

Lenguaje C: características generales.
Elementos del lenguaje. Estructura de un
programa. Funciones de librería y
usuario. Entorno de compilación.
Herramientas para la elaboración y
depuración de programas en lenguaje C.

TEMA 31 (34 SAI)

ABACUS NT

Índice

- 1. Introducción**
- 2. Características generales**
- 3. Elementos del lenguaje C**
 - 3.1. Palabras reservadas**
 - 3.2. Identificadores**
 - 3.2.1. Constantes
 - 3.2.2. Variables
 - 3.2.3. Tipos de datos
 - 3.3. Operadores**
 - 3.3.1. Operadores aritméticos
 - 3.3.2. Operadores lógicos
 - 3.3.3. Operadores de relación
 - 3.3.4. Operadores unitarios
 - 3.3.5. Operadores lógicos para manejo de bits
 - 3.3.6. Operadores de asignación
 - 3.3.7. Otros operadores
 - 3.4. Conversión de tipos**
- 4. Estructura de un programa**
- 5. Funciones de librería y de usuario**
 - 5.1.1. Funciones de usuario.
 - 5.2. Sentencias de preprocesador**
 - 5.3. Librerías estándar**
 - 5.4. Librería de funciones de usuario**
- 6. Herramientas para la elaboración y depuración de programas.**
 - 6.1. Compilador gcc (GNU compiler).
 - 6.2. Depurador gdb (GNU debugger).
 - 6.3. Herramienta make.
 - 6.4. Sistema de control de versiones CVS.
- 7. Conclusión**
 - 7.1. Relación del tema con el sistema educativo actual
- 8. Bibliografía**

1. Introducción

C es un lenguaje de programación de propósito general desarrollado por **Dennis Ritchie** entre 1969 y 1972 en los Laboratorios Bell, como evolución del anterior lenguaje B, a su vez basado en BCPL.

C es un lenguaje orientado a la implementación de sistemas operativos. C es el lenguaje de programación más popular para crear software de sistema, aunque también se utiliza para crear aplicaciones.

Se trata de un lenguaje de tipos de datos estáticos, débilmente tipificado, de medio nivel, ya que dispone de las estructuras típicas de los lenguajes de alto nivel, pero, a su vez, dispone de construcciones del lenguaje que permiten un control a muy bajo nivel. Los compiladores suelen ofrecer extensiones al lenguaje que posibilitan mezclar código en ensamblador con código C o acceder directamente a memoria o dispositivos periféricos.

La primera estandarización del lenguaje C fue en ANSI, en 1989. El lenguaje que define este estándar fue conocido vulgarmente como ANSI C. Posteriormente, en 1990, fue ratificado como estándar ISO. La adopción de este estándar es muy amplia por lo que, si los programas creados lo siguen, el código es portable entre plataformas y/o arquitecturas.

2. Características generales

C se considera como un lenguaje de **nivel intermedio**, ya que, aunque es un lenguaje de alto nivel, también incorpora características de bajo nivel como:

- **Manipulación de variables a nivel de bit**, byte y direcciones de memoria.
- Gestión de rutinas codificadas en **ensamblador**.
- Usa un lenguaje de **preprocesado**, el preprocesador de C, para tareas como definir macros e incluir múltiples archivos de código fuente.
- Acceso a memoria de bajo nivel mediante una potente **aritmética de punteros**.
- Manejo de Interrupciones mediante la biblioteca **signal**.

Las principales características del lenguaje C, son:

- Código muy portable
- Lenguaje cercano al pseudocódigo, permite generar programas de fácil modificación.
- Es débilmente tipado: no existen comprobaciones de tipos en tiempo de ejecución y permite hacer conversiones mediante casting o directas.
- Tiene solamente 32 palabras reservadas.
- Es un lenguaje estructurado, que incluye instrucciones de control, secuenciales, condicionales e iterativas.
- Soporta programación modular: mediante el uso de funciones y librerías.
- Es muy veloz y potente, lo que permite la creación de un software efectivo.
- Produce programas de código compacto y eficiente.

- Es compilado, y posee compiladores para la mayoría de plataformas hardware y sistemas operativos.
- Soporta recursividad, la cual se puede implementar fácilmente.
- Existen una gran cantidad de librerías definidas para este lenguaje, así como documentación y manuales de las mismas.
- El paso de variables es siempre por valor (copia el valor de la variable) Las llamadas con variables por referencia se implementan mediante punteros.
- El lenguaje C es sensible a mayúsculas y minúsculas (case sensitive), de modo que no es lo mismo para el compilador el identificador “*contador*” que otro denominado “*Contador*”.

3. Elementos del lenguaje C

3.1. Palabras reservadas

El estándar ANSI C utiliza las siguientes **palabras clave** (siempre en minúscula)

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

3.2. Identificadores

Los identificadores en C son constantes y variables. Están formados por secuencias de una o más caracteres alfanuméricos y el símbolo de subrayado. El estándar ANSI indica lo siguiente:

- Un identificador debe comenzar con un carácter alfabético o un símbolo de subrayado.
- Sólo se reconoce los 32 primeros caracteres, pero puede ser de cualquier tamaño.
- El compilador de C distingue entre mayúsculas y minúsculas.
- Los identificadores que empiecen con el símbolo de subrayado deben evitarse, porque pueden producir conflictos con variables o rutinas del sistema.
- Los identificadores no pueden tener la misma secuencia de caracteres que una palabra clave o reservada

3.2.1. Constantes

Las constantes se refieren a valores fijos que no pueden alterarse en el programa.

Se clasifican en constantes simbólicas (con identificador) o literales.

Constantes simbólicas

Pueden definirse constantes simbólicas de cualquiera de los tipos de datos simples. Por omisión del tipo, C toma como representación de la constante el tipo más pequeño de datos que la pueda soportar.

Se declaran colocando el modificador `const` delante del tipo de dato:

```
const tipo nombre_constante[=valor1];
```

Constantes literales

Un literal es una expresión de un valor en un tipo primitivo, esto es: numérico, carácter o bien una cadena de caracteres. En C, los caracteres literales se representan entre comillas simples y las cadenas literales entre comillas dobles; los números se pueden especificar en decimal (enteros y reales), octal o hexadecimal.

Secuencias de escape.

Los caracteres también pueden ser representados por secuencias de escape. Una secuencia de escape está formada por el carácter \ seguido de una "letra" o de una "combinación de dígitos". Son utilizadas para acciones como nueva línea, tabular y para representar caracteres no imprimibles.

3.2.2. Variables

Una variable representa un espacio de memoria para almacenar un valor de un tipo determinado. Las variables deben estar previamente declaradas antes de poder ser referenciadas:

```
tipo nombre_variable[=valor];
```

donde "tipo" es un tipo de datos válido en C con los modificadores necesarios, "nombre_variable" es el identificador de la variable, y "valor" el valor inicial.

Las variables pueden tomar un valor inicial mediante una sentencia de asignación. También se pueden declarar y asignar varias variables del mismo tipo en una sentencia simple.

Las variables, en función del lugar donde se declaren, pueden ser:

Variables locales. Se declaran dentro de un bloque delimitado por llaves {}).

Sólo pueden ser referenciadas por sentencias de su mismo bloque. Existen durante la ejecución del bloque de código en el que son declaradas, después se destruyen. Se pueden definir en cualquier bloque de código del programa y sólo pueden declararse al principio del bloque.

Variables globales. Se declaran fuera de todas las funciones. Se puede acceder a ellas desde cualquier bloque del programa, y están ocupando memoria durante toda la ejecución del mismo. Si en un bloque se declaran una variable local con el mismo nombre que alguna variable global, prevalece dentro del bloque la referencia a la variable local.

Variables static: Si una variable local se declara *static*, se crea un almacenamiento permanente para ella como si fuera global, pero sólo es conocida en el bloque en el que se declaró. Cuando se inicializa una variable *static* en la declaración dentro de una función, sólo toma ese valor en la primera ejecución y después guarda el último valor asignado.

Todas las variables deben tener asociado un tipo de dato, aunque pueden estar inicializadas o no.

3.2.3. Tipos de datos

Hay dos clases de tipos: tipos fundamentales y tipos derivados.

Tipos fundamentales.

Los podemos clasificar en:

Tipos enteros: char, short, int, long y enum.

Tipos reales: float, double y long double.

Otros: void.

Cada tipo entero puede ser clasificado por las palabras clave "signed" o "unsigned", es decir, con signo (rango desde -(máx-1) hasta +(máx)) o positivos sin signo.

Tipos derivados.

Los tipos derivados se construyen a partir de los tipos fundamentales. Algunos de ellos son los siguientes: punteros, estructuras, uniones, arrays y listas.

Punteros

Un puntero es una dirección de memoria que indica dónde se localiza un objeto de un tipo especificado. Para definir una variable de tipo puntero se utiliza el operador de *indirección* `*`.

Ejemplo: `int * p = NULL;`

Este ejemplo declara un puntero `p`. No es necesario indicar el valor nulo, pero si no se hace, puede estar apuntando a cualquier espacio de memoria, ya que no se modifica el contenido de la dirección asignada, y esta puede contener cualquier valor anterior.

Tras reservar memoria (por ejemplo, con `malloc`) podemos asignar al puntero cualquier valor entero, por ejemplo “almacena 10 en la dirección de memoria `p`”:

```
p=malloc(sizeof(int));
*p=10;
```

Si queremos asignar a p la dirección de una variable que no sea de tipo puntero, se utiliza el carácter **dirección**: &

```
p = &var;
```

Ejemplo: la dirección de p es ahora la dirección de var.

3.3. Operadores

Los operadores son símbolos que indican como son manipulados los datos. Se pueden clasificar en los siguientes grupos: aritméticos, lógicos, relacionales, unitarios, lógicos para manejo de bits, de asignación, operador ternario para expresiones condicionales y otros.

3.3.1. Operadores aritméticos

Operador	Operación
+	Suma. Los operandos pueden ser enteros o reales
-	Resta. Los operandos pueden ser enteros o reales
*	Multiplicación. Los operandos pueden ser enteros o reales
/	División. Los operandos pueden ser enteros o reales.
%	Módulo o resto de una división entera. Los operandos tienen que ser enteros

3.3.2. Operadores lógicos

Operador	Operación
&&	Operador AND
	Operador OR
!	Operador NOT

3.3.3. Operadores de relación

Operador	Operación
<	Primer operando "menor que" el segundo
>	Primer operando "mayor que" el segundo
<=	Primer operando "menor o igual que" el segundo

>=	Primer operando "mayor o igual que" el segundo
==	Primer operando "igual que" el segundo
!=	Primer operando "distinto que" el segundo

3.3.4. Operadores unitarios

Operador	Operación
-	Cambia de signo al operando (complemento a dos).
~ 126)	Complemento a 1. El operando tiene que ser entero (carácter ASCII 126)

3.3.5. Operadores lógicos para manejo de bits

Operador	Operación
&	Operación AND al nivel de bits
	Operación OR al nivel de bits (ASCII 124)
^	Operación XOR al nivel de bits
<<	Desplazamiento a la izquierda
>>	Desplazamiento a la derecha

3.3.6. Operadores de asignación

Operador	Operación
++	Incremento
--	Decremento
=	Asignación simple
*=	Multiplicación más asignación
/=	División más asignación
%=	Módulo más asignación
+=	Suma más asignación
-=	Resta más asignación
<<=	Desplazamiento a izquierda más asignación

>>=	Desplazamiento a derecha más asignación
&=	Operación AND sobre bits más asignación
=	Operación OR sobre bits más asignación
^=	Operación XOR sobre bits más asignación

3.3.7. Otros operadores

Operador coma.

Un par de expresiones separadas por una coma se evalúa de izquierda a derecha. Todos los efectos de la expresión de la izquierda son ejecutados antes de evaluar la expresión de la derecha, a continuación, el valor de la expresión de la izquierda es descartado. El tipo y el valor del resultado son el tipo y el valor del operando de la derecha.

Ejemplo:

```
aux = v1, v1 = v2, v2 = aux;
for (a = 256, b = 1; b < 512; a/=2, b*=2)
```

Ejemplo:

```
fx(a, (b = 4, b+3), c, d)
```

Hay cuatro argumentos, de los cuales el segundo tiene un valor 7.

Operador de indirección (*) .

Este operador accede a un valor indirectamente a través de un puntero. El resultado es el valor direccionado por el operando.

Operador de dirección-de (&).

Este operador da la dirección de su operando. Este operador no se puede aplicar a un campo de bits perteneciente a una estructura o a un identificador declarado con el calificador “register”.

3.4. Conversión de tipos

Cuando los operandos dentro de una expresión son de tipos diferentes, se convierten a un tipo común, generalmente al de mayor tamaño y dando preferencia a int sobre char.

4. Estructura de un programa

Un programa fuente C es una colección de cualquier número de directrices para el compilador, declaraciones, definiciones, expresiones, sentencias y funciones.

main()

Todo programa C debe contener una función denominada main (), donde el programa comienza a ejecutarse. Las llaves ({}) que incluyen el cuerpo de esta función principal, definen el principio y el final del programa.

Un programa C, además de la función principal main (), consta generalmente de otras funciones que definen rutinas con una función específica en el programa. Esto quiere decir que la solución de cualquier problema, no debe considerarse inmediatamente en términos de sentencias correspondientes a un lenguaje, sino de elementos naturales del problema mismo, abstraídos de alguna manera, que darán lugar al desarrollo de las funciones mencionadas.

Directriz # include.

La directriz # include "fichero" le dice al compilador que incluya el fichero especificado, en el programa fuente. Esto es necesario porque estos ficheros aportan, entre otras declaraciones, las funciones prototipo de las funciones de la librería estándar que utilizamos en nuestros programas.

Directriz # define.

Mediante la directriz # define "identificador valor" se le indica al compilador, que toda aparición en el programa de "identificador", debe ser sustituida por "valor".

Declaraciones y definiciones.

Toda variable debe ser declarada antes de ser utilizada. En general, las variables no son inicializadas por C, pero si se desea, pueden ser inicializadas en la propia declaración.

La definición de una variable, declara la variable y además le asigna memoria; la definición de una función, declara la función y además incluye el cuerpo de la misma.

Sentencias.

Una sentencia es la unidad ejecutable más pequeña de un programa C. Las sentencias controlan el flujo u orden de ejecución. Una sentencia C consta de una palabra clave (for, while, if ... else, etc.), expresiones, declaraciones, o llamadas a funciones.

Toda *sentencia simple* termina con un punto y coma (;).

Dos o más sentencias pueden aparecer sobre una misma línea, separadas por punto y coma.

Una *sentencia nula* consta solamente de un "punto y coma".

Sentencia compuesta o bloque.

Una sentencia compuesta o bloque, es una colección de sentencias incluidas entre ({}). Un bloque puede contener otros bloques.

5. Funciones de librería y de usuario

5.1.1. Funciones de usuario.

Una función es una colección independiente de declaraciones y sentencias, generalmente enfocadas a realizar una tarea específica. Todo programa C consta al menos de una función, la función "main()". Además de ésta, puede haber otras funciones cuya finalidad es, fundamentalmente, descomponer el problema general en subproblemas más fáciles de resolver y de mantener. La ejecución de un programa comienza con la función "main()".

Las funciones de usuario son las declaradas y definidas por el programador en el programa, aunque puede convertirlas en funciones de librería.

Cuando se llama a una función, el control se pasa a la misma para su ejecución; y cuando finaliza, el control es devuelto de nuevo al módulo que llamó, para continuar con la ejecución del mismo a partir de la sentencia que efectuó la llamada.

Definición de una función.

La definición de una función consta de la *cabecera* de la función y del *cuerpo* de la función. La sintaxis correspondiente es:

```
[clase] [tipo] nombre ([parámetros-formales])
```

```
{  
[declaraciones]  
sentencias;  
}
```

clase Define el ámbito de la función, es decir, desde donde puede ser llamada. La clase puede ser: "extern" o "static".

Una función "static" es visible solamente en el fichero fuente en el cual está definida; y una función "extern" es visible para todos los ficheros fuente que componen el programa. Por defecto, la clase de una función es "extern".

tipo Indica el tipo del valor devuelto por la función. Puede ser cualquier tipo fundamental o tipo definido por el usuario. Por defecto, el tipo es "int". Una función no puede retornar un array o función, pero si puede retornar un puntero a un array o a una función.

nombre Es un identificador que indica el nombre de la función. Si el nombre va precedido por el operador asterisco (*), el valor devuelto por la función es un puntero.

parámetros formales Componen la lista de argumentos de la función. Esta lista consta de un conjunto de variables con sus tipos, separadas por comas y encerradas entre paréntesis. Los parámetros formales son variables que reciben los valores pasados en la llamada a la función.

Si no se pasan argumentos a la función, la lista de parámetros formales puede ser sustituida por la palabra clave "void".

Cuerpo de la función.

El cuerpo de una función está formado por una sentencia compuesta que contiene sentencias que definen lo que hace la función. También puede contener declaraciones de variables utilizadas en dichas sentencias. Estas variables, por defecto, son locales a la función.

Valor returned por una función.

Sentencia return. Cada función puede devolver un valor cuyo tipo se indica en la cabecera de función. Este valor es devuelto a la sentencia de llamada a la función, por medio de la sentencia "return", cuya sintaxis es la siguiente:

```
return [expresión];
```

Si la sentencia "return" no se especifica o se especifica sin contener una expresión, la función no devuelve un valor.

Llamada a una función.

La llamada a una función tiene la forma:

```
[variable =] expresión([parámetros-actuales]);
```

- variable: especifica la variable donde va a ser almacenado el valor devuelto por la función. Notar que la llamada puede prescindir del valor devuelto por la función.

- expresión: especifica una dirección que referencia a una función. Puede ser, una expresión que es evaluada a una dirección de una función, o simplemente un identificador que corresponde con el nombre de la función llamada. Esto significa que una función puede ser llamada a través de un puntero a una función.

- parámetros-actuales: son una lista de expresiones separadas por comas. Las expresiones son evaluadas y convertidas utilizando las conversiones aritméticas usuales. Los valores resultantes son pasados a la función y asignados a sus correspondientes "parámetros formales". El número de expresiones en la lista, debe ser igual al número de parámetros formales, a no ser que se especifique un número variable de argumentos.

Declaración de una función.

La declaración de una función, denominada también "función prototipo", permite conocer el nombre, el tipo del resultado, los tipos de los parámetros formales y opcionalmente sus nombres. No define el cuerpo de la función. Esta información permite al compilador chequear los tipos de los parámetros actuales por cada llamada a la función. Una función no puede ser llamada si previamente no está definida o declarada. A esta regla hay una excepción: que la definición de la función se haga con anterioridad a la llamada.

Paso de parámetros.

En el lenguaje C, los parámetros **siempre se pasan por valor**, esto es por **copia** de la variable. De esta forma se protege el contenido de la variable original y se impide que sea modificado.

Sin embargo, también es posible pasar un puntero como variable (o incluso una dirección de variable usando el operador &). Este método de paso de variable por dirección permite realizar en realidad lo que se conoce como **paso de variable por referencia**, ya que en este caso el valor original en memoria si se verá modificado en caso de ser alterado dentro de la función a la que se le pasa.

5.2. Sentencias de preprocesador

Las directrices para el preprocesador son utilizadas para hacer programas fuente fáciles de cambiar y de compilar en diferentes situaciones. Una directriz comienza con el símbolo # como primer carácter, distinto de blanco, en una línea, e indica al preprocesador una acción específica a ejecutar. Pueden aparecer en cualquier parte del fichero fuente, pero solamente se aplican desde su punto de definición hasta el final del programa fuente.

Las instrucciones para el preprocesador son:

```
#define  #error      #import   #undef
#define  #if         #include  #using
#define  #ifdef     #line     #endif
#define  #ifndef    #pragma
```

5.3. Librerías estándar

Junto con los compiladores de C y C++, se incluyen ciertos archivos llamados bibliotecas más comúnmente librerías o bibliotecas. Las bibliotecas contienen el código objeto de muchos programas que permiten hacer cosas comunes, como leer el teclado, escribir en la pantalla, manejar números, realizar funciones matemáticas, etc.

Hay un conjunto de bibliotecas (o librerías) que se incluyen con todos los compiladores de C. Son las librerías ANSI o estándar. Las bibliotecas están clasificadas por el tipo de trabajos que hacen, hay bibliotecas de entrada y salida, matemáticas, de manejo de memoria, de manejo de textos, etc.

La declaración de librerías se debe hacer al principio de todo nuestro código, antes de la declaración de cualquier función.

La sintaxis es la siguiente:

```
#include <nombre de la librería>
```

5.4. Librería de funciones de usuario

Un usuario puede crear sus propias librerías o utilizar librerías que no sean estándar del ANSI C. Para crear una librería de usuario basta con seguir estos pasos:

1. Crear el archivo de cabeceras

Creamos un archivo con extensión “.h” en el mismo directorio del código principal, éste archivo debe tener todos los prototipos de funciones y definiciones de tipos de datos de la librería.

2. Crear el archivo del código de la librería

El archivo del código de la librería, contiene incluye el archivo de cabecera que creamos anteriormente y además contiene el código de todas las funciones que fueron escritas en el archivo de cabecera. Lo almacenamos con extensión “.c”.

Se puede guardar una copia de la librería (fichero .h y .c) directamente en la carpeta “include” del compilador. Esto permitirá llamar a la librería desde cualquier código como si fuese una librería estándar.

3. Llamar a la librería

Después, se puede utilizar la librería como si fuese una librería estándar. Para llamar a la librería usamos la instrucción #include <libreria.h>

6. Herramientas para la elaboración y depuración de programas.

6.1. Compilador gcc (GNU compiler).

El compilador gcc es rápido, flexible y riguroso con el estándar de ANSI C. Puede funcionar como compilador cruzado para un gran número de arquitecturas distintas. En realidad, gcc no genera código máquina, sino código ensamblado. La fase de ensamblado a código máquina la realiza el ensamblador de GNU (gas) y el enlazador de GNU (ld) se encarga de los objetos resultantes. Este proceso es transparente para el usuario, ya que a no ser que se especifique lo contrario, gcc realiza automáticamente el paso desde código en C a un fichero ejecutable.

Su sintaxis es gcc [opciones] fichero(s) donde fichero(s) puede ser uno o varios ficheros fuente o ficheros objeto, y opciones puede ser:

-o <ejecutable> . El fichero ejecutable generado por gcc es por defecto a.out. Mediante este modificador, se especifica el nombre del ejecutable.

- Wall. No omite la detección de ningún aviso de compilación (warning).
- g. Incluye en el ejecutable información necesaria para utilizar un depurador.
- c. Preprocesa, compila y ensambla, pero no enlaza. El resultado es un fichero objeto con extensión .o y el mismo nombre que el fichero fuente.

6.2. Depurador gdb (GNU debugger).

Se trata de un depurador asociado a gcc, que necesita que el programa sea compilado previamente con la opción -g. El depurador permite establecer puntos de ruptura condicionales y detener la ejecución del programa cuando el valor de una expresión cambie.

Su sintaxis es gdb fichero donde fichero es el nombre del ejecutable creado con gcc.

Con esta instrucción se entra dentro del depurador, que se maneja mediante línea de comandos, aunque existe una interfaz gráfica de usuario para este depurador denominado ddd (GNU data display debugger) más intuitiva de manejar.

6.3. Herramienta make.

Un programa en C normalmente está formado por varios ficheros fuente y ficheros cabecera.

Cada vez que se modifica algún fichero fuente o cabecera, el programa debe recompilarse para crear un ejecutable actualizado. Sin embargo, no es necesario recompilar todos los ficheros, sino sólo los afectados por la modificación.

La utilidad make determina automáticamente qué ficheros del programa deben ser recompilados y las órdenes que se deben utilizar para ello. Para utilizar make es necesario escribir un fichero denominado Makefile, que describe las dependencias entre ficheros y las órdenes que se deben ejecutar con cada fichero. Si en un directorio existe un fichero Makefile, la orden make se encargará de ejecutar las órdenes que contiene.

Un fichero Makefile simple está compuesto por reglas que tienen la siguiente forma:

<etiqueta>:<lista de dependencias> <orden> <orden>

Una etiqueta es un rótulo que generalmente se refiere al nombre del fichero objeto que se quiere actualizar con la regla. También hay etiquetas que indican la acción que realiza la regla.

Una lista de dependencias es el conjunto de ficheros fuente u objeto que intervienen en la regla. Si cualquiera de estos ficheros es modificado, la regla se activa. Una orden es la acción que make realiza si la regla se activa. Las líneas de órdenes empiezan siempre con un tabulador, para distinguirlas de las líneas de etiquetas.

La orden make sin etiquetas activa la primera regla del fichero Makefile, la orden make <etiqueta> activa la regla correspondiente a la etiqueta especificada. Las etiquetas que indican el nombre de una acción deben declararse explícitamente como falsas con .PHONY para que make no compruebe la existencia de un fichero con ese nombre.

Es posible definir variables para simplificar el fichero Makefile. La definición se realiza antes de la primera etiqueta de la siguiente manera: nombre_variable=texto_al_que_sustituye, y la referencia a una variable se realiza mediante \$(nombre_variable). También se puede emplear caracteres comodín para hacer referencias a nombres de ficheros: '*' equivale a cualquier cadena de caracteres y '?' equivale a un carácter. El carácter comodín se puede escapar con '\'.

6.4. Sistema de control de versiones CVS.

El sistema de control de versiones CVS (Concurrent Versión System) es un sistema descentralizado en el cual cada programador utiliza su propio directorio de trabajo. Permite la edición concurrente de ficheros e integra todos los cambios realizados en un fichero mezcla.

Todas las versiones de un fichero se guardan de manera conjunta en un único fichero incrementalmente. Se basa en dos programas: **diff** (detecta diferencias entre dos ficheros y las vuelca a otro) y **patch** (reconstruye un fichero a partir del original y el fichero de diferencias).

El repositorio es el directorio donde se almacenan todos los ficheros con las diferentes versiones de uno o varios proyectos. Es el depósito común de información del proyecto, y se recurre a él para recuperar ficheros y almacenar los cambios. El repositorio almacena información de control para CVS y las diferencias entre las versiones de cada uno de los ficheros del proyecto.

El modelo de trabajo con CVS es el siguiente:

- Un programador descarga una copia desde el repositorio con checkout. El programador edita libremente su copia. Al mismo tiempo, cualquier otro miembro del equipo puede trabajar sobre otra copia de trabajo.
- El programador finaliza sus cambios y los entrega al repositorio de CVS con commit, junto con un texto explicativo de las modificaciones realizadas. CVS integra los cambios en la copia maestra del proyecto. Otros programadores pueden solicitar a CVS comprobar si la copia maestra ha sufrido cambios recientes con update.

7. Conclusión

7.1. Relación del tema con el sistema educativo actual

Este tema puede ser desarrollado en los siguientes módulos formativos (para atribución docente de PES)

- Bachillerato – Tecnologías de la Información y la Comunicación II (PES)
- GS- GS – DAW – Desarrollo Web en Entorno Servidor DAW/DAM – Programación (PES)

Aunque los profesores de SAI no tienen una atribución docente en ningún módulo con aplicación directa del tema, siempre es probable que se diseñe algún pequeño programa en clase, para lo que recurrir a la notación de pseudocódigo o de ordinograma sería un recurso esclarecedor.

8. Bibliografía

- Introduction to Java Programming and Data Structures, Comprehensive Version. Y. Daniel Liang. Pearson, 11^a edición. 2017.
- Java 9. Francisco Javier Moldes Teo. Anaya. 2017.
- Introducción a la computación. J, Glenn Brookshear, Pearson, 11^º edición. 2012
- Fundamentos de programación. Algoritmos, estructuras de datos y objetos. Luis Joyanes Aguilar, McGraw-Hill, 4^º edición. 2008.
- Langsam, Augenstein y Tanembaum: “Estructuras de Datos con C y C++”, Prentice-Hall 1997
- Prieto A., Lloris A. y Torres J.C.: Introducción a la Informática, 4^a ed (2006) McGraw-Hill
- Lenguajes de programación, principios y práctica. 2^a Ed (2004). Kenneth C.Louden

