Sistemas y Aplicaciones Informáticas

Tema 34. 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.

1. Ámbito de docencia.

- Sistemas informáticos monousuario y multiusuario (ASI 1).
- Sistemas informáticos multiusuario y en red (DAI 1).
- Sistemas operativos en entornos monousuario y multiusuario (ESI 1).

2. Lenguaje C: Características generales.

- El lenguaje C está íntimamente relacionado con el desarrollo de UNIX. A pesar de este hecho, el lenguaje C no es exclusivo de UNIX, por lo que se ha denominado lenguaje de programación de sistemas. El C fue diseñado en 1972 por Dennis Ritchie a partir de los lenguajes B y BCPL, que a su vez descendían del ALGOL. Más tarde, en 1983 se definió el estándar ANSI C.
- C es un lenguaje de medio nivel que se caracteriza por lo siguiente:
  - Es de pequeño tamaño, tiene pocas reglas de sintaxis y un conjunto reducido de órdenes.
  - Su velocidad de ejecución es alta ya que se encuentra cerca de la máquina.
  - Es poco estricto con los tipos de datos, es decir, permite ver los datos como tipos distintos en determinadas fases de un programa.
  - Es un lenguaje estructurado de carácter modular, con compilación y enlazado separados.
  - Permite realizar manipulación directa de bits y su código es altamente portable.
  - Tiene una biblioteca extensa de funciones especiales que complementan las órdenes nativas.
  - No existe verificación de memoria (tamaños de arrays) en tiempo de ejecución.

3. Elementos del lenguaje.

3.1. Identificadores, palabras clave y sentencias.

- **Identificadores**. Son los nombres que se utilizan para representar variables, constantes, tipos de datos y funciones de un programa. 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 cuaquier 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 de una función definida en una librería, y deben ser significativos del valor que contienen.
- **Palabras clave**. Son identificadores predefinidos que tienen un significado especial para el compilador de C. Sólo pueden usarse como se han definido, y siempre van en minúsculas.
- **Sentencias**. Existen dos tipos de sentencias:
  - *Sentencia simple*. Es una expresión válida en C acabada en punto y coma (`;`).
  - *Sentencia compuesta*. Son grupos de sentencias que se tratan como si fueran una única sentencia. Comienzan con apertura de llave (`{`) y terminan con cierre de llave (`}`).

3.2. Tipos de datos simples.

- Los tipos de datos simples en C son:

    - `char`. Define un número entero que ocupa un byte con el bit más significativo como bit de signo, por lo cual el rango de valores que puede contener va de -128 a 127.
    - `int`. Sirve para definir números enteros en complemento a dos con signo. Su representación en memoria es de dos bytes, y el rango de valores es de -32768 a 32767.
    - `float`. Se usa para representar números reales en coma flotante utilizando el formato real IEEE de 32 bits (1 bit de signo, 8 bits de exponente y 23 bits de mantisa).
    - `double`. Se usa para representar números reales en coma flotante utilizando el formato real IEEE de 64 bits (1 bit de signo, 11 bits de exponente y 52 bits de mantisa).
- A estos tipos de datos se les pueden aplicar tres tipos de modificadores:
    - `signed`. Guarda un bit para el signo. Si no se especifica, se asume por defecto.
    - `unsigned`. No guarda un bit para el signo. Debe especificarse explícitamente.
    - `long`. Duplica el espacio de memoria del tipo de dato al que modifica.

3.3. Variables. Declaración, inicialización y alcance.

- Las variables deben estar previamente declaradas antes de poder hacer referencia a las mismas:

  tipo nombre_variable1[=valor1] [, nombre_variable2[=valor2]…];

  donde *tipo* es un tipo de datos válido en C con los modificadores necesarios, *nombre_variable1* y *nombre_variable2* son los identificadores de las variables, y *valor1* y *valor2* los respectivos valores iniciales de *nombre_variable1* y *nombre_variable2*.
- 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.
- 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.

3.4. Constantes.

- Las constantes se refieren a valores fijos que no pueden alterarse en el programa. Pueden definirse constantes 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_constante1[=valor1] [, nombre_constante2[=valor2]…];

3.5. Operadores.

3.5.1. Operador de asignación.

- La forma general del operador de asignación es la siguiente:

  nombre_variable = expresión;

  donde *expresión* pueden ser simple o compuesta. También se puede usar el operador asignación precedido de cualquiera de los siguientes operadores: `+`, `-`, `*`, `/`, `%`, `>>`, `<<`, `&`, `^`, `|`. Dadas dos expresiones *expresión1* y *expresión2*, las siguientes sentencias son equivalentes:

  expresión1 operador = expresión2;

  expresión1 = expresión1 operador expresión2;

- Cuando se mezclan variables de un tipo con variables compatibles de otro tipo distinto se produce conversión automática de tipos. El valor del lado derecho de la asignación se convierte al tipo de dato del lado izquierdo de la asignación.

3.5.2. Operadores aritméticos.

- Los operadores aritméticos son suma (`+`), resta (`-`), multiplicación (`*`), división (`/`), resto de la división entera (`%`), incremento (`++`) y decremento (`--`).
- El operador división (`/`) produce una división entera cuando se aplica a un `char` o un `int` y el resto se trunca, en el resto de casos se obtiene una división real. El operador resto de la división entera (`%`) no puede aplicarse a los tipos `float` o `double`.
- Los operadores incremento (`++`) y decremento (`--`) incrementan o decrementan respectivamente en uno el valor de la variable a la que se aplican. Existe una diferencia en el uso de estos operadores cuando preceden a la variable (*preincremento* o *predecremento*) o cuando están colocados detrás de la variable (*postincremento* o *postdecremento*):
  - *Preincremento* o *predecremento*. Se lleva a cabo la operación de incremento o decremento antes de usar el valor de la variable en la expresión en la que se encuentra.
  - *Postincremento* o *postdecremento*. Se lleva a cabo la operación de incremento o decremento después de usar el valor de la variable en la expresión en la que se encuentra.

3.5.3. Operadores relacionales y lógicos.

- No existe el tipo booleano en C, el valor TRUE equivale a cualquier valor distinto de cero y el valor FALSE equivale a cero. El resultado de una expresión relacional o lógica es cero (FALSE) o uno (TRUE). Las expresiones asociadas por operadores lógicos se evalúan de izquierda a derecha, una vez establecida la precedencia.
- Los operadores relacionales son mayor que (`>`), mayor o igual que (`>=`), menor que (`<`), menor o igual que (`<=`), igual (`==`) y distinto (`!=`), y los lógicos son AND (`&&`), OR (`||`) y NOT (`!`).

3.5.4. Operadores a nivel de bit.

- Las operaciones a nivel de bit se refieren a la comprobación, asignación o desplazamiento de los bits reales que componen un byte o palabra. No se aplican a los tipos `float`, `double`, `void` u otros tipos de datos complejos, y se aplican para operaciones de control de dispositivos.
- Los operadores a nivel de bit son AND (`&`), OR (`|`), XOR (`^`), complemento a uno NOT (`~`), desplazamiento a la derecha (`>>`) y desplazamiento a la izquierda (`<<`).

3.5.5. Operadores especiales.

- `?`. Este operador sustituye a expresiones `if-else`. La sentencia:

  variable = expresión1 ? expresión2 : expresión3;

  evalúa *expresión1*, si es TRUE le asigna a *variable* el valor de *expresión2*, si es FALSE le asigna a *variable* el valor de *expresión3*. Tanto *expresión2* como *expresión3* pueden ser funciones.
- `&`. Operador monario que devuelve la dirección de memoria de su operando.
- `*`. Operador monario que devuelve el contenido de la dirección de memoria de su operando.
- `sizeof()`. Es un operador monario que devuelve la longitud en bytes de la variable o del especificador de tipo al que precede. Con este operador se asegura la portabilidad, al no depender la aplicación del tamaño del tipo de datos de una máquina concreta.

3.6. Expresiones.

- Una expresión en C es cualquier combinación válida de operadores, constantes y variables. Las expresiones válidas siguen las reglas del álgebra, y las subexpresiones de una expresión pueden ser evaluadas en cualquier orden.
- El compilador, en una expresión, convierte todos los operandos al tipo de dato del mayor operando. Es posible forzar el tipo de dato de una expresión a través de un casting, que se representa de la siguiente manera: *(tipo_dato) expresión;*

3.6.1. Sentencias de selección.

3.6.1.1. Selección simple: if-else. Anidamiento.

- La sentencia de selección simple `if-else` en C tiene la siguiente sintaxis:

```
if (expresión){
      sentencias1;
} [else {
      sentencias2;
}]
```

- Las sentencias dentro de cada bloque pueden ser simples o compuestas y la cláusula `else` es opcional. Si la expresión evaluada es cierta (distinta de cero), se ejecutarán `sentencias1` y si el falsa se ejecutarán `sentencias2`. El punto y coma (`;`) es parte de la sentencia y no es un separador de sentencias. Por tanto, antes del `else` es necesario punto y coma.
- Cuando una sentencia de control aparece dentro de otra sentencia de control del mismo tipo, se produce un anidamiento. Los anidamientos pueden llegar a cualquier nivel tanto en la parte `if` como en la parte `else`. La claúsula `else` se asocia al `if` más próximo en el bloque en el que se encuentre que no tenga asociado ya un `else`.
- La estructura `if-else-if` es un caso particular de `if` anidado con la siguiente sintaxis:

```
if (expresión1){
      sentencias1;
} else if (expresión2){
      sentencias2;
…
} else {
      sentencias_n;
}
```

3.6.1.2. Selección múltiple: switch. Anidamiento.

- Se compone de las palabras clave `switch`, `case`, `default` y `break`. Compara el valor de una expresión sucesivamente con una lista de constantes enteras o de caracteres. Cuando la expresión coincide con una determinada constante, ejecuta las sentencias asociadas a ésta:

```
switch (expresión1){
      case constante1: secuencia_sentencias1; break;
      case constante2: secuencia_sentencias2; break;
      …
      [default: secuencia_sentencias]
}
```

- La sentencia `break` hace que el programa salte a la línea de código siguiente a la sentencia `switch`. Si se omite, se ejecutará el resto de casos case hasta encontrar el próximo `break`. La sentencia `default` se ejecuta cuando no se produce ninguna coincidencia. No puede haber dos sentencias `case` con el mismo valor de constante. Una constante `char` se convierte automáticamente a sus valores enteros.
- Cuando la comparación se basa en variables o se trabaja con expresiones que devuelven `float`, se debe utilizar `if-else`.La secuencia de sentencias en un `case` no es un bloque y no tiene que ir entre llaves. Por tanto, no se podría definir una variable local dentro de ellas. Sin embargo, la estructura `switch` global sí que es un bloque. Además, es posible anidar sentencias `switch`.

3.6.2. Sentencias de iteración.

3.6.2.1. Sentencia while. Anidamiento.

- La sentencia `while` tiene la siguente sintaxis:

```
while (condición){
      sentencias;
}
```

- La condición es cualquier expresión válida en C y se evalúa antes de la ejecución del bloque de sentencias. El bucle se repite mientras el valor de la condición sea verdadera (distinta de cero). Si es falsa (igual a cero) el programa ejecutará la siguiente línea de código que sigue a la estructura. Es necesario asegurar que en algún momento no se cumple la condición para salirse del bucle.
- La sentencia `while` puede no tener cuerpo. Se usa por ejemplo para generar retardos. Puede contener a su vez cualquiera de las sentencias de iteración.

3.6.2.2. Sentencia for. Anidamiento.

- La sintaxis general de la sentencia `for` es la siguiente:

```
for (expresión1; expresión2, expresión3){
      sentencias;
}
```

que es equivalente a:

```
expresión1;
while (expresión2){
      sentencias;
      expresión3;
}
```

- Están permitidos los bucles `for` con varias variables de control utilizando el operador coma.

```
for (x = 0, y = 0; x+y < 100; x++, y+=2){
      sentencias;
}
```

- En los bucles `for` la evaluación de la condición se realiza al principio de la ejecución. El bucle se repite hasta que la condición sea falsa. En ese momento el programa continua en la siguiente sentencia al `for`. Puede contener a su vez cualquiera de las sentencias de iteración.

3.6.2.3. Sentencia do-while. Anidamiento.

- La sentencia `do-while` tiene la siguente sintaxis:

```
do {
      sentencias;
} while (condición);
```

- La condición es cualquier expresión válida en C y se evalúa después de la ejecución del bloque de sentencias. El bucle se repite mientras el valor de la condición sea verdadera (distinta de cero). Si es falsa (igual a cero) el programa ejecutará la siguiente línea de código que sigue a la estructura. La diferencia con `while` y con `for` es que el bloque de sentencias se ejecutará al menos una vez. Puede contener a su vez cualquiera de las sentencias de iteración.

3.6.3. Sentencias de salto.

- `break`. Permite interrumpir la secuencia lógica del programa. Tiene dos usos:
  - Situado en una sentencia `switch` para finalizar un `case`.
  - Para salir de un bucle en el cual se ha cumplido una condición especial. El programa continuará en la siguiente línea de código que sigue al bucle. En el caso de bucles anidados la sentencia `break` causa la salida únicamente del bloque en el que se encuentre.
- `continue`. Sólo se puede utilizar en los bucles `for`, `while` y `do-while`. Su aplicación es similar a la sentencia `break`, con la diferencia de que `continue` hace que el bucle salga de la iteración en curso, como si se hubiese llegado a la última instrucción del bucle.

4. Estructura de un programa.

- En general un programa en C tiene la siguiente estructura:

```
#directivas de preprocesador
declaración de prototipos de funciones;
typedefs de macros, estructuras y variables enumeradas;
declaración de variables globales;

int main(int argc, char **args){

      declaración de variables locales;
      código;
}

int funcion(parámetros formales){

      declaración de variables locales;
      código;
}
```

- Un comentario es todo aquel texto que se inicia con `/*` y finaliza con `*/`. Puede estar en cualquier parte del código. Debe ser significativo y puede ocupar varias líneas.

- Toda sentencia que comienza por # se denomina directiva de preprocesador. Una de estas directivas es `#include <xxxx.h>`, por la cual se indica al compilador que incluya el código del archivo `xxxx.h` dentro del código fuente del programa.
- Los archivos `xxxx.h` se denominan archivos de cabecera y suelen contener identificadores, constantes, variables globales, macros y prototipos de funciones. Permiten tener las declaraciones fuera del programa principal, lo que implica mayor modularidad.
- Todo programa en C debe tener una función `main` desde la que siempre comienza la ejecución del programa, y desde la que se realizan llamadas al resto de funciones del programa.

5. Funciones de librería y usuario.

5.1. Definición. Declaración y llamada.

- Las funciones son bloques de código en los cuales se realiza una tarea específica. Un programa en C está formado por la función `main` que es el bloque principal, por funciones propias del programador y por funciones de librerías propias del compilador.
- La definición general de una función en C es la siguiente:

```
tipo nombre_función(tipopar1 variable1, tipopar2 variable2, …) {
    cuerpo de la función;
    [return()];
}
```

  donde *tipo* es el tipo de datos que devuelve la función en la sentencia *return()* y la lista de parámetros formales está formada por variables separadas por comas con sus tipos asociados, las cuales reciben los valores de los argumentos cuando se llama a la función. Si no se especifica *tipo*, por defecto devuelve un entero. La lista de parámetros puede estar vacía, aunque los paréntesis de la función deben colocarse de igual manera. Es aconsejable escribir las definiciones de las funciones a continuación de la función `main`.
- La declaración de la función se denomina prototipo de la función, y tiene la siguiente sintaxis:

```
tipo nombre_función(tipopar1, tipopar2, …);
```

  donde *tipopar* es el tipo de los parámetros usados en la definición de la función. El prototipo se coloca antes de la función `main`. Permite localizar cualquier conversión ilegal de tipos entre los los argumentos utilizados en la llamada a la función y los tipos definidos para los parámetros.
- La llamada a una función es una sentencia formada por el nombre de la función y los argumentos que se le pasan entre paréntesis. Tienen lugar dentro del `main` o en el cuerpo de otra función.

5.2. Retorno y devolución de valores. Reglas de ámbito.

- Una función termina su ejecución y vuelve al punto donde se la llamó cuando se encuentra la llave del final, o cuando se le fuerza la salida antes del final de su ejecución con la sentencia `return`. Puede utilizarse para devolver un valor. Una función puede contener varias sentencias `return`. La sintaxis es la siguiente: *return(expresión);*
- Los paréntesis son opcionales. El valor de salida de la función se especifica en la sentencia `return`, si no se incluye esta sentencia el valor queda indefinido. La expresión debe ser válida

en C, y si el tipo de datos de la expresión no coincide con el tipo de datos que debe devolver la función, automáticamente se convierte el tipo de datos para que exista coincidencia.

- Todas las funciones pueden devolver variables de cualquier tipo de dato, excepto aquellas declaradas como `void`. Una función que devuelva un valor se puede usar como operando en cualquier expresión válida en C. Sin embargo, no puede estar a la izquierda en una asignación.
- Las reglas de ámbito hacen referencia al conocimiento que sobre un bloque del programa tiene otro bloque del programa. En C el código y los datos son privados. El resto de las funciones no pueden interactuar con partes del código de una función ni con sus datos. Una excepción a la regla de ámbito de los datos son las variables globales y estáticas.

5.3. Argumentos de las funciones. Llamadas por valor y por referencia.

- Lo normal es emplear funciones con argumentos, para lo cual se deben declarar variables denominadas parámetros formales que recojan los valores de los argumentos. Si no se esperan argumentos se utiliza `void` en el lugar de los parámetros formales en la declaración.
- Los parámetros formales son variables locales que se crean cuando se ejecuta la función y se destruyen al acabar la misma. Es necesario asegurar que el tipo de datos de los parámetros formales es compatible con el tipo de datos usado para los argumentos de llamada a la función. C en general no dará error si esto no se controla, pero se puede producir un resultado inesperado.
- Se pueden pasar argumentos a las funciones de dos maneras:
  - *Por valor*. En este tipo de llamada se copia el valor del argumento en el parámetro formal de la función. Por tanto todas las modificaciones que se realicen en el valor del parámetro no influyen en el valor de las variables usadas como argumento en la llamada de la función.
  - *Por referencia*. En este caso se copia la dirección del argumento en el parámetro formal de la función, por tanto el parámetro formal debe declararse como puntero. Por tanto todas las modificaciones que se realicen en el valor del parámetro afectan al valor de las variables usadas como argumento en la llamada de la función.
- Cuando se pasa un array como argumento se pasa la dirección del primer elemento del array, es decir, se pasa por referencia. La llamada a la función usará el nombre del array como argumento, puesto que el nombre del array en sí mismo es un puntero al primer elemento del array. Para declarar un parámetro formal que va a recibir un puntero a un array se declara así:

  tipo nombre_función(tipopar par_array[]);

  siendo *tipopar* el tipo de dato del array y *par_array[]* el parámetro formal del array. Dentro de la función, el parámetro formal se maneja con índices como un array normal.
- Cuando se pasa una estructura como argumento, se pasa la estructura íntegra por valor.

6. Entorno de compilación.

6.1. Compilación y regiones de memoria.

- La compilación se produce en tres pasos:
  - Creación del código fuente como un fichero de texto estándar con extensión “.c”.
  - Compilación del código fuente y creación del código objeto.
  - Enlazado del código objeto con el código objeto de las librerías empleadas.

- Al ejecutar el programa, se crean distintas regiones de memoria:
  - *Código del programa*. Almacena instrucciones en codigo máquina de las funciones y los procedimientos, y su tamaño puede fijarse en tiempo de compilación.
  - *Memoria estática*. Almacena las variables globales y constantes (enteros, reales, strings, ...), variables estáticas (`static`), direcciones reservadas por el sistema operativo y código y datos estáticos cargados desde librerías o módulos precompilados.
  - *Memoria dinámica*. Zona de tamaño fijo, que se reparte dinámicamente entre:
    - **Pila**. Mantiene las variables locales e información de control adicional de los procedimientos que se han llamado. Esta zona comienza en la dirección más baja de la región de memoria dinámica, y crece hacia las direcciones más altas en las llamadas a procedimientos y decrece hacia las direcciones más bajas al retornar de las llamadas.
    - **Heap**. Guarda las variables cuyo tamaño varía en tiempo de ejecución o que no se pueden mantener en memoria estática ni en la pila. Esta zona comienza en la dirección más alta de la región de memoria dinámica, y crece hacia las direcciones más bajas. Su espacio puede ser asignado y desasignado en cualquier momento.

6.2. Preprocesador. Compilación separada.

- En C se pueden incluir instrucciones dirigidas al compilador en el código fuente. Estas instrucciones se denominan directivas de preprocesamiento, que están siempre precedidas del símbolo #. No puede haber dos directivas de preprocesamiento en la mísma línea.
- Las directivas más utilizadas son las siguientes:
  - `#define`. Permite sustituir una expresión por un identificador denominado macro. Una macro previamente definida puede ser usada como parte de otra directiva `#define`. La expresión puede ser una cadena y puede tener argumentos. La sintaxis es la siguiente:

    #define nombre_macro expresión
  - `#include`. Hace que el compilador incluya el código de un archivo fuente dentro del código fuente del programa. El uso de comillas dobles en el nombre del archivo indica al compilador que busque el archivo en el directorio actual de trabajo, mientras que el uso de <> indica al compilador que busque el archivo en los directorios de las librerías estándar.
  - `#ifdef, #ifndef`. Comprueban si está definida una macro con `#define`:

    #ifdef nombre_macro
    secuencias;
    #endif

    #ifndef nombre_macro
    secuencias;
    #endif

    Se compilarán las secuencias de instrucciones si se ha definido o no respectivamente la macro. Se suelen utilizar en los archivos de cabecera para no duplicar declaraciones.
- La compilación separada es el procedimiento por el cual las partes de un programa se compilan independientemente y se enlazan para formar el programa final. La salida del compilador es un archivo de código objeto reubicable, y la salida del enlazador es un archivo ejecutable.
- El enlazador primero combina físicamente los archivos especificados dentro de la lista de enlace en un archivo del programa. Después resuelve las referencias externas, es decir, decide las direcciones de salto y llama a instrucciones situadas en el formato reubicable del archivo objeto.

7. Herramientas para la elaboración y depuración de programas.

7.1. 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.

7.2. 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.

7.3. 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 '\'.

7.4. 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*.