el **código de curso** , **número de equipo** y los **apellidos de los integrantes del equipo**.
- Las hojas deben estar enumeradas en el pie de las mismas con el formato "**Hoja n de m**".
- El **código fuente de cada componente del TP** debe comenzar con un **comentario encabezado**, con todos los datos del equipo de trabajo: **curso**; **legajo**, **apellido y nombre de cada integrante del equipo** y **fecha de última modificación.**
- La **fuente** (estilo de caracteres) a utilizar en la **impresión** de los **códigos fuente** y de las **capturas de las salidas** debe ser una fuente de ***ancho fijo*** (e.g. **`Courier New`**, `Lucida Console`).

El TP tiene **tres secciones imporatnes, una por cada TAD**, a su vez, **cada sección** tiene las siguientes **tres grandes sub-secciones**:

Nombre del TAD.

1. Especificación.

Especificación completa, extensa y sin ambigüedades de los valores y de las operaciones del TAD.

2. Implementación

Biblioteca que implementa el TAD.

2.1. **Listado** de código fuente del **archivo encabezado**, **parte pública**, *TAD*.h.

2.2. **Listado** de código fuente de la **definición de la Biblioteca**, **parte privada**, *TAD*.c. Las funciones privadas deberán ser precedidas por su documentación, un comentario con la documentación de la función indicando, entre otras cosas, propósito de la función, semántica de los parámetros in, out e inout.

1.2.3. **Salidas**. Captura impresa de la salida del **proceso de traducción** (BCC32 y TLIB).

3. Aplicación de Prueba

3.1. **Código Fuente**. Listado del código fuente de la aplicación de prueba, *TAD*Aplicacion.c.

3.2. **Salidas**

3.2.1. Captura impresa de la salida del **proceso de traducción** (BCC32).

3.2.2. Captura impresa de las salidas de la **aplicación de prueba**.

3.2.3. Impresión de **archivos de prueba de entrada** y de **archivos de salida generados durante la prueba**.

· El TP será acompañado por:

A. Copia Digitalizada

CD ó disquette (preferentemente CD) con copia de **solamente los 3 archivos de código fuente de cada TAD** (i.e.: *SistemaPlanetario.c, SistemaPlanetario.h, SistemaPlanetarioAplicacion.c Planeta.c, Planeta.h, PlanetaAplicacion.c, NumeroAstronomico.c, NumeroAstronomico.h y NumeroAstronomicoAplicacion.c*) **No se debe entregar ningún otro archivo**.

B. Formulario de Seguimiento de Equipo.

Tiempo

- Luego de la aprobación del TP se **evaluará individualmente a cada integrante del equipo**.

Normativas de Codificación

Introducción

Este artículo presenta las normas de codificación para ANSI C que se aplican en SSL. El objetivo es facilitar el entendimiento y modificación de programas. Su gran mayoría son aplicables a distintos lenguajes de programación y no se restringen solo a ANSI C.

Indentación (Sangría)

De **[K&R1988] "Chapter 1 – A Tutorial Introduction – 1.2 Variables and Arithmetic Expressions"**

El cuerpo de un `while` puede ser unas o más sentencias incluidas en llaves, como en el convertidor de temperatura, o una sola declaración sin las llaves, como en:

```
while (i < j)
i = 2 * i;
```

En cualquier caso, siempre indentaremos las sentencias controladas por el `while` con un tabulado (que hemos mostrado como cuatro espacios) así podemos ver de un vistazo qué sentencias están dentro del ciclo. La indentación acentúa la estructura lógica del programa

Aunque a los compiladores de C no les interesa la apariencia de un programa, las indentaciones y espaciados correctos son críticos para hacer los programas fáciles de leer. Recomendamos escribir solo una sentencia por línea, y usar espacios en blanco alrededor de operadores para clarificar la agrupación. La posición de las lleves es menos importante, aunque hay personas que mantienen creencias apasionadas sobre la ubicación de las mismas. Hemos elegido uno de varios estilos populares. Escoja un estilo que le quede bien, y luego úselo de manera consistente. >>

```
/* Incorrecto */
unsigned long int Factorial(unsigned int n)
{
if(n<=1)
return 1; return n * Factorial(n-1);}
/* Correcto */
unsigned long int Factorial(unsigned int n){
→ if( n <= 1 )
→ → return 1;
→ return n * Factorial( n – 1 );
}
```

Estilos de Indentación

A continuación se muestran diferentes estilos, la elección es puramente subjetiva, pero su aplicación al largo de un mismo desarrollo debe ser consistente.

Estilo K&R – También conocido como "The One True Brace Style"

El libro **[K&R1988]** usa siempre este estilo, salvo para las definiciones de las funciones, que usa el estilo **BSD/Allman**. Las Secciones principales de **Java** también usan este estilo. Este es el estilo que recomienda y que usa la Cátedra para todas las construcciones.

```
while( SeaVerdad() ) {
      HacerUnaCosa();
      HacerOtraCosa();
}
HacerUnaUltimaCosaMas();
```

Estilo BSD/Allman

Microsoft Visual Studio 2005 impone este estilo por defecto. Nuevas secciones de Java usan este estilo. Es un estilo recomendable.

```
while( SeaVerdad() )
{
      HacerUnaCosa();
      HacerOtraCosa();
}
HacerUnaUltimaCosaMas();
```

Estilo Whitesmiths

```
while( SeaVerdad() )
  {
      HacerUnaCosa();
      HacerOtraCosa();
  }
HacerUnaUltimaCosaMas();
```

Estilo GNU

```
while( SeaVerdad() )
{
      HacerUnaCosa();
      HacerOtraCosa();
}
HacerUnaUltimaCosaMas();
```

Estilo Pico

```
while( SeaVerdad()
{ HacerUnaCosa();
HacerOtraCosa(); }
HacerUnaUltimaCosaMas();
```

Estilo Banner

```
while( SeaVerdad() ) {
HacerUnaCosa();
```

```
HacerOtraCosa();
}
HacerUnaUltimaCosaMas();
```

Formato del Código

Espacios

Utilizar los espacios horizontales y verticales de manera tal que el código quede compacto pero que a la vez permita una fácil lectura.

Tabulado de la Indentación

Utilizar el tabulado para indentar, no utilizar múltiples espacios.

Longitud de las líneas

Formatear el código de manera que las líneas no se corten en medios con solo 70 columnas.

Convención de Nomenclatura para los Identificadores

Separamos el estilo de los identificadores en dos partes: el estilo de "capitalización" y el estilo para identificar los elementos de las distintas categorías (e.g. variables, funciones, tipos de datos).

Estilos de "Capitalización"

En el contexto de los lenguajes de programación *case-sensitive* (i.e. diferencian mayúsculas de minúsculas), como ANSI C, las posibles combinaciones de mayúsculas, minúsculas y *underscores* (guiones bajos) establecen estilos para construir los identificadores.

PascalCase

Todas las palabras juntas, sin espacios. Todas las palabras, inclusive la primera, comienzan con mayúsculas y siguen en minúsculas. Ejemplos:

```
EstaEsUnaFraseEnPascalCase
Pila
AgregarElemento
```

camelCase

Todas las palabras juntas, sin espacios. Todas las palabras comienzan con mayúsculas y siguen en minúsculas, excepto la primera que está en minúsculas. Ejemplos:

```
estaEsUnaFraseEnCamelCase
unaPila
laLongitud
```

UPPERCASE

Todas las palabras en mayúsculas, pueden estar todas juntas o todas separadas con underscores. Ejemplos:

```
ESTAESUNAFRASEENUPPERCASE
ESTA_ES_OTRA
MAXIMO
```

Underscores (Guiones Bajos "_")

· Sí usar con `UPPER_CASE`.

· No usar delante de identificadores, ya que la biblioteca estándar tiene reservado esa forma de identificadores.

[K&R1988] "Chapter 1 – Chapter 2 - Types, Operators and Expressions – 2.1 Variables names"

· No usar con `camelCase` ni con `PascalCase`. Hay una excepción a esta última regla relacionada a los prefijos de los identificadores para las operaciones de los TAD.

Identificadores para diferentes categorías de elementos

La regla general es: usar *nombres significativos* para *identificadores significativos* y *nombres no significativos* para *identificadores no significativos*.

Los nombres significativos son en general largos y los no significativos cortos.

La misma regla se puede aplicar para identificadores de acceso global e identificadores de acceso local.

Así, una función que permite calcular un balance –tiene acceso global y es un concepto significativo– debe tener un identificador como `GetBalance`; mientras que una variable que se utiliza para iterar un arreglo –tiene acceso local y no es un concepto significativo– debe tener un identificador tan simple como `i`.

Nombres de Tipo (typedef)

En PascalCase, sustantivo en singular.

```
SistemaPlanetario
Planeta
NumeroAstronomico.
```

Funciones y Macros-con-parámetros

En PascalCase, deben comenzar con un verbo en infinitivo y en general son seguidas por un sustantivo o frase.

```
OrdenarArreglo()
SepararUsuarioDeDominio()
```

Cuando la función o macro-con-parámetros implementan una operación de un TAD el identificador debe comenzar con un prefijo formado por el nombre del TAD y un *underscore*:

```
Planeta_SetNombre()
NumeroAstronomico_EsOverflow()
```

Variables Locales y Parámetros

En camelCase, sustantivo, en plural para arreglos.

Variables Externas (Globales)

En PascalCase, sustantivo, en plural para arreglos.

Enumeraciones

El nombre del `typedef` de la enumeración y sus elementos en PascalCase y en singular.

Constantes Simbólicas (#define)

En `UPPER_CASE_CON_UNDERSCORES`.

De **[K&R1988] "Chapter 6 – A Tutorial Introduction – 1.4 Symbolic Constants"**

<< Las constantes simbólicas se escriben por convención en mayúsculas para que puedan ser rápidamente distinguidas de los nombres de variables escritos en minúscula. >>

No Usar Notación Húngara

La notación Húngara es una notación, que entre otras cosas, promueve la práctica de prefijar los identificadores para incluir *metadata* (datos sobre los datos) al identificador, como por ejemplo el tipo de dato de una variable.

Charles Simonyi desarrolló esta notación en Microsoft –llamada así por lo extraño de los identificadores resultantes y por la nacionalidad del autor– pero luego, la propia Microsoft desestimó su uso.

Una copia del paper original por *Charles Simonyi* se encuentra en

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs600/html/HungaNotat.asp

<< No use notación Húngara. Los nombres buenos describen semántica, no tipo de dato. >>

Este tipo de notación genera dependencia con el lenguaje y complica el mantenimiento de los programas. La funcionalidad de la Notación Húngara hoy la proveen los IDEs (Integrated Development Enviroments) modernos, que con solo posicionar el cursor sobre el identificador informa los atributos anunciados en su declaración. Por otro lado, las funciones (procedimientos, métodos) deben ser diseñadas de una forma tal que requieran pocas líneas de código, por lo que la declaración de las variables y parámetros se encuentran en un contexto acotado que facilita la ubicación de las declaraciones.

[Design Guidelines for Class - Naming Guidelines - Parameter Naming Guidelines]

http://msdn.microsoft.com/library/en-us/cpgenref/html/cpconFieldUsageGuidelines.asp

"Best Practices"

Buenas Prácticas de Programación

No Usar Números Mágicos

De **[K&R1988] "Chapter 6 – A Tutorial Introduction – 1.4 Symbolic Constants"**

<< Es mala práctica enterrar en un programa "números mágicos" como 300 y 20; contienen poca información para alguien que pueda tener que leer el programa más adelante, y son difíciles de cambiar de una manera sistemática [durante el mantenimiento].

Una forma a tratar el tema de los números mágicos es darles nombres significativos. Una línea *#define* define a un nombre simbólico o constante simbólica como una cadena particular de caracteres:

`#define` <*nombre*> <*texto de reemplazo*>

Después de eso, cualquier ocurrencia del nombre (pero no entre comillas ni como parte de otro nombre) será substituida por el texto de reemplazo correspondiente. El nombre tiene la misma forma que un nombre de variable: una secuencia de letras y de dígitos que comienza con una letra. El texto de reemplazo puede ser cualquier secuencia de caracteres; no se limita a los números.>>

Evitar Variables Innecesarias

De **[K&R1988] "Chapter 6 – A Tutorial Introduction – 1.3 The For Statement"**

<<… en cualquier contexto donde se permite usar el valor de algún tipo, se pude usar una expresión más complicada de ese tipo. >>

Esto evita el uso de variables innecesarias

```
/* Suponiendo una variable int a */
/* Correcto */
printf("Cociente: %d\nResto: %d\n", a / 2, a % 2);
/* Suponiendo una variable int a */
/* Incorrecto */
int c = a / 2;
int r = a % 2;
printf("Cociente: %d\nResto: %d\n", c, r);
```

Escribir las Constantes Punto Flotante con Puntos Decimales

De **[K&R1988] "Chapter 1 – A Tutorial Introduction – 1.2 Variables and Arithmetic Expressions"**

<< … escribir constantes de punto flotante con puntos decimales explícitos inclusive cuando son valores enteros enfatiza su naturaleza de punto flotante a los lectores humanos. >>

No Usar Sentencias `goto`

De **[K&R1988] "Chapter 3 – Control Flow – 3.8 Goto and labels"**

<< Formalmente, la sentencia `goto` nunca es necesaria, y en la práctica es casi siempre fácil escribir código sin ella…>>

<<…Sin embargo, hay algunas situaciones donde los `goto`s pueden encontrar su lugar. El más común es abandonar el proceso dentro de alguna de la estructura profundamente anidada, por ejemplo salir de dos o ciclos inmediatamente.

La sentencia `break` no puede ser usada directamente ya que solo sale del ciclo más interno…>>

<< …El código que involucre un `goto` puede siempre ser escrito sin él, aunque quizás al precio de algunas pruebas repetidas o una variable extra…>>

<< … Con algunas excepciones como las citadas aquí, un código que utilice sentencias `goto` es en general más difícil de comprender y de mantener que un código sin `goto`s. Aunque no somos dogmáticos sobre el tema, sí parece que las sentencias goto se deben utilizar en raras oportunidades, o quizás nunca…>>

De **[K&R1988] "Chapter 5 – Pointers and Arrays"**

<<…Los punteros, junto con la sentencia goto, han sido duramente castigados y son conocidos como una manera maravillosa de crear programas imposibles de entender. Pero esto es verdad solo cuando son utilizados negligentemente…

>>

Usar Variables Externas (globales) Solo Cuando se Justifica

Las variables externas son llamadas así porque son definidas fuera de cualquier función. Son también conocidas como variables globales por que son globalmente accesibles.

De **[K&R1988] "Chapter 5 – Pointers and Arrays"**

<< Basarse demasiado en variables externas es peligroso puesto que conduce a programas con conexiones de datos que no son del todo obvias –las variables se pueden cambiar en maneras inesperadas e incluso inadvertidas– y el programa se torna difícil de modificar. La segunda versión del programa "línea más larga" es inferior a la primera, en parte por estas razones, y en parte porque destruye la generalidad de dos útiles funciones al escribir dentro de ellas los nombres de las variables que manipulan. >>

Estructurar los archivos del proyecto correctamente

Conozca la forma correcta de diseñar la distribución de archivos que forman parte de un proyecto. Utilice los archivos encabezados para incluir declaraciones públicas de cualquier tipo salvo definiciones de funciones. Las definiciones de las funciones se escriben en un archivo fuente aparte, mientras que en el encabezado se ubican los prototipos las funciones públicas. No utilice directivas del tipo:

```
#include "mal.c"
```

Diseñe bibliotecas de forma genérica para que pueden volver a utilizarse eviatnado así el "copy-paste" de código.

No Permitir Inclusiones Múltiples de Headers (Archivos Encabezado)

El contenido de los archivos header debe estar "rodeado" por las siguientes directivas al preprocesador:

```
#ifndef HEADER_NOMBRE_YA_INCLUIDO
#define HEADER_NOMBRE_YA_INCLUIDO
/* Contenido del archivo encabezado. */
#endif
```

Esto permite la compilación condicional, evitando incluir más de una vez el contenido del archivo encabezado. Reemplazar `NOMBRE` por el nombre del header.

Diseñar las Funciones Correctamente

De **[K&R1988] "Chapter 6 – A Tutorial Introduction – 1.7 Functions"**

<< Una función proporciona una manera conveniente de encapsular un cómputo, la cual puede entonces ser utilizada sin preocuparse de su implementación. Con funciones correctamente diseñadas, es posible ignorar *cómo* se hace un trabajo; saber *qué* es lo que se hace es suficiente. C hace el tema de funciones fácil, conveniente y eficiente; usted verá a menudo una función corta definida y llamada una única vez, solo porque clarifica una cierta sección del código.>>

Las funciones deben abstraer un proceso, acción, actividad o cálculo. Lo importante para quien invoca a la función es "que" hace pero no "como" lo hace. Para diseñar correctamente una función considerar los siguientes parámetros de diseño:

Ocultamiento de Información

Diseñe las funciones de tal manera que no expongan detalles de implementación. No debe ser conocido por quien invoca la función "como" es que la función realiza la tarea.

Alta Cohesión

Significa que una función debe realizar solo una tarea.

Supóngase que se esta escribiendo un programa que resuelve integrales, la función `MostrarResultado` deberá solo imprimir un resultado, no calcularlo. En forma similar, la función encargada de calcular este resultado, `Integrar`, no deberá imprimirlo. Esto no descarta que exista una función que utilice a ambas funciones para resolver la integral y mostrar su resultado. Tampoco es correcto que dos funciones hagan una tarea a medias. Una manera de estimar la cohesión es por medio de su longitud: ***una función no debe tener más de 20 sentencias o declaraciones***.

Bajo Acomplamiento

Implica que una función tiene poca dependencia con el resto del sistema, para poder utilizar una función debe ser suficiente con conocer su prototipo (su parte pública).

Un ejemplo de alto acoplamiento es una función que retorna un valor modificando una variable global; si el identificador de la variable global cambia, se debe cambiar la función para que siga comportándose correctamente.

Sobre la semántica algunos agregados

Terrence Pratt: "Programming Languages – Design and Implementation, 2nd edition", Prentice Hall

Pratt p.26

La SINTAXIS de un LP es la forma en la cual los programas son escritos.

La SEMÁNTICA de un LP es el significado otorgado a los diferentes constructos sintácticos. Este significado tiene que ver con lo que sucede en tiempo de ejecución.

Ejemplo 1:

Sea en Pascal la declaración **V: array [1..10] of real;**

Esta declaración brinda la sintaxis de un vector de 10 elementos reales llamado V.

La semántica de esta declaración (el significado de la misma) es, por ejemplo: esta declaración se coloca al comienzo de un subprograma para crear, en cada invocación a ese subprograma, el vector mencionado, cuyo espacio será liberado al terminar de ejecutarse el subprograma.

Ejemplo 2:

Analicemos una situación en ANSI C. Sea la sentencia **while**.

Su sintaxis es: **while (** *expresión* **)** *sentencia*

Su semántica, según el MROC, dice: la evaluación de la expresión de control se realiza antes de cada ejecución del cuerpo del ciclo. Entonces, el cuerpo del ciclo es ejecutado repetidamente hasta que el expresión sea nula (0, 0.0, etc).

Ejemplo 3:

En ANSI C, la semántica de **while** (2) 3; representa un ciclo infinito.

Pratt p.345
Problemas en Semántica

Problema 1: Definición de Semántica

El problema práctico

Un manual de un LP debe definir el significado de cada construcción del lenguaje, tanto en forma aislada como en conjunción con otras construcciones del lenguaje. Un lenguaje provee una variedad de diferentes constructos, y tanto el usuario del lenguaje como el implementador requieren una definición precisa de la semántica de cada constructo. El programador necesita la definición para escribir programas correctos y para ser capaz de predecir el efecto de la ejecución de cualquier sentencia del programa. El implementador necesita la definición para poder construir una implementación correcta del LP.

En la mayoría de los manuales de los LPs la definición de la semántica está dada en lenguaje natural. Comúnmente, una producción (o producciones) de una BNF son dadas para definir la sintaxis de un constructo, y luego unos párrafos en lenguaje natural y algunos ejemplos son dados para definir la semántica.

Pratt p.317
Análisis Semántico

Completa el Análisis Sintáctico y, además, puede producir el código objeto (generalmente para una MV).

El Análisis Semántico se lleva a cabo a través de un conjunto de rutinas, cada una de las cuales se ocupa de una actividad o de un constructo.

Ejemplo 4:

Las declaraciones de los arreglos pueden ser manejadas por una rutina semántica; las expresiones aritméticas por otra, etc. Una rutina semántica apropiada es llamada por el Parser cada vez que éste reconoce un constructo que debe ser procesado.

Una rutina semántica puede actuar conjuntamente con otras rutinas semánticas para llevar a cabo una tarea determinada.

Pratt p.318
Mantenimiento de la TS.

Una TS es una estructura central en todo compilador. Una TS típica contiene una entrada para cada identificador que existe en el programa fuente. El Scanner coloca el identificador detectado, pero el Análisis Semántico (AS) tiene la responsabilidad fundamental después de ello.

En general, la TS no solo contiene a los identificadores sino que también contiene los atributos de cada uno de ellos: su clase (variable simple, arreglo, función, parámetro, etc.), tipo de los valores (entero, real, etc.), alcance, otra información que proveen las declaraciones (y definiciones), etc.

El AS, a través de sus diferentes rutinas, ingresa esta información en la TS a medida que procesa declaraciones, encabezamientos de funciones y sentencias del programa.

Otras rutinas del AS utilizan esta información para construir, luego, códigos ejecutables eficientes.

- - - - - -

Otros Ejemplos de descripción de la Semántica, según Pratt:

(a)
Describa la semántica de la sentencia compuesta ANSI C:

```
{i = 0; while (i < 20){ scanf("%s", cad); printf("cadena: %s\n", cad); i++; } }
```

Respuesta:
Despliega por pantalla 20 cadenas ingresadas por teclado, a razón de una cadena por línea, con el formato cadena: seguida de la cadena que previamente fue ingresada por teclado. Al final, el cursor queda al comienzo de la siguiente línea.
O bien,
Se ingresan 20 cadenas por teclado, a razón de una cadena por línea. Por cada cadena ingresada, debajo de ella, se despliega por pantalla cadena: y la cadena previamente ingresada. Quedando el cursor al comienzo de la siguiente línea .

(b)
Describa la semántica de la sentencia compuesta ANSI C:

```
{ int a = 0; do printf("%d\n",++a); while(a < 20); }
```

Respuesta:
Despliega por pantalla los valores del 1 al 20, a razón de uno por línea. El cursor queda al comienzo de la siguiente línea.

(c)
Describa la semántica de la sentencia compuesta ANSI C:

```
{ int b = 0; do printf("%d\n",b++); while(b < 40); }
```

Respuesta:
Despliega por pantalla los valores del 0 al 39, a razón de uno por línea. El cursor queda al comienzo de la siguiente línea.

(d)
Defina la semántica de los siguientes definiciones/declaraciones en ANSI C:

(1) typedef struct {int a; double b; } XX;

Respuesta:
Se declara a XX como el nombre de un tipo que es una estructura de dos campos: el primero int llamado a y el segundo double llamado b.

(2) struct {int a; double b; } XX;

Respuesta:
Se define la variable XX como una estructura formada por dos campos: el primero int llamado a y el segundo double llamado b.

(3) double mat[10][20];

Respuesta:
Se define el arreglo bidimensional (o la matriz) llamado mat de 10 filas por 20 columnas, en donde cada elemento es double.

(4) long a, b, c=0;

Respuesta:
Se definen tres variables de tipo long, lamadas a, b y c, donde la variable c comienza con el valor 0.

(e)
Defina la semántica de la siguiente función en ANSI C:

```
int XX (char *s, char *t)
{ if (strlen(s) == strlen(t)) return 1;
  return 0;
}
```

Respuesta:
Retorna 1 si las longitudes de las cadenas dadas son iguales y 0 si no lo son.

(f)
Describa la Semántica de la siguiente función ANSI C:

```
int XX (char *s, char *t) {
   if (strcmp(s,t)   return 1;
   return 0;
}
```

Respuesta:
Retorna 1 si las dos cadenas dadas son iguales; cosa contrario, retorna 0.

(g)
Sea la sentencia ANSI C:

```
printf ("%d", AS(24));
```

Describa la semántica de esta sentencia teniendo en cuenta lo que realiza la función:

```
int  AS (int x) {
   return x-4;
}
```

Respuesta:
Muestra (o despliega) por pantalla el valor 20 y el cursor queda a continuación en la misma línea.

(h)
Sea la siguiente sentencia compuesta ANSI C:

```
{ int i=0, t=10; while (i < 5) { printf ("%d\n", MM(t)); t+=4;  i++;} }
```

Describa la semántica de esta sentencia compuesta teniendo en cuenta lo que realiza la función:

```
int MM (int x) {
    return x*2;
}
```

Sugerencia: Para poder describir la Semántica de esta sentencia compuesta, primero haga una prueba de escritorio para comprender qué es lo que hace la sentencia compuesta.

Respuesta:
Muestra (o despliega) por pantalla los valores 20, 28, 36, 44 y 52, a razón de uno por línea, y el cursor queda al comienzo de la siguiente línea.

(i)
Describa la semántica de la siguiente función ANSI C:

```
float SS (void) {
  int i;
  float g25, cc=0.0;
  for (i=0; i < 20; i++)  {
     scanf("%f",  &g25);
     cc+=g25;
  }
 }
```

Respuesta:
Retorna la suma de los primeros 20 valores reales ingresados por teclado.

(j) Describa la Semántica de la siguiente función ANSI C:

```
long XX (long w, long z) {
    if (w > z)   return w;
    return z;
}
```

Aclaración: los valores de w y de **z** siempre serán mayores a cero.

Respuesta:
Dados dos valores enteros, retorna el mayor de ellos.

Bibliografía

Bibliografía Consultada

- **The C Programming Language**, 2nd Edition [K&R1988]
- **Hugarian Notation**

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs600/html/HungaNotat.asp

- **Identation Styles**

http://en.wikipedia.org/wiki/Indent_style

Bibliografía Recomendada

- **The practice of programming**, por Kernighan & Pike, Addison-Wesley
- **Java Coding Style Guide**, por Reddy, Sun Microsystems, Inc.
- **C# Language Specification – Naming guidelines**, ECMA/Microsoft.—