

Apuntes de fundamentos de la programación

Lara

September

Máquina, programa, cómputo, programación, código máquina, lenguaje simbólico o fuente, compilador, intérprete.

1 Modelos abstractos de cómputo

1.1 Modelo funcional

Reducción, reescritura

Un programa funcional será siempre en último extremo una aplicación de una función a unos argumentos, para obtener un resultado. El proceso de cómputo, llamado reducción, se basa en reemplazar progresivamente cada función por el resultado de la misma. Este sistema de evaluación por sustitución es la base del llamado *calculo*— λ . La programación funcional permite la definición por parte del programador de nuevas funciones a partir de las ya existentes.

El proceso, llamado reescritura, consiste en reemplazar una función por su definición, sustituyendo los argumentos simbólicos en la definición por los argumentos reales en el cómputo.

1.2 Modelo de flujo de datos

En este modelo de cómputo, un programa corresponde a una red de operadores interconectados entre sí. Un operador espera hasta tener valores presentes en sus entradas, y entonces se activa él solo, consume los valores en las entradas, calcula el resultado, y lo envía a la salida. Después de esto vuelve a esperar que le lleguen nuevos valores por las entradas.

1.3 Modelo de programación lógica

Denominado *programación declarativa*, *declarar hechos y reglas*.

Un programa consiste en plantear de manera formal un problema a base de declarar una serie de elementos conocidos, y luego preguntar por un resultado, dejando que sea la propia máquina la que decida cómo obtenerlo. En programación lógica los elementos conocidos que pueden declararse son hechos y reglas. Un hecho es una relación entre objetos concretos. Una regla es una relación general entre objetos que cumplen ciertas propiedades. Para realizar una consulta escribiremos el esquema de un hecho en que alguno de los elementos sea desconocido. La consulta será respondida indicando todos los valores posibles que puedan tomar las incógnitas. La verdadera potencia de la programación lógica aparece cuando declaramos reglas. Al realizar consultas basadas en reglas la máquina realiza automáticamente las inferencias (deducciones) necesarias para responderla.

1.4 Modelo imperativo

Arquitectura Von Neumann

El modelo de programación imperativa responde a la estructura interna habitual de un computador, o una lista de instrucciones u órdenes elementales que han de ejecutarse una tras otra, en el orden en que aparecen en el programa. El nombre de programación imperativa deriva del hecho de que un programa aparece como una lista de órdenes a cumplir.

El orden de ejecución puede alterarse en caso necesario mediante el uso de instrucciones de control. Con ello se consigue ejecutar o no, o repetir, determinadas partes del programa dependiendo de ciertas condiciones en los datos. Las instrucciones de un programa imperativo utilizan datos almacenados en la memoria del computador. Esta capacidad de almacenamiento de valores se representa en los programas imperativos mediante el uso de variables. Una variable no tiene aquí el mismo significado que en matemáticas, sino que representa un dato almacenado bajo un nombre dado. Una variable contiene un valor que puede ser usado o modificado tantas veces como se desee.

Un programa imperativo se plantea como el cálculo o modificación de sucesivos valores intermedios hasta obtener

el resultado final. Las instrucciones típicas de un programa imperativo son la de asignación, que consisten en obtener un resultado parcial mediante un cálculo elemental que puede ser realizado por la máquina, y que se almacena en una variable para ser utilizado posteriormente. En los lenguajes de programación simbólicos las instrucciones u órdenes se denominan sentencias.

Elementos de la programación declarativa: Procesador, entorno, acciones.

Definiremos como procesador a todo agente capaz de entender las órdenes del programa Y ejecutarlas. El procesador es esencialmente un elemento de control. Para ejecutar las instrucciones empleará los recursos necesarios, que formarán parte del sistema en el cual se ejecute el programa, se necesitarán dispositivos de almacenamiento para guardar datos que habrán de ser utilizados posteriormente, o dispositivos de entrada-salida que permitirán tomar datos de partida del exterior y presentar los resultados del programa. Todos estos elementos disponibles para ser utilizados por el procesador constituyen el entorno.

Las órdenes o instrucciones del programa definen determinadas acciones que deben ser realizadas por el procesador. Un programa imperativo aparece la descripción de una serie de acciones a realizar en un orden preciso.

Acciones primitivas y compuestas

La manera en la que varias acciones sencillas se combinan para realizar una acción complicada se denomina esquema de la acción compuesta.

La llamada programación estructurada sugiere el uso de tres esquemas generales denominados secuencia, selección e iteración, con los cuales (junto con la definición de operaciones abstractas) se puede llegar a desarrollar de forma comprensible un programa tan complicado como sea necesario.

2 Elementos básicos de programación

Notación BNF: Descripción de cada elemento gramatical en función de otros más sencillos, según determinados esquemas o construcciones. Cada uno de estos esquemas se define mediante una *regla de producción*

Valores y tipos

Un dato es un elemento de información que puede tomar un valor entre varios posibles. En general un dato sólo puede tomar valores de una clase. En programación a las distintas clases de valores se les denomina tipos.

En $C\pm$:

$::=$ Metasímbolo de definición

| Metasímbolo de alternativa

() Metasímbolo de agrupación

{ } Metasímbolo de repetición

[] Metasímbolo de opción

valores enteros, valores reales (exponente y mantisa).

Caracteres: " ", la colección o juego de caracteres de cada máquina, se van a representar en el programa solo aquellos que tengan un símbolo gráfico asociado.

Los caracteres de control no tienen un símbolo gráfico asociado y se utilizan como secuencia de escape.

" \ n": Salto al comienzo de una nueva línea de escritura

" \ r": Retorno al comienzo de la misma línea de escritura

" \ t": Tabulación

" \'": Apostrofe

" \ ": Barra inclinada

" \ f": Salto a una nueva página o borrado de pantalla.

Cadena de caracteres: string

En $C\pm$ se usa apostrofes ' ' para un solo carácter y dobles comillas " " para strings

Tipos predefinidos: dentro de la misma clase de valores pueden distinguirse varios tipos diferentes, tanto a nivel de tipos predefinidos en el lenguaje, como de tipos definidos por el programador.

Recordaremos que un tipo de datos define:

1. Una colección de valores posibles

2. Las operaciones significativas sobre ellos

En $C\pm$ hay cuatro tipos predefinidos: int, float, char, bool, así como mecanismos para definir nuevos tipos a partir de ellos.

tipo int: definido igual que los números enteros pero es finito, con un rango dependiendo de la plataforma (INT_MIN...0...INT_MAX)

Las operaciones predefinidas entre valores enteros son las operaciones aritméticas básicas, que se realizan entre enteros y devuelven como resultado valores enteros.

(El operador `/` realiza la división entre dos números enteros y obtiene como resultado el cociente entero truncado al valor más próximo a cero.)

tipo `float`: trata de representar los valores reales positivos y negativos pero esta representación no puede ser siempre exacta. Se representan con una mantisa y un factor de escala. Las operaciones entre valores reales son las operaciones aritméticas básicas que se realizan entre reales y devuelven como resultado valores reales. Las operaciones entre reales dan como resultado un real con la precisión de la plataforma.

tipo `char`: Cada carácter no se representa internamente como un dibujo (el glifo del carácter), sino como un valor numérico entero que es su código. La colección concreta de caracteres y sus códigos numéricos se establecen en una tabla (charset) que asocia a cada carácter el código numérico (codepoint) que le corresponde. Dependiendo del número de bits reservado para representar el código de cada carácter podremos tener tablas más o menos amplias.

ASCII (7 bits), ISO-8859-1 (8 bits), UNICODE-BPM (16 bits), UNICODE (32 bits).

Tabla (<i>charset</i>)	Tamaño del carácter	Repertorio de caracteres
ASCII	7 bits	Letras inglesas mayúsculas y minúsculas. Algunos signos de puntuación y códigos de control.
ISO-8859-1 (llamado también Latin-1)	8 bits	Lo anterior, más letras con acentos y nuevos signos de puntuación
UNICODE-BMP (Basic Multilingual Plane)	16 bits	Incluye además los alfabetos griego, cirílico, árabe, chino/japonés/coreano, signos matemáticos, etc.
UNICODE completo	32 bits	Incluye la práctica totalidad de caracteres utilizados en cualquier idioma o notación textual existente en nuestro mundo actual.

Figure 1: charset1

Compatibles entre sí, cada una contiene a la anterior manteniendo sus códigos numéricos. Pero esto no es siempre así, lo vemos con tablas de 8bits como:

Tabla (<i>charset</i>)	Tamaño del carácter	Repertorio de caracteres
ISO-8859-7	8 bits	Repertorio ASCII más el alfabeto griego
ISO-8859-15	8 bits	Repertorio Latin-1 revisado. Incluye el símbolo de Euro
IBM-PC-437 (original del sistema operativo MS-DOS)	8 bits	Repertorio ASCII más símbolos semigráficos y letras con acentos, pero con códigos distintos de Latin-1
IBM-PC-850 (usado también en MS-DOS)	8 bits	Casi el mismo repertorio de Latin-1 pero con códigos diferentes, y diferentes también de IBM-PC-437.
Windows-1252 (usado en los sistemas operativos MS-Windows)	8 bits	Coincide con Latin-1 excepto en un rango de 32 códigos de Latin-1 que repiten códigos de control

Figure 2: charset2

En $C\pm$ (como en `C` y `C++`) los valores del tipo `char` ocupan 8 bits e incluyen el repertorio ASCII. Además incluyen otros caracteres no-ASCII que dependen de la tabla de caracteres establecida. Asumiremos que se dispone de los caracteres comunes a Latin-1 y Windows-1252. Por lo tanto la colección de valores del tipo `char` incluye caracteres alfabéticos, numéricos, de puntuación y caracteres de control.

Se puede representar cualquier carácter mediante la notación `char(x)` siendo `x` el código del carácter.

Sobre la tabla ASCII:

`char(10)`: salto al comienzo de una nueva línea

`char(13)`: retorno al comienzo de una misma línea

`char(65)`: letra A mayúscula

siendo: `int('A') = 65`

`int('Z') = 90`

De forma inmediata se puede decir que, para cualquier carácter `c`, cuyo código sea `x`, se cumplirá que:

`char(int(c)) = c` `int(char(x)) = x`

Conviene saber:

- Los caracteres correspondientes a las letras mayúsculas de la 'A' a la 'Z' están ordenados en posiciones consecutivas y crecientes según el orden alfabético.
- Los caracteres correspondientes a las letras minúsculas de la 'a' a la 'z' están ordenados en posiciones consecutivas y crecientes según el orden alfabético.
- Los caracteres correspondientes a los dígitos del '0' al '9' están ordenados en posiciones consecutivas y crecientes.

Esto facilita para obtener por cálculo el valor numérico equivalente al carácter de algún dígito.

En C (y en $C\pm$) se puede usar también el módulo de la librería `ctype` (cabecera `<ctype.h>`), que facilita el manejo de diferentes clases de caracteres. Este módulo incluye funciones tales como: `isalpha(e)` Indica si `e` es una letra

`isascii(e)` Indica si `e` es un carácter ASCII

`isblank(e)` Indica si `e` es un carácter de espacio o tabulación

`isctrl(e)` Indica si `e` es un carácter de control

`isdigit(e)` Indica si `e` es un dígito decimal (0-9)

`isslower(e)` Indica si `e` es una letra minúscula `isspace(e)` Indica si `e` es espacio en blanco o salto de línea o página

`isupper(e)` Indica si `e` es una letra mayúscula

`tolower(e)` Devuelve la minúscula correspondiente a `e`

`toupper(e)` Devuelve la mayúscula correspondiente a `e`

Operaciones aritméticas: conjunto de operandos y operadores. Mejor explicitar el orden de operaciones con paréntesis para cálculos complejos. Cuidado con combinar en una misma operación distintos tipos de dato. Aunque en $C\pm$ permite la ambigüedad que supone la mezcla de tipos de datos diferentes en la misma expresión sin tener que exigir una conversión explícita, utiliza el convenio $C/C++$ de convertir previamente de manera automática todos los valores de la misma expresión al tipo de mayor rango y precisión. Por tanto, el resultado siempre se obtendrá también en el mayor rango y precisión utilizado en la expresión, puede dar a errores si no se sabe esta regla implícita. En el *manual de Estilo* se obligará a hacer la conversión de tipos de manera explícita.

2.1 Operaciones de estructura simple

El objetivo de un programa es obtener unos resultados. Estos resultados deben ser emitidos al exterior del computador a través de un dispositivo de salida de datos. Las acciones que envían resultados al exterior se llaman, en general, *operaciones de escritura*.

Para simplificar la escritura de resultados los lenguajes de programación prevén sentencias de escritura apropiadas para ser usadas con cualquier tipo de dispositivo, facilitando la tarea de programación al especificar la escritura de resultados de una manera uniforme, con independencia de las particularidades del dispositivo físico que se utilice en cada caso.

Al diseñar un lenguaje de programación se puede optar por usar sentencias o instrucciones especiales para ordenar la escritura de resultados, o bien ordenar la escritura del resultado con las mismas sentencias general que se empleen para invocar operaciones definidas por el usuario.

Los primeros lenguajes de programación solían emplear la primera alternativa. Los lenguajes más modernos utilizan con preferencia la segunda, que simplifica la complejidad del lenguaje en sí, a costa de permitir a veces una cierta variación en las operaciones de escritura entre diferentes versiones del lenguaje.

En $C\pm$ como en $C/C++$ adoptan la segunda alternativa. Las operaciones de escritura se definen como procedimientos. Estos procedimientos están disponibles en los módulos de librería disponibles de antemano. En

todas las versiones de C/C++ deben estar disponibles ciertos módulos estándar con la definición de operaciones de escritura normalizada.

operación disponible en el módulo llamada *stdio*

El procedimiento **printf** pertenece al módulo *stdio*-

Una cadena de caracteres con formatos deberá incluir en su interior una especificación de formato por cada valor que se quiera insertar. La forma más simple de especificar un formato es mediante %x, es decir, usando el carácter fijo % seguido de una letra de código que indica el tipo de formato a aplicar. Ej:

Código	Nemotécnico (inglés)	Tipo de valor
d	decimal	entero
f	fixed point	real
e	exponential	real con notación exponencial
g	general	real con/sin notación exponencial
c	character	un carácter
s	string	una cadena de caracteres

Figure 3: printf

```
printf("%d", 120/12) resultado= 10
```

```
printf("Datos:%d#%d", 23*67, -50) resultado= Datos:1541#-50
```

Como se puede apreciar en los ejemplos estos formatos simples usan sólo el número de caracteres estrictamente necesarios para escribir el valor de cada dato, sin añadir espacios en blanco. Si se quiere separar con espacios unos valores de otros, entonces hay que incluirlos en el formato.

Otra forma de conseguir espacios en los resultados es indicar explícitamente cuántos caracteres debe ocupar el valor de cada dato escrito. Esto se hace poniendo el número de caracteres entre el símbolo de %y el código del formato.

```
printf("%5d", 120/12) resultado= ...10
```

```
printf("Datos:%7d#%5d", 23*67, -50) resultado= Datos:...1541#..-50
```

```
printf("%3d", 34*1000) resultado= 34000
```

Cuando el número de caracteres indicado es insuficiente para representar completamente el valor, como ocurre en el último ejemplo, se utilizan tantos caracteres como sean necesarios para que el resultado aparezca completo.

Además, cuando se utiliza un formato f , e ó g se puede especificar también el número de cifras decimales que se deben escribir después del punto decimal. Salvo que se indique otra cosa, los resultados obtenidos mediante

Operación de escritura	Resultado
printf("%10.3f", 1.2);1.200
printf("%10.4e", 23.1*67.4);	0.1557E+04
printf("%15.3g", -50.6E-6);-0.506E-04

Figure 4: printf1

sucesivas sentencias de escritura van apareciendo en el dispositivo de salida uno tras otro en la misma línea de texto.

```
printf( "Area = " );
printf( "%10.4f\n", 24.45 );
printf( "Mi ciudad es Avila\n" );
printf( "Descuento: " );
printf( "%5.2d", 12.5 );
printf( "%c\n", '%' );
```

se obtendrá como resultado:

```
Area =      24.4500
Mi ciudad es Avila
Descuento: 12.50%
```

Figure 5: printf2

2.2 Escritura de programa completo

Necesitamos un IDE y un compilador

Un programa $C\pm$ se engloba todo en una estructura principal o `main()`.

El símbolo `#` es una *directiva para el ordenador*, en concreto en una línea de código de `#include <stdio.h>` con `include` se indica al ordenador que utilice el módulo de la librería `stdio`. La directiva `#include` da a ser la única que utilizemos en $C\pm$

El cuerpo del programa contiene las sentencias ejecutables correspondientes a las acciones a realizar, escritas entre los símbolos `{}` de comienzo y final. Cada sentencia de programa termina en un punto y coma(`;`). Los programas en $C\pm$ se deben guardar en un fichero de extensión `.cpp`. El nombre del fichero fuente será: `sunombre.cpp`

Uso de comentarios:

En $C\pm$ los comentarios se incluyen dentro de los símbolos `/* y */`

Cada directiva debe ocupar una línea del programa ella sola, en C/C++ hay una gran cantidad de directivas, pero en $C\pm$ usaremos casi exclusivamente `include`: el programa utilizará un determinado módulo de una librería. El parámetro `Nombre_módulo` corresponde en realidad al nombre del fichero de cabecera (header) del módulo. Diremos que un bloque puede contener una secuencia de sentencias.

Ejemplos:

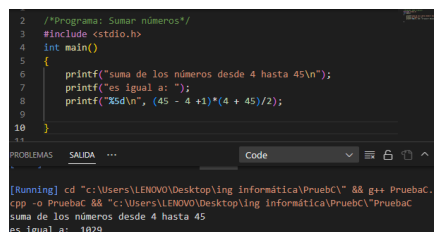
Suma de números consecutivos

Si n es el número de términos, a_1 el primer término, y a_n el último, la fórmula general de la suma es:

$$\sum_{i=1}^n = nx \frac{a_1 + a_n}{2}$$

Y para una serie de números enteros consecutivos de cumple que:

$$n = a_n - a_1 + 1$$



```
2  /*Programa: Sumar números*/
3  #include <stdio.h>
4  int main()
5  {
6      printf("suma de los números desde 4 hasta 45\n");
7      printf("es igual a: ");
8      printf("%d\n", (45 - 4 + 1)*(4 + 45)/2);
9  }
10
11
[Running] cd "c:\Users\LENOVO\Desktop\ing informática\PruebaC\" && g++ PruebaC.cpp -o PruebaC && "c:\Users\LENOVO\Desktop\ing informática\PruebaC\PruebaC.exe"
suma de los números desde 4 hasta 45
es igual a: 1029
```

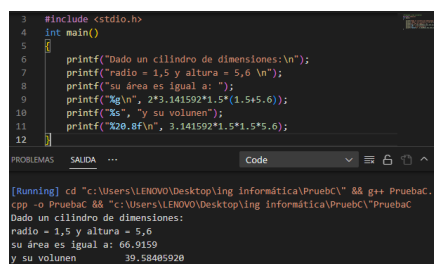
Figure 6: programa1

Área de un cilindro:

A partir de un radio R y su altura A :

$$area = 2\pi R^2 + 2\pi RA = 2\pi R(R + A)$$

$$volumen = \pi R^2 A$$



```
3  #include <stdio.h>
4  int main()
5  {
6      printf("Dado un cilindro de dimensiones:\n");
7      printf("radio = 1,5 y altura = 5,6\n");
8      printf("su área es igual a: ");
9      printf("%g\n", 2*3.141592*1.5*(1.5+5.6));
10     printf("%s", "y su volumen");
11     printf("%20.8f\n", 3.141592*1.5*1.5*5.6);
12 }
[Running] cd "c:\Users\LENOVO\Desktop\ing informática\PruebaC\" && g++ PruebaC.cpp -o PruebaC && "c:\Users\LENOVO\Desktop\ing informática\PruebaC\PruebaC.exe"
Dado un cilindro de dimensiones:
radio = 1,5 y altura = 5,6
su área es igual a: 66.9159
y su volumen          39.58485920
```

Figure 7: programa2

3 Constantes y variables

La existencia de variables constituye elemento diferenciador del paradigma de la programación imperativa respecto a los demás.

Identificadores:

En programación se llaman identificadores a los nombres usados para identificar cada elemento del programa. Identificadores como tipo de dato, operaciones de escritura, esto se suministra con un significado determinado, Ahora se inventarán nombres para designar variables y constantes en el programa.

En $C\pm$ los identificadores son una palabra formada con caracteres alfabéticos o numéricos seguidos, sin espacios en blanco ni signos de puntuación intercalados, y que deben comenzar con una letra. Pueden usarse las 52 letras mayúsculas y minúsculas del alfabeto inglés, el guión bajo (-), y los dígitos decimales del 0 al 9.

En $C\pm$

Identificador ::= Letra {Letra — Guión — Dígito}

a la hora de inventar nombres conviene seguir unas reglas de estilo. en el manual de estilo de $C\pm$ se sugieren tres cosas, entre otras: escribir todo en minúscula, en mayúscula solo los nombres de constantes que sean globales o parámetros generales del programa (PI NULO MAXIMO) y usar guines o mayúsculas intermedias para los nombres compuestos.

A parte los identificadores tenemos las *palabras claves* que son elementos fijos del lenguaje, que sirven para delimitar las construcciones del lenguaje de programación. También son elementos fijos del lenguaje los nombres de los tipos fundamentales y los de algunas funciones especiales incorporadas en el propio lenguaje, este conjunto de elementos se denominan las *palabras reservadas*.

bool	break	case	catch	char	const	default
delete	do	else	enum	extern	false	float
for	if	int	new	private	return	struct
switch	true	try	typedef	union	void	while

Figure 8: palabras reservadas

and	and_eq	asm	auto	bitand	bitor
bool	break	case	catch	char	class
compl	const	const_cast		continue	default
delete	do	double	dynamic_cast		else
enum	explicit	extern	false	float	for
friend	goto	if	inline	int	long
mutable	namespace	new	not	not_eq	operator
or	or_eq	private	protected	public	register
reinterpret_cast		return	short	signed	sizeof
static	static_cast		struct	switch	template
this	throw	true	try	typedef	typeid
typename	union	unsigned	using	virtual	void
volatile	wchar_t	while	xor	xor_eq	

Figure 9: palabras reservadas lista completa

Hay algunos identificadores que sin ser reservados en C++ tienen un significado preciso para cada programa en el que aparezcan: main, NULL, std, string..

3.1 Constantes:

Una constante. es un valor fijo que se utiliza en un programa. El valor debe ser siempre el mismo para cualquier ejecución del programa, es decir, el valor no puede cambiar de una ejecución a otra. La declaración de un valor constante con nombre consiste en asociar un identificador a dicho valor constante. Ej en $C\pm$ en un programa podemos representar valores constantes escribiéndolo explícitamente, en forma de *constantes literales*, como se las denomina en programación o como *constantes simbólicas* o *constantes con nombre*.

Declaración de constantes con nombre:

Básicamente las que se les pone: **const tipo nombre = lo q sea**

Consiste en asociar un identificador a dicho valor constante.

De esta forma:

```
const float Pi = 3.14159265;
```

Se debe declarar la constante antes de ser utilizada. Una posibilidad interesante es poder declarar el valor de una constante en forma de expresión. En $C\pm$ solo se permite hacer esto si la expresión puede ser evaluada por el compilador en el momento de traducir el programa fuente a programa objeto. Para ello es necesario que todos los operandos que intervengan en la expresión sean valores constantes, y que las operaciones entre ellos sean operadores fijos del lenguaje o funciones predefinidas, en este caso la expresión se denomina *expresión constante*. Los operadores constantes pueden ser valores explícitos o constantes con nombre declaradas en algún punto anterior del programa.

```
const char dospuntos = ":",
```

Operación de escritura:

```
printf("%c", dospuntos);
```

Resultado:

:

Reglas precisas para declaración de constantes con nombres:

```
Declaración_de_constante ::=  
    const Tipo Nombre = Expresión_constante ;  
Tipo ::= Identificador  
Nombre ::= Identificador
```

Figure 10: Declaración de constantes

3.2 Variables:

El concepto de variable en programación imperativa es diferente del concepto de variables algebraica.

Las variables de un programa se designan mediante nombres o identificadores. El identificador de una variable representa el valor almacenado en dicha variable

Declaración de variable

En $C\pm$ cada variable debe tener asociado un tipo de valor determinado. Las variables han de ser declaradas en el programa antes de ser utilizadas. La declaración simple de una variable especifica su nombre y el tipo de valor asociado. Si varias variables tienen el mismo tipo se pueden declarar todas conjuntamente.

Ej: int edad;

```
int dia, mes, anno;
```

(Declaración *_de_variable* ::= Tipo Nombre {, Nombre };

El tipo declarado para cada una de las variables determina las operaciones que posteriormente se podrán realizar con ella. de la misma forma que sucedía para los valores literales. Para hacer operaciones con operandos de distintos tipos hay que hacer conversión de a un mismo tipo. ej. int() ó float()

Antes de usar la variable hay que declararla y asignarla un valor inicial.

```
Declaración_de_variable ::= Variable_simple | Lista_de_variables ;  
Variable_simple ::= Tipo Nombre [ = Expresión ]  
Lista_de_variables ::= Tipo Nombre { , Nombre }  
Tipo ::= Identificador  
Nombre ::= Identificador
```

Figure 11: Declaración de variables

3.3 Sentencia de asignación

Una forma de conseguir que una variable guarde un determinado valor es mediante una sentencia de asignación. Esta sentencia es característica de la programación imperativa, y permite inicializar una variable o modificar el

valor que tenía hasta el momento. El signo igual (=) es el operador de asignación. Este operador indica que el resultado de la expresión a su derecha debe ser asignado a la variable cuyo identificador está a su izquierda. Si intervienen variables en la expresión a la derecha de una sentencia de asignación, se usará el valor que tenga la variable en ese momento.

Un caso especial, que requiere cierta atención, es aquél en que a una variable se le asigna el valor de una expresión de la que forma parte la propia variable. Ej $\text{dias} = \text{dias} + 30$;

3.3.1 Sentencias de autoincremento y autodecremento

Sentencia autoincremento: $\text{variable} = \text{variable} + 1$;

utilizamos el símbolo ++

$\text{variable}++$;

Sentencia autodecremento $\text{variable} = \text{variable} - 1$;

lo escribimos –

$\text{variable}--$;

3.3.2 Compatibilidad de tipos

Aunque $C\pm$ no es un lenguaje *fuertemente tipado* y permite ambigüedad de asignación de tipos, utiliza un convenio de convertir previamente de manera automática el valor a asignar al tipo del valor de la variable.

De todas formas, en el *Manual de Estilo* se establece que para la realización de programas en $C\pm$ es obligatorio que se realice siempre una conversión explícita de tipos en estos casos.

Declaración:				
int posi, dato;				
float ejeX, ejeY;				
Traza de ejecución:				
ejeX	ejeY	dato	posi	Secuencia de sentencias
?	?	?	?	ejeX = 34.89;
34.89	?	?	?	dato = 67;
34.89	?	67	?	posi = int(ejeX) + dato;
34.89	?	67	101	dato = int(ejeY) + posi;
34.89	?	?	101	ejeY = ejeX + float(posi);
34.89	135.89	?	101	dato = posi / 5;
34.89	135.89	20	101	posi = posi % 5;
34.89	135.89	20	1	posi = posi * dato;
34.89	135.89	20	20	ejeY = ejeY/ejeX;
34.89	3.8948	20	20	ejeX = ejeX/2.0;
17.44	3.8948	20	20	posi = int(ejeY) - posi;
17.44	3.8948	20	-17	...

Figure 12: Declaración de variables

3.4 Operaciones de lectura simple

Los programas habituales suelen resolver problemas genéricos. Por ejemplo, obtener la suma de N números desde uno inicial a otro final, o calcular cada mes la nómina de cada uno de los empleados de una empresa. Un programa que produzca cada vez resultados diferentes deberá operar en cada caso a partir de unos datos distintos, y dichos datos no pueden ser, por tanto, valores constantes que formen parte del programa. Para resolver cada vez el problema concreto que se plantea, el programa debe leer como datos de entrada los Valores concretos a partir de los cuales hay que obtener el resultado. Por tanto las operaciones de lectura son fundamentales dentro de cualquier modelo de programación. En $C\pm$ los datos leídos han de ser guardados inmediatamente en variables del programa, por tanto, otra manera de asignar un valor a una variable es almacenar en ella un valor introducido desde el exterior del computador mediante el teclado u otro dispositivo de entrada de datos. Por defecto, el dispositivo de entrada suele estar asociado al teclado del terminal por el que se accede al computador. Estos procedimientos están incluidos también en el *módulo de librería* stdio.

3.4.1 El procedimiento scanf

El procedimiento scanf pertenece al módulo de librería stdio.

scanf(cadena-con-formatos, &variable1, &variable2. ... &variableN);

Es importante observar que los nombres de las variable a leer van precedidos del carácter: &

Declaración de variables	int mes, dia; float saldo;
Datos de entrada	123 4.5 6
Orden de lectura	scanf("%d %f %d", &mes, &saldo, &dia);
Resultado	mes = 123; saldo = 4.5; dia = 6;

Figure 13: Procedimiento scanf

De forma similar a printf los formatos numéricos pueden incluir la especificación del tamaño de dato. Pero a diferencia de printf en que ese tamaño era el mínimo número de caracteres a escribir, en scanf significa el tamaño máximo del dato de entrada a leer.

Datos	Formato	Dato leído	Datos restantes
12345xx	%d	12345	xx
12345xx	%3d	123	45xx
12xx345	%3d	12	xx345

Figure 14: Tamaño dr formato en scanf

3.4.2 Lectura interactiva

Cuando un programa se comunica con el usuario mediante un terminal de texto se suele programar cada operación de lectura inmediatamente después de una escritura en la que se indica qué dato es el que se solicita en cada momento.

Cualquier combinación es válida: una sentencia única para leer dos valores o dos sentencias para leer un valor cada una, e introducir los datos en una o dos líneas, indistintamente.

3.5 Estructura de un programa con declaraciones

```
Bloque ::= { Parte_declarativa Parte_ejecutiva }
Parte_declarativa ::= { Declaración }
Parte_ejecutiva ::= { Sentencia }
Declaración ::=
    Declaración_de_constante | Declaración_de_variable | ...
Sentencia ::= Llamada_a_procedimiento | Asignación | ...
```

Figure 15: Programa con declaraciones

El contenido de un bloque se organizará en dos partes. La primera de ellas contendrá todas las declaraciones de constantes, variables, etc., y la segunda incluirá las sentencia..., ejecutables correspondientes a las acciones a realizar. Las declaraciones pueden hacerse en el orden que se quiera, con la limitación de que cada nombre debe ser declarado antes de ser usado. Las sentencias ejecutables deben escribirse exactamente en el orden en que han de ser ejecutadas.

3.6 Ejemplos de programas

4 Metodología de Desarrollo de Programas (I)

Aquí se tratan explícitamente los primeros conceptos metodológicos relacionados con la programación en general y con la programación imperativa en particular. Se presenta el desarrollo por refinamientos sucesivos aunque

limitado al empleo de la estructura secuencial.

La labor de programación puede considerarse como un caso particular de la resolución de problemas.

Un programa en el modelo de programación imperativa, se expresa como una serie de instrucciones u órdenes que gobiernan el funcionamiento de una máquina. La máquina va ejecutando dichas instrucciones en el orden preciso que se indique. La tarea de desarrollar dicho programa equivale, por tanto, a la de expresar la estrategia de resolución del problema en los términos del lenguaje de programación utilizado.

4.1 Descomposición en subproblemas

Cualquier problema de cierta complejidad necesitará una labor de desarrollo para expresar la solución. El método más general de resolución de problemas no triviales consiste en descomponer el problema original en subproblemas más sencillos, continuando el proceso hasta llegar a subproblemas que puedan ser resueltos de forma directa. La resolución de problemas en que la estrategia de solución consiste en realizar acciones sucesivas. Según esta idea, para desarrollar la estrategia de resolución, habrá que ir identificando subproblemas que se resolverán ejecutando acciones cada vez más simples.

4.2 Desarrollo por refinamientos sucesivos

Esta técnica es parte de las recomendaciones de una metodología general de desarrollo de programación denominada *programación estructurada*. La técnica de refinamientos consiste en expresar inicialmente el programa a desarrollar como una acción global, que si es necesario se irá descomponiendo en acciones más sencillas hasta llegar a acciones simples que puedan ser expresadas directamente como sentencias del lenguaje de programación. Cada paso de refinamiento consiste en descomponer cada acción compleja en otras más simples. Esto exige: - identificar las acciones componentes - identificar la manera de combinar las acciones componentes para conseguir el efecto global.

La forma en que varias acciones se combinan en una acción compuesta constituye el esquema de la acción compuesta.

4.2.1 Desarrollo de un esquema secuencial

Para desarrollar una acción compuesta según un esquema secuencial se necesitará:

(a) Identificar las acciones componentes de la secuencia. Identificar las variables necesarias para disponer de la información adecuada al comienzo.

de cada acción, y almacenar el resultado. (b) Identificar el orden en que deben ejecutarse las acciones componentes.

4.3 Aspectos de estilo

Una buena metodología de desarrollo de programas debe atender no sólo a cómo se van refinando las sucesivas acciones, sino a cómo se expresan las acciones finales en el lenguaje de programación. El estilo de redacción del programa en su forma final es algo fundamental para conseguir que sea claro y fácilmente comprensible por parte de quienes hayan de leerlo.

4.3.1 Encolumnado

Un recurso de estilo de presentación es el *encolumnado* o sangrado (*indent*). Ampliando el margen izquierdo para las partes internas del programa.

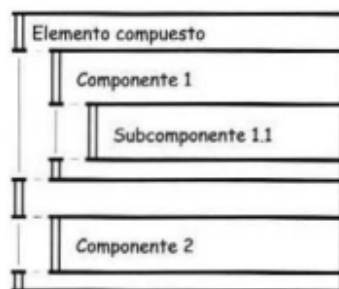


Figure 16: Encolumnado de elementos compuestos

4.3.2 Comentarios. Documentación

Otro recurso utilizable para mejorar la claridad de un programa es el empleo de comentarios. Es aconsejable seguir ciertas pautas para facilitar la lectura del programa, estas pautas corresponden a diferentes clases de comentarios, cada uno con un propósito diferente;

- Cabeceras de programa: a tiene como finalidad documentar el programa como un todo.
- Cabeceras de sección: Las cabeceras de sección sirven para documentar partes importantes de un programa relativamente largo. Al igual que la cabecera del programa, se presentan en forma de "caja" al comienzo de la sección correspondientes, ocupando todo el ancho del listado.
- Comentarios-orden: son un elemento metodológico fundamental, y sirven para documentar los refinamientos empleados en el desarrollo del programa.
- Comentarios al margen: Estos comentarios sirven para aclarar el significado de ciertas sentencias del programa, que pueden ser difíciles de interpretar al leerlas tal como aparecen escritas en el lenguaje de programación empleado. Una recomendación de estilo es situar estos comentarios hacia la parte derecha del listado, en las mismas líneas que las sentencias que se comentan. Los comentarios al margen se utilizan muchísimo para explicar el significado de cada variable usada en un programa, poniéndolos en la misma línea en que se declaran.

```

/*****
 * Programa: Silla
 *
 * Descripción:
 *   Este programa imprime de forma esquemática la silueta
 *   de una silla usando caracteres normales de la impresora.
 *****/

/*=====
   DIRECTIVA DE COMPILACIÓN
   =====*/
#include <stdio.h>

/*=====
   PARTE EJECUTABLE DEL PROGRAMA
   =====*/
int main() {
    /*-- Imprimir el respaldo --*/ {
        printf( "!\\n" );
        printf( "!\\n" );
    }
    /*-- Imprimir el asiento --*/ {
        printf( "====\\n" );
    }
    /*-- Imprimir las patas --*/ {
        printf( "!    !\\n" );
        printf( "!    !\\n" );
    }
}

```

Figure 17: Programa con todos los tipos de comentarios

4.3.3 Elección de nombres

Los nombres que tenga que inventar el programador deben ser elegidos con un criterio nemotécnico, o sea manera que recuerden fácilmente el significado de los elementos nombrado. Para que los nombres o identificadores resulten significativos hay que procurar que tengan la categoría gramatical adecuada al elemento nombrado. En concreto:

- Los valores (constantes, variables, etc.) deben ser designados mediante sustantivos.
- Las acciones (procedimientos, etc.) deben ser designadas con verbos.
- Los tipos deben ser designados mediante nombres genéricos.

```

/*****
* Programa: CalcularDias
*
* Descripción:
* Este programa calcula los días que faltan para el
* cumpleaños de una persona.
*****/
#include <stdio.h>
#include "fechas.h"

/*=====
En el módulo fechas.h están definidos:

T_fecha = Tipo de valor FECHA
LeerFecha = Procedimiento para leer una fecha
EscribirFecha = Procedimiento para escribir una fecha
DiasEntre = Función para calcular los días que hay
entre dos fechas
Hoy = Variable en la que se mantiene actualizada la
fecha de hoy
=====*/

int main() {

    T_fecha fechaCumple; /* cumpleaños */
    T_fecha fechaHoy;     /* fecha de hoy */
    int dias;             /* días que faltan */

    /*-- Obtener la fecha del cumpleaños --*/ {
        printf( "¿Cuál es tu próximo cumpleaños? " );
        LeerFecha( fechaCumple );
    }
    /*-- Obtener la fecha de hoy --*/ {
        fechaHoy = Hoy;
    }
    /*-- Calcular los días que faltan para el cumpleaños --*/ {
        dias = DiasEntre( fechaHoy, fechaCumple );
    }
    /*-- Imprimir el resultado --*/ {
        printf( "\nFaltan%4d", dias);
        printf( " días para tu cumpleaños" );
    }
}

```

Figure 18: Elección de nombres

4.3.4 Uso de mayúsculas y minúsculas

Los lenguajes de programación que permiten distinguir entre letras mayúsculas y minúsculas facilitan la construcción de nombres en programas largos, en que es preciso inventar un gran número de ellos. Ej: Los nombres de tipos, procedimientos y funciones empiezan por mayúscula.

Los nombres de variables y constantes empiezan por minúscula.

Los nombres que son palabras compuestas usan mayúsculas intercaladas al comienzo de cada siguiente palabra componente.

4.3.5 Constantes con nombre

La posibilidad de declarar constantes con nombres simbólicos puede aprovecharse para mejorar la claridad del programa. En lugar de usar directamente valores numéricos en las expresiones de algunos cálculos, pueden resultar ventajoso definir determinados coeficientes o factores de conversión con un nombre simbólico que tenga un buen significado nemotécnico, y usar la constante con ese nombre en los cálculos. Otra forma ventajosa de usar el mecanismo de definición de constantes con nombre se da en el caso de que el comportamiento de un programa venga dado en función de ciertos valores generales, fijos, pero que quizá fue interesante cambiarlos en el futuro. Este tipo de valores se denominan a veces parámetros del programa, y es conveniente que su valor aparezca escrito sólo una vez en un lugar destacado del programa.

4.4 Ejemplos de programas (105)

5 Estructuras Básicas de la Programación Imperativa

Secuencia, Selección e Iteración.

IF-THEN-ELSE y WHILE.

la programación estructurada es una metodología de programación que fundamentalmente trata de construir programas que sean fácilmente comprensibles.

Esta metodología está basada en la técnica de desarrollo de programas por refinamientos sucesivos: se plantea la operación global a realizar por el programa, y se descompone en otras más sencilla... A su vez, estas últimas vuelven a ser descompuestas nuevamente en otras todavía más elementales. Este proceso de descomposición continúa hasta que todo se puede escribir utilizando las estructuras básicas disponibles en el lenguaje de programación que se está empleando.

5.1 Representación de la estructura de un programa

La estructura de los programas imperativos se representa tradicionalmente mediante diagramas de flujo. Estos diagramas contienen dos elementos básicos, correspondientes a acciones y condiciones. Las acciones representadas mediante rectángulos, las condiciones mediante rombos. Las condiciones equivalen a preguntas a las que se puede responder si o no.

La programación estructurada recomienda descomponer las acciones usando las estructuras más sencillas posibles. Entre ellas se reconocen tres estructuras básicas, que son: Secuencia, Selección e Iteración. Estas tres estructuras están disponibles en todos los lenguajes modernos de programación imperativa en forma de sentencias del. Combinando unos esquemas con otros se pueden llegar a construir programas con una estructura tan complicada como sea necesario.

6 Metodología de desarrollo de programas II

Ampliación de la metodología por refinamientos sucesivos con la posibilidad de usar subprogramas como técnica de abstracción.

Las funciones y procedimientos introducen la posibilidad de descomposición de un problema en subproblemas realmente independientes.

Una abstracción es una visión simplificada de una cierta entidad, de la que sólo consideramos sus elementos esenciales, prescindiendo en lo posible de los detalles. Las entidades que podemos abstraer para materializarlas como subprogramas son, en general, operaciones. Con la palabra operación englobamos tanto la idea de acción como la de función.

Especificación y realización

Al plantear operaciones abstractas habremos de definir dos posibles visiones. La visión abstracta o simplificada, y la visión detallada, completa. La visión abstracta es la que permite usar dicha operación sin más que conocer qué hace dicha operación. La visión detallada es la que define cómo se hace dicha operación, y permite que el procesador la ejecute. La primera visión representa el punto de vista de quienes han de utilizar la operación. Se dice que esa visión abstracta es la especificación o interfaz de la operación. La visión detallada representa el punto de vista de quien ha de ejecutar dicha acción, y se dice que expresa su realización o implementación.

Especificación: qué hace esta operación (punto de vista de quien lo invoca)

Realización: cómo lo hace (punto de vista de quien lo ejecuta)

En su forma más sencilla la especificación o interfaz consiste simplemente en indicar cuál es el nombre de la operación y cuáles son sus argumentos. En C++ la especificación puede ser simplemente una cabecera de subprograma. Esa forma simplificada de especificación indica solamente cuál ha de ser la sintaxis o forma de uso de la operación. La especificación completa debe establecer también cuál es la semántica o significado de la operación. Para ello podemos añadir un comentario en que se indique qué relación hay entre los argumentos y el resultado de la operación.

La realización, por su parte, debe suministrar toda la información necesaria para poder ejecutar la operación. En C++ la realización o implementación será la definición completa del subprograma, en forma de bloque de código.

Con ello se pone de manifiesto la idea de que la especificación es una visión abstracta de qué hace la función, con independencia de los detalles de cómo lo hace. Precisamente las reglas de visibilidad de C++ permiten usar

subprogramas como operaciones abstractas, con ocultación de los detalles de realización.

Es importante comprender que si describimos la semántica en lenguaje humano, impreciso, tendremos sólo una especificación informal. Si se necesita mayor rigor se puede recurrir a expresiones lógico-matemáticas para especificar formalmente las condiciones que relacionan los datos de entrada y los resultados. La especificación formal evita ambigüedades, pero también suele resultar más costosa de escribir y más difícil de leer. Por ello conviene que vaya acompañada siempre de una especificación en lenguaje humano.

Podemos anotar un código con la PRECONDICIÓN y POSTCONDICIÓN correspondientes a su especificación formal.

En programación la idea de función surge al aplicar el concepto de abstracción a las expresiones aritméticas. Una expresión representa un nuevo valor obtenido por cálculo a partir de ciertos valores ya conocidos que se usan como operandos.

Si buscamos que el concepto de función en programación se aproxime al concepto matemático de función, el paso de argumentos debería ser siempre por valor. El concepto matemático de función es una aplicación entre conjuntos, cuyo cómputo se limita a suministrar un resultado, sin modificar el valor de los argumentos. Aunque algunas veces, por razones de eficiencia, pueda ser aconsejable pasar por referencia argumento de funciones, seguirá siendo deseable, para mantener la máxima claridad en el programa, que la llamada a la función no modifique el valor de los argumentos. Desde el punto de vista de claridad del programa, y con independencia de cuál sea el mecanismo de paso de argumentos empleado, la cualidad más deseable al utilizar funciones es conseguir su transparencia referencial. Tal como se mencionó anteriormente, la transparencia referencial significa que la función devolverá siempre el mismo resultado cada vez que se la invoque con los mismos argumentos.

La transparencia referencial se garantiza si la realización de la función no utiliza datos exteriores a ella, si no se emplea:

- Variables externas al subprograma, a las que se accede directamente por su nombre, de acuerdo con las reglas de visibilidad de bloques.
- Datos procedentes del exterior, obtenidos con sentencias de lectura.
- Llamadas a otras funciones o procedimientos que no posean transparencia referencial. Las sentencias de lectura son en realidad un caso particular de éste

Las funciones que cumplen la cualidad de transparencia referencial se denominan funciones puras.

Acciones abstractas. Procedimientos

Las funciones las podemos considerar como expresiones abstractas parametrizadas, así los procedimientos los podemos considerar como acciones abstractas parametrizadas.

Procedimiento: acción que se define por separado y se invoca con su nombre.

Como acción abstracta podemos tener dos visiones de un procedimiento: especificación, formada por la cabecera y una descripción de qué hace dicho procedimiento que sería la realización, en que se detalla codificada en el lenguaje de programación elegido, cómo se hace la acción definida.

En programación imperativa las acciones consisten habitualmente en modificar los valores de determinadas variables. Por esta razón se considera normal que los procedimientos usen argumentos pasados por referencia.

De todas maneras conviene seguir una cierta disciplina para que los programas resulten claros y fáciles de entender. Para ello podemos recomendar que los procedimientos se escriban siempre como procedimientos puros, entendiendo por ello que no produzcan efectos laterales o secundarios. Con esto se consigue que la acción que realiza un procedimiento se deduzca en forma inmediata de invocación de dicha acción. Se garantiza que un procedimiento cumple con esta cualidad si su realización no utiliza:

- Variables externas al subprograma, a la que se accede directamente por su nombre, de acuerdo con las reglas de visibilidad de bloques.
- Llamadas a otros subprogramas que no sean procedimientos o funciones puras.

6.1 Desarrollo usando abstracciones

La metodología de programación estructurada puede ampliarse con la posibilidad de definir operaciones abstractas mediante subprogramas. A continuación se describen dos estrategias de desarrollo diferentes, según qué se escriba primero, si la definición de los subprogramas, o el programa principal que los utiliza.

6.1.1 Desarrollo descendente

La estrategia de desarrollo descendente (Top-Down), es simplemente el desarrollo por refinamientos sucesivos, teniendo en cuenta además la posibilidad de definir operaciones abstractas. En cada etapa de refinamiento de una operación habrá que optar por una de las alternativas siguientes:

- Considerar la operación como operación terminal, y codificarla mediante sentencias del lenguaje de programación.
- Considerar la operación como operación compleja, y descomponerla en otras más sencillas.
- Considerar la operación como operación abstracta, y especificarla, escribiendo más adelante el subprograma que la realiza.

Hay que evaluar si una operación debe refinarse como operación abstracta. En general resultará ventajoso refinar una operación como operación abstracta si:

- Evita mezclar en un determinado fragmento de programa operaciones con un nivel de detalle muy diferente.
- Evitar escribir repetidamente fragmentos de código que realicen operaciones análogas.

Hay que decir que esto implica un costo ligeramente mayor en términos de eficiencia, ya que siempre se ejecuta más rápidamente una operación si se escriben directamente las sentencias que la realizan, que si se invoca el subprograma que contiene dichas sentencias. La llamada al subprograma representa una acción adicional que consume un cierto tiempo de ejecución.

por otro lado, hay un aumento de eficiencia en ocupación de memoria si se codifica como subprograma una operación que se invoca varias veces en distintos puntos del programa. En este caso el código de la operación aparece solo una vez, mientras que si se escribiesen cada vez las sentencias equivalentes el código aparecería repetido varias veces.

6.2 Reutilización

la realización de ciertas operaciones como subprogramas independientes facilita lo que se llama reutilización de software. Si la operación identificada como operación abstracta tiene un cierto sentido en sí misma, es muy posible que resulte de utilidad en otros programas.

La escritura de otros programas que utilicen esa misma operación resulta más sencilla, ya que se aprovecha el código de su definición, que ya estaba escrito.

Para aplicar las técnicas de desarrollo por reutilización de software es preciso pensar en las posibles aplicaciones de un cierto subprograma en el momento de especificarlo, con independencia de las necesidades particulares del programa que se está desarrollando en ese momento.

Esta estrategia de desarrollo tiene ventajas e inconvenientes. La principal ventaja es que se amplía el conjunto de aplicaciones en que se podrá reutilizar adelante el subprograma que se está desarrollando ahora. Su principal inconveniente es que será más costoso hacer el desarrollo del subprograma planteado como operación de uso general, que planteado como operación particular, hecha a medida del programa que lo utiliza en este momento.

6.3 Desarrollo ascendente

La metodología de desarrollo ascendente (en inglés Bottom-Up) consiste en ir creando subprogramas que realicen operaciones significativas de utilidad para el programa que se intenta construir, hasta que finalmente sea posible escribir el programa principal.

La técnica tiene una cierta analogía con el desarrollo de subprogramas pensando en su reutilización posterior. Al hablar de desarrollo para reutilización se ha dicho que los subprogramas podían surgir en el proceso de refinamiento de un programa concreto, al identificar ciertas operaciones, pero debían definirse: pensando en

futuras aplicaciones. En este caso se trata de que la identificación de las operaciones no surja de un proceso de descomposición o refinamiento de alguna acción en particular, sino simplemente pensando en el programa que se desarrolla, casi como una más de las posibles aplicaciones futuras.

6.4 Programas robustos

La corrección de un programa exige que los resultados sean los esperados. siempre que el programa se ejecute con unos datos de entrada aceptables. La cuestión que nos ocupa en este momento es: ¿cuál debe ser el comportamiento del programa si los datos son incorrectos?

Es frecuente que un programa se escriba sin tener en cuenta la posibilidad de que los datos no sean los esperados, pues con ello se simplifica su desarrollo. Sin embargo esta postura no es admisible en la práctica.

La llamada **programación a la defensiva** (en inglés, defensive programming) con que cada programa o subprograma esté escrito de manera que desconfíe sistemáticamente de los datos o argumentos con que se le invoca, y devuelva siempre como resultado:

- a) El resultado correcto, si los datos son admisibles, o bien
- b) Una indicación precisa de error, si los datos no son admisibles

Lo que no debe hacer nunca el programa es devolver un resultado como si fuera normal, cuando en realidad es erróneo, ni "abortar". Esto da lugar a una propagación de errores, que puede aumentar la gravedad de las consecuencias, y hacer que la identificación del fallo del programa resulte mucho más difícil, ya que el efecto se puede manifestar sólo más adelante, en otra parte del programación sin relación aparente con la que falló.

La mejora de la robustez del programa tiene como contrapartida una cierta pérdida de eficiencia, al tener que hacer comprobaciones adicionales.

Tratamiento de excepciones:

Ante la posibilidad de errores en los datos con que se opera, hay que considerar dos actividades diferentes:

1. Detección de la situación de error
2. Corrección de la situación de error

Si una operación se ha escrito como subprograma, la programación a la defensiva recomienda que la primera actividad (detección del posible error) se haga dentro del subprograma, sin confiar en que quienes usen el subprograma invoquen siempre con datos correctos.

Existen varios esquemas de programación posibles para tratamiento de errores modelo recomendado es el modelo de terminación. En este modelo se detecta un error en una sección o bloque del programa, la acción de tratamiento del error reemplaza al resto de las acciones pendientes de dicha sección, con lo cual tras la acción correctora se da por terminado el bloque. En algunos lenguajes de programación, tales como el lenguaje Ada, Java y C++, existen construcciones o sentencias adecuadas para programar este esquema.

En C++ la sentencia throw provoca la terminación del subprograma de manera semejante a una sentencia return. Sin embargo, ambas terminaciones son distintas: con return se realiza una terminación normal y con throw se realiza una terminación por excepción. La sentencia throw puede devolver cualquier tipo de resultado en excepción, además, la sentencia throw es la encargada de indicar que se ha detectado una situación de error, lanzar el mecanismo de tratamiento de excepciones. Quien utiliza el subprograma será encargado de realizar la corrección de la situación de error

7 tipos definidos

primero tipos escalares simples definidos por enumerados, el caso de bool.

Luego definición de tipos estructurados y las dos formas mas importantes: array o formación (ej vectores o cadenas de caracteres) y struct o registro.

Una de las ventajas fundamentales de los lenguajes de alto nivel es la posibilidad que ofrecen al programador de definir sus propios tipos de datos.

Mediante la definición de nuevos tipos de datos por el programador se consigue que cada información que maneja el computador tenga su sentido específico. El tipo establece los posibles valores que puede tomar ese dato. Además, al igual que sucedía con los tipos predefinidos, a cada nuevo tipo que se define se asocian un conjunto de opera-

ciones que se pueden realizar con él.

La declaración en C siempre se hace junto con la declaración de variables y constantes, cada nuevo tipo siempre se inicia con la palabra clave **typedef**

En estas declaraciones se definen nuevos tipos dándoles un nombre o identificador y haciéndolos equivalentes o sinónimos de otros tipos ya definidos (en este caso, los predefinidos `int`, `char`, `float`)

Es importante señalar que igual que se han utilizado los tipos predefinidos, en la definición de un nuevo tipo se pueden utilizar (y normalmente se utilizan) otros tipos definidos previamente, según veremos a lo largo de este tema. Precisamente esta característica es la más importante de la posibilidad de declarar nuevos tipos.

La definición de tipos es solamente una declaración de los esquemas de datos que se necesitan para organizar la información de un programa. Para almacenar información es necesario declarar y utilizar variables de los correspondientes tipos, de la misma forma que se hace con los tipos predefinidos

Declaración de tipos, tenemos: tipo sinónimo, tipo enumerado, tipo array, tipo struct, tipo unión y tipo puntero. tipo sinónimo puede parecer trivial o meramente teórico, sin embargo, tiene una utilidad bastante importante como mecanismo de parametrización del programa.

(`typedef int entero`)

7.1 Definición de tipos enumerados

Una manera sencilla de definir un nuevo tipo de dato es enumerar todos los posibles valores que puede tomar. En C el nuevo tipo enumerado se define detrás de la palabra clave `enum` y a continuación se detalla la lista con los valores entre llaves separados por comas: `typedef enum TipoDia {lunes, martes, miercoles, jueves, viernes, sabado, domingo }`

Cada posible valor se describe mediante un identificador. Los identificadores al mismo tiempo quedan declarados como valores constantes.

La enumeración implica un orden que se establece entre los valores enumerados.

Uso de tipos enumerados Los tipos enumerados se emplean de manera similar a los tipos predefinidos. El identificador de tipo se puede emplear para definir variables de ese tipo y los identificadores de los valores enumerados se emplean como las constantes con nombre.

Al igual que para el resto de los tipos ordinales, con los tipos enumerados se puede utilizar la notación `int(e)` para obtener la posición de un valor en la lista de valores del tipo.

la operación inversa, que permita conocer qué valor enumerado ocupa una determinada posición, se consigue mediante la notación inversa que hace uso del identificador del tipo enumerado y se invoca: `TipoEnumerado(N)`

7.2 Tipo predefinido bool

El tipo predefinido `bool` responde a la siguiente definición de tipo enumerado.

```
typedef enum bool {false, true};  
int(false)==0  
int(true)==1
```

El tipo booleano, como cualquier otro tipo enumerado, se puede pasar Como argumento de un procedimiento o función y puede ser devuelto como resultado de una función. De hecho es frecuente definir funciones cuyo resultado es un valor booleano cuando se quiere realizar un test sobre los argumentos de la función. Este tipo de funciones se denomina Con la librería `<ctype.h>` en C tenemos:

`bool isalpha(char c)` indica si `c` es una letra

`bool isascii(char c)` indica si `c` es un carácter ASCII

`bool isblank(char c)` indica si `c` es un espacio o tabulación

`bool isdigit(char c)` indica si `c` es un dígito decimal (0-9)

`bool islower(char c)` indica si `c` es una letra minúscula.

`bool isspace(char c)` indica si `c` es un blanco o salto de línea o página

`bool isupper (char c)` indica si `c` e una letra mayúscula.

7.3 Tipos estructurados

Hasta ahora todos los tipos de datos presentados hasta este momento se denominan tipos escalares, y son datos simples, en el sentido de que no se pueden descomponer.

En muchas aplicaciones resulta conveniente, o incluso necesario, manejar globalmente elementos de información que agrupan colecciones de datos. Por ejemplo, puede ser apropiado manejar como un dato único el valor de una fecha que incluye la información del día, el mes y el año como elementos componentes separados. Con este objetivo, los lenguajes de programación dan la posibilidad de definir tipos de datos estructurados. Un tipo estructurado de datos, o estructura de datos, es un tipo cuyos valores construyen agrupando datos de otros tipos más sencillos. Los elementos de información que integran un valor estructurado se denominan componentes. Todos los tipos estructurados se definen, en último término, a partir de tipos simples combinados.

7.4 Tipo formación (array)

Las estructuras formaciones o array permiten la generalización de la declaración, la referencia y la manipulación de colecciones de datos todos del mismo tipo. La formación mas elemental son los vectores.

7.4.1 Tipo vector

un vector está constituido por una serie de valores, todos ellos del mismo tipo, a los que se les da un nombre común que identifica a toda la estructura globalmente. $V = (V_0, V_1, V_2, \dots, V_{n-2}, V_{n-1})$

En cuanto al aspecto de programación se puede establecer un paralelismo entre la estructura de programación en la que se repite la misma acción un número de veces determinado: sentencia for, y la estructura de datos vector en la que también se repiten un número de veces determinado el mismo tipo de dato. Como se verá posteriormente, la sentencia for es la que mejor se adecúa al manejo de los vectores y en general de todo tipo de formaciones.

Declaración de vectores

```
typedef TipoVector[ NumeroElementos];
```

En muchos casos el tamaño del vector es un parámetro del programa que podría tener que cambiarse al adaptarlo a nuevas necesidades. Si es así, resulta aconsejable que la declaración del número de elementos se realice como una constante con nombre.

De esta manera, el programa queda pararnetrizado por dichas constantes. En las modificaciones posteriores, si se quiere adaptar el tamaño del vector sólo es necesario modificar esta constante. Además, como se verá posteriormente, es habitual utilizar el número de elementos del vector en las operaciones de recorrido y búsqueda de los vectores, que se pueden entonces programar en función de la misma constante.

En C es obligatorio declarar todas las variables, está expésamente prohibido la declaración de variables de tipo anónimo. Se dice que una vairable es de *tipo anónimo* cuando su estructura se detalla es una misma declaración de la variable, como si se estuviera declarando un tipo de datos anónimos para esta única variable.

La sintaxis de la declaración de los tipos array es la siguiente: Tipoformación::= typedef IdentificadordeTipoElemento IdentificadosdeTipoDimensiones;

Dimensiones::=Tamaño {Tamaño}

Tamaño ::= [NumerodeElementos]

La referencia a un elemento concreto de un vector se hace mediante el nombre del vector seguido, entre corchetes, del Índice del elemento referenciado.

la comprobación de que el Índice para acceder a un elemento de vector está dentro del rango permitido es responsabilidad del programador. Muchos ataques informáticos aprovechan la falta de previsión de esta comprobación para alterar el funcionamiento normal de un programa suministrándole datos de mayor tamaño que el previsto y provocar lo que se denomina en inglés buffer overrun.

En lenguajes tales como Pascal, Modula-2, Ada, etc. existe la posibilidad de realizar una asignación global de un vector a otro, siempre que estos sean compatibles entre sí.

Sin embaro, en C± no existe esta posibilidad y la asignación se tiene que programar explícitamente mediante un bucle que realice la copia elemento a elemento.

El modo por defecto de paso de argumentos de tipo formación, y más concretamente de tipo vector, es el paso por referencia.

```
void LeerVector(TipoVector v){...}
```

```
void ConocerEstado (TipoEstados e) {...}
```

Cuando se invocan estos procedimientos, además de emplear en la llamada argumentos reales que deben ser compatibles con el correspondiente argumento formal, hay que tener en cuenta que el argumento real puede ser modificado.

hay que recordar que en $C\pm$, exceptuando el caso de las formaciones, cuando un argumento formal se quiere pasar por referencia debe ir precedido del símbolo & en la cabecera de declaración del subprograma.

Cuando se utilizan argumentos de tipo formación y no se quiere que se modifiquen los parámetros reales en la llamada al procedimiento, los argumentos formales deben ir precedidos de la palabra clave **const**

Ej: **void** EscribirVector(**const** TipoVector v){...}

De manera que si se ejecuta: EscribirVector(vectorDos); las variables vectorDos permanecerán inalteradas.

Por tanto, cuando declaramos argumento de tipo formación o vector precedido de la palabra clave const equivalente al paso de dicho argumento por valor.

Para ser exactos, esta forma de paso de argumentos es más restrictiva que paso por valor. En realidad el vector se pasa por referencia, pero se prohíbe usar asignaciones a sus elementos en el cuerpo del subprograma. En el paso por valor de otros tipos de datos el argumento formal se ve como variable local dentro del subprograma, y de hecho es una copia que puede modificarse.

sin alterar el argumento real usado en la llamada. En cambio los argumentos de tipo vector declarados como const se ven como constantes dentro del subprograma, y sus elementos no pueden ser modificados en modo alguno . (dif entre argumentos formales y reales, los strings funcionan como vectores abiertos)

7.5 Vectores de caracteres: Cadena(string)

Esto se debe a que en realidad las cadenas son vectores de caracteres y los vectores no han sido estudiados hasta este tema.

Sin embargo, en todos los lenguajes es habitual que las cadenas de caracteres tengann ciertas peculiaridades que no tienen el resto de los vectores y por esta razón son objeto de este apartado específico. Cualquier vector de la forma: **typedef char** Nombre[N]

Una cadena de caracteres (en inglés string) es un vector en el que se pueden almacenar textos de diferentes longitudes (si caben). Para distinguir la longitud útil en cada momento se reserva siempre espacio para un carácter más, y se hace que toda cadena termine con un carácter nulo "barra0" situado al final. El caracter nulo no se puede escribir, va en el utimo caracter. Librería < string.h > con strcpy(c1,c2), strcat(c1,c2), strlen(c1), strcmp(c1,c2)

Al igual que con cualquier otro tipo de vector, no es posible realizar asignaciones globales entre cadenas. Para realizar csta operación usando el procedimiento general se programaria un for que copie elemento a elemento.

En el procedimiento de escritura printf se pueden utilizar constantes o variables empleando el formato de escritura %s para string o cadena.

7.6 Tipo tupla y su necesidad

Otra forma de construir un dato estructurado consiste en agrupar elementos de información usando el esquema de *tupla* o *agregado*. En este esquema el dato estructurado está formado por una colección de componentes, cada uno de los cuales puede ser de un tipo diferente. Una tupla, como cualquier otro dato compuesto, puede verse de forma abstracta como un todo, prescindiendo del detalle de sus componentes. La posibilidad de hacer referencia a toda la colección de elementos mediante un nombre único correspondiente al dato compuesto simplifica al muchos casos la escritura del programa

7.7 Tipo registro (struct)

Los esquemas de tupla pueden usarse en C definiéndolos como struct. Un registro o struct es una estructura de datos fomrada por una olección de elementos de informción llamados campos.

La declaración de un tipo registro en C se hace utilizando la palabra clave struct de la siguiente forma:

typedef struct Tipo-registro { ... } Como siempre, para declarar variables de tipo registro es necesario haber realizado previamente la definición del tipo del registro. Al manejar datos estructurados de tipo registro se dispone de dos posibilidades: operar con el dato completo, o bien operar con cada campo por separado. Las posibilidades de operar con el dato completo son bastante limitadas. La única operación admisible es la de asignación.

Las operaciones de tratamiento de estructuras registro consisten normalmente en operar con sus campos por separado. La forma de hacer referencia a un campo es mediante la notación: registro.campo Cada campo se puede usar como cualquier otro dato del correspondiente tipo.

8 Ampliación de estructuras de control

Antes de pasar al estudio de las estructuras de datos complejas, se completa el repertorio de estructuras de control más frecuentes en los lenguajes imperativos.

8.1 Sentencia Do (while)

Su estructura es:

```
do { Acción };  
while (Condición);
```

Una situación típica en que resulta cómodo el empleo de esta sentencia es la que se produce cuando al finalizar cada iteración se pregunta al operador si desea continuar con una nueva. En todos estos casos, el programa siempre ejecuta la primera iteración y pregunta si se desea o no realizar otra más. También resulta adecuado el empleo de la repetición cuando solamente son válidos unos valores concretos para una determinada respuesta. Si la respuesta es correcta se solicitará de nuevo y no se continuará hasta obtener una respuesta dentro de los valores válidos.

En general, es aconsejable su uso cuando se sepa que al menos es necesaria una iteración y por tanto utilizando la sentencia while es necesario forzar las condiciones para que dicha iteración se produzca.

8.2 Sentencia continue

La sentencia continue dentro de cualquier bucle (for, while, do), finaliza la iteración en curso e inicia la siguiente iteración. A veces, dependiendo de la evolución de los cálculos realizados en una iteración, no tiene sentido completar la iteración que se está realizando y resulta más adecuado iniciar una nueva. Esto puede suceder cuando alguno de los datos suministrados para realizar un cálculo es erróneo y puede dar lugar a una operación imposible (división por cero, raíz de un número negativo, etc.). En este caso lo adecuado es detectar la situación y dar por finalizada la iteración e iniciar una nueva iteración con nuevos datos de partida. La sentencia continue siempre estará incluida dentro de otra sentencia condicional

8.3 Estructuras complementarias de condición

La falta de claridad cuando se utilizan varias selecciones anidadas aconseja disponer de una sentencia de selección en cascada (else if). Por las mismas razones de claridad y sencillez es habitual disponer de una sentencia que permite una selección por casos.

Sentencia Switch

Cuando la selección entre varios casos alternativos depende del valor que toma determinada variable o del resultado final de una expresión, es necesario realizar comparaciones de esa misma variable o expresión con todos los valores que puede tomar, uno por uno, para decidir el camino a elegir.

Si lo que se necesita es comparar el resultado de una expresión, dicha expresión se reevaluará tantas veces como comparaciones se deben realizar (a menos que disponga de un compilador optimizante). En este caso y por razones de simplicidad y eficiencia es aconsejable guardar el resultado de la expresión en una variable auxiliar y realizar las comparaciones con dicha variable utilizando el esquema de selección por casos. 260.

Si el tipo de valor determina la selección es un tipo ordinal: int, char o enumerado, se dispone de la sentencia **switch** cuya estructura permite agrupar los casos que tienen el mismo tratamiento y en el que se evalúa solamente una vez la expresión x. las distintas vías de ejecución se asocian a grupos de valores que puedan tomar la expresión o variable x.

```
switch(expresión){  
case valor1:  
acciónA;  
break;  
case valor2:  
case valor3:  
case valor4:  
acciónB; break;  
.....  
default:  
}
```

El Manual de Estilo para C \pm impone que cada acción finaliza siempre con la sentencia break para que finalice

la sentencia switch después de cada acción. Si por error se omite la sentencia break, se continuaría ejecutando la acción del siguiente o siguientes casos. Esta sentencia no se puede utilizar cuando la variable o el resultado de la expresión que controla la selección sea de tipo float u otro tipo no simple. En estos casos no queda más remedio que emplear la sentencia de selección general o en cascada.

En la sentencia switch se deben incluir todos los posibles valores que tomar la variable o expresión. Cuando se obtiene un valor que no está asociado a ninguna vía (y no hay alternativa default), el programa finaliza por error. Si lo que sucede es que existen valores para los que no se debe realizar ninguna acción, entonces estos valores se deben declarar asociados a una acción vacía. Otra forma de conseguir esto mismo es mediante alternativa default vacía utilizando sólo un punto y coma (;)

8.4 Equivalencia entre estructuras

Las estructuras básicas estrictamente necesarias para la programación estructurada son la selección entre dos alternativas if y iteración while. Éstas se pueden considerar las estructuras primarias. El resto, que denominaremos estructuras secundarias, son en general más complejas y tienen como objetivo lograr programas más sencillos en situaciones particulares.

Cualquiera de las estructuras secundarias siempre puede ser expresada en función de las primarias. Sin embargo, no siempre es posible expresar una sentencia primaria en función de una secundaria. En este apartado a modo de resumen se muestra cómo cualquier estructura secundaria se puede realizar mediante las estructuras primarias.

Selección por casos:

La estructura switch también se puede realizar mediante selecciones en cascada.

bucle con contador

La estructura for se puede hacer mediante un control explícito del contador del bucle.

9 Estructuras de datos

Se generaliza la estructura vectorial mediante el uso de vectores abiertos y matrices o formaciones de varias dimensiones, así como los tipos unión, conjuntos, y estructuras de datos complejas que combinan varias estructuras más sencillas.

9.1 Argumentos de tipo vector abierto

Si un subprograma debe operar con un vector recibido como argumento, necesita toda la información del tipo de dicho vector, es decir, el tipo y número de sus elementos. Lo interesante está en escribir un procedimiento general de escritura de vectores de números enteros, y pasar como parámetro el tamaño del vector. Para eso hace falta un mecanismo para expresar que un argumento de tipo vector puede tener un tamaño cualquiera, es decir, indefinido. Los vectores con un tamaño indefinido se denominan vectores abiertos.

```
void EscribirVectorAbierto(const int v[ ], int numElementos) {  
    for(int i = 0; i < numElementos; i++) {  
        printf("%10d", v[i]);  
    }  
}
```

El precio que hay que pagar por disponer de esta facilidad es tener que pasar siempre la longitud concreta del vector como argumento, en cada llamada. Por supuesto, hay otras alternativas al paso explícito de la longitud del vector. Los subprogramas estándar para operar con cadenas de caracteres (strings) Usan la técnica de almacenar un carácter nulo al final del valor efectivo de cada cadena. Esto exige disponer siempre de espacio para un carácter más, al menos, pero evitan tener que pasar la longitud como argumento.

9.2 Formaciones anidadas. Matrices

En esta sección se generaliza el concepto de vector para disponer de formaciones de más de una dimensión. Las matrices son estructura de tipo formación (array) de dos o más dimensiones. Una forma sencilla de plantear la definición de estas estructuras es considerarlas como vectores cuyos elementos son a su vez vectores (o matrices).

Declaración de matrices:

```
const int NumFilas = 10;  
const int NumColumnas = 15;
```

```
typedef int TipoElemento;
typedef TipoElemento
TipoMatriz[NumFilas][NumColumnas];
TipoMatriz matriz;
```

Operaciones con matrices:

Las operaciones con elementos individuales de una matriz pueden hacerse directamente, de forma análoga a la operación con variables simples de ese tipo. En cambio las operaciones globales con matrices han de plantearse de manera similar a las operaciones globales con vectores. En general habrá que operar elemento a elemento, o a lo sumo por filas completas.

```
void EscribirMtriz(const TipoMatriz m){
for(int i = 0; i < NumFilas; i++){
for(int j = 0; j < NumColumnas; j++){ }
}
}
```

Los lenguajes C y C++ no permiten el uso de argumentos de tipo matriz abierta. Si realmente se necesita definir operaciones genéricas con formaciones de dimensiones indefinidas hay que recurrir al uso explícito de punteros.

9.3 El esquema de unión

Hay aplicaciones en las que resultaría deseable que el tipo de un dato variase según las circunstancias. Si las posibilidades de variación son un conjunto finito de tipos, entonces se puede decir que el tipo del dato corresponde a un esquema que es la unión de los tipos particulares posibles. Como situaciones típicas en las que se pueden aplicar los esquemas unión tenemos, entre otras, las siguientes:

- Datos que pueden representarse de diferentes maneras.
- Programas que operan indistintamente con varias clases de datos.
- Datos estructurados con elementos opcionales.

El tipo union:

Un tipo union se define como una colección de campos alternativos, de tal manera que cada dato particular sólo usará uno de esos campos en un momento dado, dependiendo de la alternativa aplicable. La definición es similar a la de un agregado o struct, usando ahora la palabra clave union. Ej: typedef struct TipoFraccion {

```
int numerador;
int denominador;
};
typedef union TipoNumero {
int valorEntero;
float valorReal;
TipoFraccion valorRacional;
};
```

La referencia a los elementos se hace también como en los tipos struct:

```
TipoNumero numero, otro, fraccion1, fraccion2;
numero.valorEntero = 33;
otro.valorReal = float(numero.valorEntero);
fraccion2.valorRacional = fraccion1.valorRacional;
fraccion1.valorRacional.numerador = 33;
fraccion1.valorRacional.denominador = 44;
```

Es fácil darse cuenta de que un dato de un tipo unión no contiene en sí mismo información de cuál es la variante activa en un momento dado. Dicha información debe estar disponible de forma clara por otros medios.

Registros con variantes:

El hecho de que un dato de tipo unión deba ir acompañado de información complementaria para saber cuál es la variante aplicable hace que los tipo unión aparezcan casi siempre formando parte de estructuras más complejas. Un ejemplo frecuente es lo que se denominan registros con variantes. Se trata de agregados o tuplas en los que hay una colección de campos fijos, aplicables en todos los casos, y campos variantes que se definen según el esquema unión. Además suele reservarse un campo fijo para indicar explícitamente cuál es variante aplicable en cada momento. A dicho campo se le llama discriminante. Ej:

```
typedef enum ClaseNumero {Entero, Real, Fraccion};
typedef struct TipoFraccion {
```

```

int valorEntero;
float valorReal;
TipoFracción valorRacional;
}
typedef struct TipoNumero {
ClaseNumero clase; /* discriminante*/
TipoValor valor;
}
void EscribirNumero( TipoNumero n) {
switch(n.case){
case Entero:
printf("%d", n.valor.valorEntero);
break;
case Real:
printf("%f", n.valor.valorReal);
break;
case Fraccion:
printf("%d%d", n.valor.valorRacional.numerador, n.valor.valorRacional.denominador);
break;
default:
printf("???"); }
}

```

Si el tratamiento de los registros con variantes se hace sólo mediante subprogramas que comprueban siempre el discriminante antes de operar, el código resulta mucho más seguro. Esta garantía de seguridad compensa perfectamente el esfuerzo adicional que representa definir y usar estructuras anidadas.

9.4 Esquemas de datos y esquemas de acciones

Una vez presentadas las estructuras de datos de tipos formación y registro, incluidos los registros con variantes, es interesante hacer notar la analogía que puede establecerse entre los esquemas de acciones y los esquemas de datos.

La programación estructurada recomienda usar los esquemas más sencillos posibles para organizar un elemento complejo de un programa a partir de elementos más simples. Recordaremos que estos esquemas básicos eran la secuencia, la selección, y la iteración, en el caso de acciones compuestas. Estos esquemas se corresponden, respectivamente, con los esquemas tupla, unión y formación, definidos para las estructuras de datos. La analogía se pone de manifiesto si describimos dichos esquemas de forma generalizada, común a los datos y acciones.

Tupla - Secuencia: Colección de elementos de tipos diferentes, combinados con un orden fijo.

Unión - Selección: Selección de un elemento entre varios posibles, de tipos diferentes.

Formación - Iteración: Colección de elementos del mismo tipo.

Todavía puede manifestarse la analogía con más intensidad si observamos que el tratamiento de las componentes de una estructura de datos se realiza fácilmente con una acción compuesta de tipo análogo. Ej:

```

/*ESQUEMAS DE DATOS*/
/*Esquema Tupla*/
typedef struct TipoTupla {
char uno, dos, tres;
};
TipoTupla agregado;
/*Esquema Unión*/
typedef union TipoUnion { int alfa;
float beta;
};
TipoUnion variante;
bool numeroEntero; /* Esquema Formación */
typedef int TipoFormacion[10];
tipoFormacion vector;
/*ESQUEMA DE ACCIONES*/
/* Secuencia para imprimir los campos una Tupla */ printf("%c", agregado.uno);
printf("%c", agregado.dos);

```



```

printf("%c", agregado.tres);
/*Selección para imprimir las variantes de una Unión*/
if(numeroEntero){
printf("%d", variante.alfa); } else { printf("%d", variante.beta);
}
/* Iteración para imprimir una Formación*/
for (int k=0; k <10; k++){
printf("%d", vector[k]);
}

```

9.5 Estructuras combinadas

En cualquier otro lenguaje de programación actual se combinar entre sí los esquemas tupla, unión y formación para definir estructuras de datos complejas, exactamente igual a como se combinan la secuencia, selección e iteración para construir el código de acciones complejas. Se pueden definir estructuras cuyos componentes son a su vez estructuras, sin límite de complejidad de los esquemas de datos resultantes.

Formas de combinación

Cuando se utilizan estructuras combinadas, para hacer referencia a una componente en particular hay que usar los selectores apropiados, encadenados uno tras otro. La combinación es admisible cuando un componente de una estructura es su vez una estructura de datos. A cada estructura se debe aplicar el selector que le corresponda.

Tablas

Aunque el estudio de las estructuras de datos excede del ámbito de este libro, resulta interesante mencionar algunos esquemas típicos que se obtienen combinando estructuras básicas. Este es el caso del esquema de tabla, que puede plantearse como una formación simple de registros. En otros contextos se le da también el nombre de diccionario o relación. Los esquemas de tabla son el fundamento de las bases de datos relacionales. Pueden construirse estructuras de datos bastante complejas combinando de manera que en algunas de ellas hagamos referencia a datos almacenados en otra. Una forma de hacer referencia a los datos de una tabla es usando el índice correspondiente a la posición de cada registro. Ej gestion tarjetas embarque 306.

10 Esquemas típicos de operación con formaciones

Cuando se trabaja con colecciones de datos es habitual que las operaciones globales sobre la colección se realicen a base de operar con los elementos uno a uno. Esto exige el empleo de esquemas de programas en los que se recorren los elementos de la colección en un orden adecuado. De momento se han introducido las estructuras de tipo formación (vectores y matrices) para almacenar colecciones de datos. En este tema se introducen los esquemas típicos de operación con formaciones, incluyendo recorrido, búsqueda, inserción y ordenación así como ciertas técnicas particulares que facilitan su programación tales como centinelas y matrices orladas, para los esquemas básicos se realizan previamente su especificación formal y se razona sobre la corrección de las implementaciones empleando los conceptos de precondition, postcondition, invariante y variante.

10.1 Esquemas de recorrido

El esquema de recorrido consiste en realizar cierta operación con todos y cada uno de los elementos de una formación (en algunos casos con parte de ellos). Evidentemente el esquema de recorrido se puede aplicar a formaciones de cualquier dimensión tales como matrices con dos o más índices. la forma mas general del recorrido sería:

```

iniciar operacion
while(quedan elementos sin tratar){ elegir uno de ellos y tratarlo
}
completar operación

```

Recorrido de matrices

Si la formación es de tipo matriz, para hacer el recorrido se necesitan tanto for anidados como dimensiones tenga la formación. Por ejemplo, el recorrido es la operación típica que se debe hacer con una formación cuando se quieren inicializar todos sus elementos.

Recorrido no lineal

Tratando el flujo con condicionales y jugando con el índice del bucle for.

10.2 Búsqueda secuencial

En las operaciones de búsqueda secuencial se examinan uno a uno los elementos de la colección para tratar de localizar los que cumplen una cierta condición. Si realmente queremos encontrar todos los que existan, entonces la búsqueda equivale a un recorrido como los mostrados en la sección anterior. Si no necesitamos localizar todos los elementos que cumplen la condición sino sólo uno de ellos, si lo hay, entonces no necesitamos recorrer la colección en su totalidad. El recorrido se debe detener en cuanto se encuentre el elemento buscado, y por tanto sólo será un recorrido completo cuando no se encuentre el elemento buscado dentro de la colección. Este planteamiento del problema indica que no se puede utilizar directamente una sentencia `for`.

10.3 Inserción

El problema que se plantea aquí es insertar un nuevo elemento en una colección de elementos ordenados, manteniendo el orden de la colección. Se supone que los elementos están almacenados en un vector, ocupando las posiciones desde el principio, y que queda algo de espacio libre al final del vector (si el vector estuviese lleno se podrían insertar nuevos elementos). La operación se puede realizar de forma iterativa, examinando los elementos empezando por el final hasta encontrar uno que sea inferior o igual al que se quiere insertar. Los elementos mayores que el que se quiere insertar se van moviendo una posición hacia adelante, con lo que va quedando un hueco en medio del vector. Al encontrar un elemento menor que el nuevo, se copia el nuevo elemento en el hueco que hay en ese momento. Un esquema general de código sería:

iniciar inserción

```
while(!Final && !Encontrar hueco){ Desplazar elemento
```

```
Pasar al siguiente elemento
```

```
}
```

```
Insertar vnuevo elemento
```

Ordenación por inserción directa Basado en el esquema de inserción mostrado anteriormente.

```
typedef...TElemento...;
```

```
void Ordenar(TElemento v[], int N){
```

```
<< PRE :>>
```

```
TElemento valor;
```

```
int j;
```

```
<< INVARIANTE :  $\forall k \in (1..i - 1) : v[k] \geq v[k - 1]$  y los anteriores están ordenados >>
```

```
for (int i = 1; i < N; i++) {
```

```
    v[j] = v[i-1];
```

```
    j--;
```

```
}
```

```
    v[j] = valor
```

```
}
```

```
<< POST :  $v[0..N - 1]$  está ordenado y contiene los valores dev >>
```

```
}
```

10.4 Búsqueda por dicotomía

Cuando los datos están ordenados la búsqueda resulta mucho más rápida. Si comparamos el elemento a buscar con el que está justo en la mitad de los datos ordenados, podemos decidir si es éste el elemento que buscamos, debemos continuar buscando, pero sólo en la mitad derecha o sólo en la mitad izquierda. El mismo proceso se puede repetir con la mitad elegida, comparando con el elemento que está en el centro de dicha mitad. En cada comparación, la búsqueda se reduce a comprobar si el dato buscado está entre la mitad de los anteriores. La búsqueda finaliza cuando el elemento se encuentra, o bien cuando la zona pendiente de examinar queda reducida a un sólo elemento (o ninguno) después de sucesivas divisiones en mitades. Esta búsqueda se denomina búsqueda por dicotomía.

iniciar operación

```
while (quedan elementos por examinar y no se ha encontrado ninguno aceptable){
```

```
    elegir el elemento central y ver si es aceptable
```

```
}
```

completar operación

La búsqueda por dicotomía se puede aplicar, por ejemplo, en el esquema de ordenación por inserción, para localizar el lugar en que hay que insertar nuevo elemento.

10.5 Simplificación de condiciones de contorno

La programación de operaciones con vectores, realizadas elemento a elemento exige con frecuencia realizar un tratamiento especial de los elementos extremos del vector o, en general, de los elementos del contorno de una formación. A continuación veremos algunas técnicas particulares para evitar la necesidad de detectar de manera explícita si se ha llegado a un elemento del contorno y/o realizar con él un tratamiento especial.

Técnica del centinela

En el procedimiento general de búsqueda es necesario comprobar en cada iteración una condición doble: si no se ha alcanzado todavía el final del vector, y si se encuentra el elemento buscado.

La doble condición del bucle de iteración complica el código y supone tiempo adicional en la ejecución de cada iteración. Si garantizamos que dato buscado está dentro de la zona de búsqueda, antes o después se terminara encontrándolo y ya no será necesario comprobar explícitamente Si se alcanza el final del vector. La manera de asegurar esto es incluir el dato buscado en el vector antes comenzar la búsqueda. el vector se amplía se copia el dato a buscar antes de iniciar la búsqueda para que actúe como centinela (C) y asegurarse de que la búsqueda nunca acaba infructuosa.

Matrices orladas

Cuando se trabaja con matrices, también se pueden simplificar las condiciones de contorno utilizando matrices orladas. Estas matrices se dimensionan con dos filas y dos columnas más de las necesarias. La matriz original la forman las filas 1 a M y las columnas 1 a N. Las filas y columnas extra garantizan que todos los elementos de la matriz original tienen elementos vecinos en todas las direcciones. Antes de operar con la matriz inicializan las filas y columnas extra con un valor de contorno (e) que permitan simplificar la operación de modo que el tratamiento de los elementos del borde de la matriz original sea idéntico al de los demás. Ejemplos de programas que se opera a fuerza bruta que son poco eficientes pero muy sencillos de programar.

11 Punteros y variables dinámicas

En este tema se introducen estructuras de datos potencialmente ilimitadas. Se justifica su interés y se describe en particular la estructura secuencia.

11.1 Estructuras de datos no acotadas

Tras el análisis anterior debe resultar evidente que sería útil disponer de estructuras de datos que no tuvieran un tamaño fijado de antemano, sino que pudieran ir creciendo o reduciendo su tamaño en función de los datos particulares que se estén manejando en cada ejecución del programa. Estas estructuras de datos se denominan, en general, estructuras dinámicas, y poseen la cualidad de que su tamaño es potencialmente ilimitado, aunque, naturalmente, no podrá exceder la capacidad física del computador que ejecute el programa.

La estructura secuencia

La estructura secuencia puede definirse como un esquema de datos del tipo iterativo, pero con un número variable de componentes. La estructura secuencia resulta parecida a una formación con número variable de elementos. En realidad existen diferentes esquemas secuenciales de datos. Aun teniendo en común que el número de elementos pueda variar, hay varias formas posibles de plantear las operaciones sobre secuencias. Para describir las distintas alternativas distinguiremos entre operaciones de construcción y de acceso. Con las primeras podremos añadir o eliminar componentes de la secuencia. Con las segundas podremos obtener o modificar el valor de las componentes que existen en un momento dado.

Las operaciones de construcción pueden incluir:

- Añadir o retirar componentes al principio de la secuencia.
- Añadir o retirar componentes al final de la secuencia.
- Añadir o retirar componentes en posiciones intermedias de la secuencia.

Las operaciones de acceso pueden ser:

Acceso secuencial: los componentes deben tratarse una por una, en el orden en que aparecen en la secuencia.

Acceso directo: Se puede acceder a cualquier componente directamente indicando su posición, como en una formación o vector.

En muchos casos, y en particular cuando el acceso es secuencial, el tratamiento de una secuencia se realiza empleando un cursor. El cursor es una variable que señala a un elemento de la secuencia. El acceso, inserción o eliminación de componentes de la secuencia se hace actuando sobre el elemento señalado por el cursor. Para actuar

sobre el cursor se suelen plantear las siguientes operaciones:

-Iniciar: Pone el cursor señalando al primer elemento

-Avanzar: El cursor pasa a señalar el siguiente elemento.

-Fin: Es una función que indica si el cursor ha llegado al final de la secuencia.

11.2 Variables dinámicas

Una manera de realizar estructuras de datos ilimitadas es mediante el empleo de variables dinámicas. Una variable dinámica no se declara como tal, sino que se crea en el momento necesario, y se destruye cuando ya no se necesita. Las variables dinámicas no tienen nombre, sino que se designan mediante otras variables llamadas punteros o referencias.

Punteros

Los punteros o referencias son variables simples cuyo contenido es precisamente una referencia a otra variable. El valor de un puntero no es representable como número o texto. En su lugar usaremos una representación gráfica en la que utilizaremos una flecha para enlazar una variable de tipo puntero con la variable a la que hace referencia. Los punteros de C están tipados, al igual que los demás valores manejados en el lenguaje. El tipo de puntero especifica en realidad el tipo de variable a la que puede apuntar. La declaración es:

```
typedef Tipo-de-variable* Tipo-puntero;
```

Una vez declarado el tipo, se pueden declarar variables puntero de dicho tipo. Una variable puntero se puede usar para designar la variable apuntada mediante la notación:

```
*puntero
```

por ejemplo:

```
typedef int* tp-Entero;
```

```
tp-Entero pe;
```

```
*pe = 33;
```

```
printf("%d", *pe);
```

Estas sentencias asignan el valor 33 a la variable dinámica, señalada por el puntero pe, y luego la imprimen. Para que estas sentencias funcionen correctamente es necesario que exista realmente la variable apuntada. Si el puntero no señala realmente a una variable dinámica, el resultado de usar *puntero será imprevisible.

Para poder detectar si un puntero señala realmente o no a otra variable, existe en C el valor especial NULL (que no es una palabra clave, sino que está definido en la librería estándar stdlib.h y también en otras librerías). Este valor es compatible con cualquier tipo de puntero, e indica que el puntero no señala a ninguna parte. Por lo tanto debería ser asignado a cualquier puntero que sepamos que no señala a ninguna variable. Normalmente se usará para inicializar las variables de tipo puntero al comienzo del programa. La inicialización no es automática, sino que debe ser realizada, expresamente por el programador.

```
if(pe != NULL){
```

```
  *pe = 33;
```

```
  printf("%d", *pe);
```

```
}
```

En principio esta sentencia garantizaría que sólo se usa la variable apuntada cuando realmente existe. En realidad eso no es del todo cierto, ya que sólo una correcta disciplina en el uso de punteros permite asumir que sólo tienen valor no nulo los punteros que realmente señalan a variables que existen. El lenguaje C en sí mismo no puede garantizarlo. Por ejemplo, se puede destruir una variable dinámica pero conservar punteros que la referenciaban, que ahora señalan a algo inexistente.

Uso de variables dinámicas:

Una variable de un programa se corresponde, en general, con una zona concreta de la memoria que el compilador reserva para almacenar en ella el valor de la variable. Las variables normales de un programa tienen esa zona de memoria reservada de antemano al empezar a ejecutarse el programa o subprograma en que se declaran, y por tanto pueden ser usadas en cualquier momento. Conviene recordar que las variables declaradas en un subprograma sólo existan mientras se ejecutan las sentencias de ese subprograma.

Las variables dinámicas no tienen ese espacio de memoria reservado de ante-mano, sino que se crean a partir de punteros en el momento en que se indique.. Además, una vez creadas siguen existiendo incluso después de que termine la ejecución del subprograma donde se crean. La forma más sencilla de crear una variable dinámica es mediante el operador new.

```
typedef Tipo-de-variable* Tipo-puntero;
```

```
Tipo-puntero puntero;
```

```
puntero = new Tipo-de-variable;
```

El operador new crea una variable dinámica del tipo indicado y devuelve una referencia que puede asignarse a

un puntero de tipo compatible. Como en cualquier otra asignación, el valor anterior del puntero se pierde. Tal como se ha dicho antes, la variable dinámica no tiene nombre, y sólo se puede hacer referencia a ella a través del puntero.

La variable dinámica se crea a base de reservar el espacio necesario en una zona general de memoria gestionada dinámicamente. En principio no se puede asumir que la variable recién creada tenga un valor concreto, igual que las variables normales que se declaran sin un valor inicial explícito.

Las variables dinámicas, una vez creadas, siguen existiendo hasta que se indique explícitamente que ya no son necesarias, en cuyo caso el espacio que se había reservado para ellas quedará otra vez disponible para crear nuevas variables dinámicas. Para ello existe la sentencia delete, que permite destruir variable dinámica a la que señala un puntero.

```
delete puntero;
```

Esta sentencia destruye la variable apuntada pero no garantiza que el puntero quede con un Valor determinado. En particular no garantiza que tome Valor NULL. Una variable dinámica puede estar referenciada por más de un puntero. Esto ocurre cuando se copia un puntero en otro.

```
typedef int* tp-entero;
```

```
tp-entero p1, p2;
```

```
p1 = new int;
```

```
p2 = p1;
```

Un problema delicado al manejar variables dinámicas es que pueden quedar perdidas, sin posibilidad de hacer referencias a ellas. En este caso la variable creada mediante `p2 = new int;` queda perdida, sin posibilidad de ser usada, además el espacio ocupado por la variable dinámica sigue reservado, lo cual es totalmente inútil y representa una pérdida de capacidad de memoria disponible (en inglés se denomina *memory leak*).

11.3 Realización de secuencias mediante punteros

Los punteros son un elemento de programación de muy bajo nivel. Los lenguajes de programación simbólicos deberían evitar su empleo, sustituyéndolo por mecanismos más potentes de declaración de estructuras de datos, que permitiesen definir directamente estructuras dinámicas ilimitadas. Desgraciadamente, muchos lenguajes están diseñados pensando en que su compilación no sea demasiado complicada. Las estructuras de datos con tamaño variable presentan algunas complicaciones para ser manejadas de manera eficiente, y es frecuente que los lenguajes de programación no incorporen directamente esquemas de datos de tamaño variable. Esta limitación facilita el trabajo de compilación, ya que todas las variables tendrán un tamaño fijo que puede ser calcularlo por el compilador, y determinar así el espacio de memoria que ha de reservarse para cada una.

C no dispone de esquemas de datos de tamaño variable. Dichos esquemas pueden ser realizados indirectamente por el programador mediante el uso de punteros. Al hacerlo convendrá tener cuidado, y emplearlos de una manera precisa, traduciendo a punteros los mecanismos de definición de alto nivel que deberían estar disponibles. La definición simbólica de una estructura ilimitada basándose en esquemas con número fijo de elementos será, normalmente, recursiva. Una definición recursiva es aquella en que se hace referencia a sí misma. Para definir una secuencia ilimitada tendremos que recurrir al empleo de variables dinámicas y punteros. Una manera de hacerlo es usar punteros para enlazar cada elemento de la secuencia con el siguiente. Cada elemento de la secuencia se materializa como un registro con dos campos: el primero contiene el valor de una componente, y el segundo es un puntero que señala al siguiente. El último elemento tendrá el puntero al siguiente con valor NULL. La secuencia completa es accesible a través de un puntero que señala al comienzo de la misma. Ej:

```
typedef struct Tipo-nodo {
```

```
Tipo-componente primero;
```

```
Tipo-nodo *resto;
```

```
};
```

```
typedef Tipo-nodo* Tipo-secuencia;
```

Esta pareja de definiciones es válida en C, aunque tiene una característica excepcional: se usa identificador `Tipo-nodo` antes de haber sido definido completamente. Esto solo es posible hacerlo en declaraciones de punteros como equivalente a una definición recursiva de un esquema de datos.

```
Tipo-componente valor;
```

```
Tipo-secuencia secuencia, siguiente;
```

```
if(secuencia != NULL){
```

```
secuencia -> primero = valor;
```

```
siguiente = secuencia -> resto;
```

```
}
```

11.3.1 Operaciones con secuencias enlazadas

Describiremos la manera de realizar algunas operaciones típicas sobre secuencias enlazadas con punteros. En ellas supondremos la existencia de un cursor que va señalando a las componentes una tras otra. El cursor será simplemente un puntero.

DEFINICIÓN: La definición de la secuencia será:

```
typedef struct TipoNodo {
    int valor;
    TipoNodo *siguiente;
}
typedef TipoNodo * TipoSecuencia;
```

RECORRIDO: El recorrido de toda la secuencia se consigue mediante un bucle de acceso a elementos y avance del cursor. Puesto que la secuencia tiene un número indefinido de elementos, no se usará un bucle con contador. Usaremos un esquema while. Como ejemplo describimos la escritura de los valores de la secuencia.

```
typedef TipoNodo * TipoPuntNodo;
TipoPuntNodo cursor;
cursor = secuencia;
while(cursor != NULL){
    printf("%5d", cursor->valor);
    cursor = cursor->siguiente;
}
```

BÚSQUEDA: La búsqueda en una secuencia enlazada ha de hacerse de forma secuencial. La búsqueda es parecida al recorrido, pero la condición de terminación cambiará. De hecho habrá una doble condición de terminación: que se localice el elemento buscado, y/o que se agote la secuencia. A continuación se presenta la búsqueda de la posición en que ha de insertarse un nuevo número en la secuencia ordenada. La posición será la que ocupe el primer valor igual o mayor que el que se quiere insertar.

```
int numero;
TipoPuntNodo cursor, anterior;
cursor = secuencia;
anterior = NULL;
while(cursor != NULL && cursor->valor < numero){
    anterior = cursor;
    cursor = cursor->siguiente;
}
```

Al salir del bucle cursor queda señalado al punto en que deberá insertarse el nuevo elemento, y anterior señala al elemento que lo precede. Esto resulta útil para realizar luego operaciones de inserción o borrado, como se verá a continuación.

INSERCIÓN: La inserción de un nuevo elemento se consigue creando una variable dinámica para contenerlo, y modificando los punteros para enlazar dicha variable dentro de la secuencia. El caso más sencillo es el de insertar un nuevo elemento dentro de uno dado. El orden de las operaciones a realizar resulta esencial para que no se produzca la pérdida de ninguna variable dinámica.

```
int numero; /*valor a insertar*/
TipoPuntNodo cursor, anterior, nuevo;
nuevo = new TipoNodo;
nuevo->valor = numero;
nuevo->siguiente = anterior->siguiente;
anterior->siguiente = nuevo;
```

BORRADO: Para borrar un elemento hay que quitar el nodo que lo contiene, enlazando directamente el anterior con el siguiente. Es la operación inversa a la inserción. Si el nodo que contenía el elemento ya no es necesario hay que destruirlo explícitamente. Para hacer el código más robusto se ha forzado el cursor a valor nulo, ya que de no haberlo así quedaría apuntando a un lugar que ya no existe.

```
TipoPuntNodo cursor, anterior;
anterior->siguiente = cursor->siguiente;
delete cursor;
cursor = NULL;
```

11.4 Punteros y paso de argumentos

El manejo de punteros cuando se utilizan como argumentos de un subprograma tiene ciertas peculiaridades que requieren un estudio más detallado.

Paso de punteros como argumentos

Como cualquier otro dato, un puntero puede pasarse como argumento a un subprograma. Así, la operación de imprimir la lista de números enteros del ejemplo anterior podría redactarse como procedimiento que reciba como argumento la secuencia enlazada. Tal como se veía, una secuencia enlazada se maneja a partir del puntero al primer elemento.

```
void ImprimirLista(TipoSecuencia lista){
TipoPuntNodo curspr = lista;
while(cursor != NULL){ printf("%5d", cursor->valor);
cursor = cursor->siguiente;
}
printf("barra n");
}
```

```
TipoSecuencia secuencia;
ImprimirLista(secuencia);
```

Por defecto, los datos de tipo puntero se pasan como argumentos por valor. Es lo que ocurre en el ejemplo anterior. Si se desea usar un subprograma para modificar datos de tipo puntero, entonces habrá que pasar el puntero por referencia. Por ejemplo, si planteamos como subprograma la operación de búsqueda en una secuencia enlazada podríamos escribir:

```
void Buscar(TipoSecuencia lista, int numero, TipoPuntNodo & cursor, TipoPuntNodo & anterior) {
cursor = lista ;
anterior = NULL;
while(cursor != NULL && cursor->valor != numero) {
anterior = cursor;
cursor = cursor->siguiente;
}
}
TipoSecuencia secuencia;
TipoPuntNodo encontrado, previo;
int dato;
Buscar(secuencia, dato, encontrado, previo);
```

Paso de argumentos mediante punteros

En general el valor de un puntero en sí mismo no es significativo, sino que el puntero es sólo un medio para designar la variable apuntada. Desde un punto de vista conceptual el paso de un puntero como argumento puede ser considerado equivalente a pasar como argumento la variable apuntada. Si queremos pasar como argumento una variable dinámica podemos recurrir a un puntero como elemento intermedio para designarla. Esto representa una dificultad añadida para entender cómo funciona determinado fragmento de código, y de hecho representa un peligro potencial de cometer determinados errores de codificación. La dificultad reside en que pasar un puntero por valor no evita que el subprograma pueda modificar la variable apuntada. Al establecer la analogía entre el paso como argumento del puntero y el paso como argumento de la variable apuntada, la distinción entre paso por valor y por referencia se pierde. El paso por valor de un puntero equivale al paso por referencia de la variable apuntada. En realidad los punteros se usan implícitamente para pasar argumentos por referencia. Cuando se declara un argumento pasado por referencia, lo que hace realmente el compilador es pasar un puntero a la variable externa usada como argumento actual en la llamada.

De hecho la notación & para indicar el paso de argumento por referencia es una mejora de C++ respecto a C. En lenguaje C hay que usar siempre un puntero explícito para pasar argumentos por referencia. Por supuesto, hace falta entonces un operador especial para obtener el valor de un puntero a una variable (estática) y usarlo en la llamada al subprograma. En lenguaje C ese operador corresponde también al símbolo &.

Ahora debe quedar clara la manera de invocar ciertas funciones estándar de C tal como scanf().

Lo que se está haciendo en esta llamada es pasar como argumento un puntero que señala a la variable numero, a través del cual se puede modificar su valor. Hay que hacer notar que el puntero en sí se está pasando por valor, y eso es equivalente a pasar el dato apuntado por referencia. En bastantes casos cuando se trabaja con variables dinámicas, que sólo son accesibles a través de punteros, resulta natural usar un puntero explícito para pasar como

argumento la variable dinámica apuntada. Insistiremos en que el paso del puntero equivale a pasar la variable apuntada siempre por referencia.

11.5 Punteros y valores en C y C++

En C/C++ existe una estrecha relación entre las formaciones y los punteros. Sin embargo desde un punto de vista metodológico esta relación resulta bastante confusa, disminuye la claridad y aumenta la ambigüedad de los programas. Lamentablemente, en el lenguaje C± no ha sido posible incorporar restricciones sintácticas capaces de impedir el manejo de punteros como formaciones debido al carácter semántico de esta relación. Por ello, es en el manual de estilo donde se incorpora una regla de obligado cumplimiento que prohíbe el uso de punteros como formaciones.

12 Tipos abstractos de datos (TADs)