Sistemas y Aplicaciones Informáticas

Tema 28. Programación Modular. Diseño de Funciones: Recursividad. Librerías.

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. Programación modular.

2.1. Conceptos de módulo, procedimiento y función.

- Un módulo es un segmento de programa que recibe una serie de datos y devuelve un resultado o realiza alguna acción. Según esta definición, el concepto de módulo engloba a un programa completo, a una unidad de compilación independiente, a un procedimiento o a una función.
- Un procedimiento es un módulo que realiza una determinada tarea, desempeñando el papel de una instrucción. Una función es un módulo que siempre devuelve un valor de un determinado tipo, desempeñando el papel de una expresión.
- En un módulo hay que distinguir dos aspectos fundamentales:
  - **Definición**. Es la especificación del nombre del módulo, seguido del nombre y el tipo de dato de cada uno de los parámetros. Incluye además una parte de declaraciones de variables locales y una parte de instrucciones que indican las acciones a realizar.
  - **Llamada**. Es una sentencia que pasa el control de un módulo a otro. Cuando el módulo llamado acaba su ejecución, el control vuelve a la sentencia siguiente a la llamada.

2.2. Programación modular.

- La programación modular es una técnica complementaria a la programación estructurada, consistente en aplicar el diseño descendente y separar cada tarea sencilla en un módulo. Esto reduce la complejidad del programa, elimina código duplicado y facilita su legibilidad.
- Los módulos pueden ser invocados desde cualquier parte del código y su funcionamiento no depende del resto del programa, por lo que es más fácil encontrar los errores y realizar el mantenimiento. El programador se concentra en codificar cada módulo haciendo más sencilla esta tarea. Para dar lugar a la aplicación final debe realizarse una integración de módulos.

2.3. Parámetros.

2.3.1. Parámetros actuales y formales. Correspondencias.

- Los datos externos que necesita un procedimiento o una función para realizar sus tareas se pasan a través de un número limitado de parámetros en el momento de la llamada. Cada parámetro está definido por un nombre y un tipo de dato asociado.
- En la definición de un procedimiento o una función, los datos genéricos sobre los que deben actuar se expresarán mediante una lista de parámetros formales. En la llamada, los datos concretos sobre los que actúan se expresarán mediante una lista de parámetros actuales.
- Al llamar un procedimiento o una función, se produce una correspondencia entre los parámetros formales y los parámetros actuales:
  - *Correspondencia posicional*. Se emparejan según la posición que tengan en la definición del procedimiento o de la función, y en la llamada realizada respectivamente. Esto obliga a que coincidan el número de parámetros y el tipo de dato de cada asociación actual-formal.

- *Correspondencia por nombre*. La correspondencia entre los parámetros formales y los actuales se indica explícitamente en la llamada al procedimiento o a la función.

2.3.2. Tipos de parámetros.

- Cuando termina la ejecución de un procedimiento o una función, se puede conseguir que las modificaciones realizadas sobre los parámetros formales se reflejen sobre los correspondientes parámetros actuales de la llamada.
- Bajo esta consideración, se puede hablar de tres tipos de parámetros:
  - *Parámetros de entrada*. El parámetro actual de la llamada tiene un valor definido, que además es el valor que tomará inicialmente el correspondiente parámetro formal. Si al finalizar la acción no interesa que las posibles modificaciones del parámetro formal se reflejen sobre el parámetro actual asociado, deberá definirse como de entrada.
  - *Parámetros de salida*. El parámetro actual de la llamada no tiene un valor definido. Si al finalizar la acción interesa que las posibles modificaciones del parámetro formal se reflejen sobre el parámetro actual asociado, deberá definirse como de salida.
  - *Parámetros de entrada/salida*. El parámetro actual de la llamada tiene un valor definido, que además es el valor que tomará inicialmente el correspondiente parámetro formal. Si al finalizar la acción interesa que las posibles modificaciones del parámetro formal se reflejen sobre el parámetro actual asociado, deberá definirse como de entrada/salida.

2.3.3. Paso de parámetros.

- **Paso por valor**. Los parámetros formales declarados se inicializan con los valores de los parámetros actuales. De esta forma, se trabaja con una copia de los parámetros actuales que desaparece al finalizar el procedimiento o la función. Cualquier modificación de los parámetros formales no se refleja en los correspondientes parámetros actuales.
- **Paso por valor resultado**. Los parámetros formales reciben una copia de los valores de los parámetros actuales y al finalizar la ejecución del subprograma se realiza el proceso inverso.
- **Paso por referencia**. Los parámetros formales declarados se inicializan con la dirección de los parámetros actuales. De esta forma, la dirección en memoria del parámetro formal coincide con la del parámetro real y cualquier referencia a dicho parámetro formal en el interior del procedimiento o de la función realmente se refiere al contenido de esa dirección. Indirectamente se puede llegar a modificar el valor original del parámetro actual.

2.4. Ámbitos.

2.4.1. Estructura de bloques. Ámbito de procedimientos y funciones.

- El programa principal, los procedimientos y las funciones en él declarados, y los que a su vez pudieran declararse dentro de ellos, constituyen un conjunto de bloques anidados que determinan el ámbito de validez de las variables y de los procedimientos y funciones.
- El ámbito de un procedimiento o una función es el conjunto de todos aquellos procedimientos o funciones que pueden llamarle. Si un procedimiento o una función está declarado en otro procedimiento u otra función, se dice que el primero es descendiente del segundo, y que el segundo es ascendiente del primero.
- En general, el ámbito de un procedimiento o una función comprende lo siguiente:

- Los ascendientes pueden llamar a sus descendientes, pero los ascendientes indirectos no pueden llamar a descendientes indirectos.
- Cualquier procedimiento o cualquier función puede llamarse a sí mismo.
- Un procedimiento o una función puede llamar a todos los descendientes directos del principal o del ascendiente directo que hayan sido declarados con anterioridad.

2.4.2. Variables locales y globales. Efectos laterales y precedencia de nombre.

- Las variables locales son aquellas que sólo tienen significado dentro del procedimiento o de la función en que están definidas y no son visibles fuera de él.
- Las variables globales son aquellas que están declaradas fuera de cualquier procedimiento o función y que son visibles en todo el programa.
- Debe tenerse en cuenta lo siguiente:
  - Todas las variables usadas por un procedimiento o una función que no sean variables globales deben declararse como variables locales de ese procedimiento o esa función.
  - Los parámetros de un procedimiento o una función actúan como variables locales.
  - La información que se pasa entre procedimientos y funciones debe hacerse a través de una lista de parámetros y no a través de variables globales. Esto hace que sean más independientes y facilita tanto el diseño como la prueba de cada uno de ellos.
- Cuando se modifica una variable global en un procedimiento o una función, se dice que se produce un efecto lateral. Los efectos laterales deberán evitarse pues introducen dependencias indeseables entre procedimientos y funciones.
- En caso de que una variable local y una variable global tengan el mismo identificador, cuando se haga una referencia a éste siempre prevalecerán los datos que presenten un ámbito más reducido, es decir, las variables locales frente a las variables globales (precedencia de nombre).

3. Diseño de funciones.

3.1. Concepto. Objetivos.

- El diseño de funciones consiste en obtener la estructura modular y los detalles de proceso del sistema. Un buen diseño modular debe organizar la complejidad del problema para que el esfuerzo de desarrollo, prueba y mantenimiento pueda ser controlado y minimizado.
- Se trata de conseguir que la estructura modular cumpla las siguientes características:
  - *Módulos pequeños*. Si el tamaño de los módulos es reducido, las modificaciones afectarán a un mayor número de módulos, pero la cantidad de código a considerar será menor.
  - *Encapsulamiento*. Es necesario aislar los detalles de implementación de los módulos para evitar que las modificaciones en ellos afecten a la funcionalidad general.
  - *Jerarquía*. Se debe lograr un tipo de estructura jerárquica en árbol en donde los módulos de niveles medios y altos del diagrama ejerzan el trabajo de coordinación y manipulación de los módulos de niveles más bajos, que son los que deben realizar tareas de cálculo y edición.
  - *Independencia*. Cuanto mayor es la independencia de los módulos menor es el esfuerzo total necesario para su desarrollo. Por tanto, el diseño debe reducir la compartición de ficheros, de datos, de dispositivos y las llamadas desde o hacia otros módulos.

3.2. Diseño descendente. Abstracción funcional. Refinamiento.

- El diseño descendente consiste en dividir el problema en varios subproblemas que se pueden resolver por separado, para después recomponer los resultados y obtener la solución al problema.
- Los subproblemas se dividen a su vez en subproblemas más pequeños hasta que su solución pueda implementarse fácilmente, dando lugar a sucesivos módulos. Se aplica la técnica de abstracción funcional, mediante la cual en cada descomposición se supone que los subproblemas más pequeños están resueltos, dejando su realización al siguiente nivel de descomposición.
- Posteriormente se realiza el refinamiento de los módulos finales mediante la descripción de los pasos a llevar a cabo. Para que el refinamiento sea correcto hay que definir con precisión el cometido de cada módulo, garantizar su corrección, y ensamblar y organizar bien entre sí los subalgoritmos resultantes. El tamaño de un módulo depende de su función y de su aplicación.
- La utilización de estas técnicas de diseño tiene los siguientes objetivos básicos:
  - Reducción del tamaño de los módulos.
  - Hacer el sistema más fácil de entender y modificar.
  - Minimizar la duplicidad de código.
  - Crear módulos que puedan utilizarse en cualquier otra parte.

3.3. Evaluación del diseño.

3.3.1. Acoplamiento. Niveles.

- Se puede definir acoplamiento como la medida de la interacción de los módulos que constituyen un programa. El objetivo del diseño es conseguir un acoplamiento mínimo por lo siguiente:
  - Cuantas menos conexiones existan entre dos módulos, menos oportunidad habrá de que un defecto o una modificación en un módulo afecte al otro.
  - Cuando se está modificando un módulo, es deseable no necesitar preocuparse en los detalles internos de cualquier otro módulo.
- Existen distintos niveles de acoplamiento, que se describen de mejor a peor a continuación:
  - *Acoplamiento normal de datos*. Se produce cuando todo lo que comparten dos módulos se especifica en la lista de parámetros del módulo llamado.
  - *Acoplamiento normal por estampado*. Ocurre si en la comunicación entre módulos se pasan datos con estructura de registro. Este acoplamiento no es deseable si el módulo que recibe el registro sólo necesita una parte de los elementos que se le pasan.
  - *Acoplamiento normal de control*. Ocurre si un módulo pasa datos a otro módulo que le indican lo que debe hacer, ya que implica que el módulo superior conoce detalles internos del módulo inferior. Es tolerable si el dato de control tiene sentido ascendente ya que en ese caso sirve para informar de la situación de terminación del módulo inferior.
  - *Acoplamiento externo*. Se produce cuando dos rutinas utilizan los mismos datos globales o dispositivos de E/S. Si los datos son de sólo lectura, el acoplamiento se puede considerar aceptable. En general, no es deseable porque la conexión entre los módulos no es visible.
  - *Acoplamiento de contenido*. Ocurre cuando un módulo utiliza el código de otro o altera sus datos locales. Si ocurre esto, habrá que evitarlo descomponiendo el módulo al que se accede o duplicando esa parte de código en el módulo que llama.

3.3.2. Cohesión. Niveles.

- Se puede definir cohesión como la medida del grado de identificación de un módulo con una función concreta. El objetivo del diseño es conseguir una cohesión máxima.
- Existen distintos niveles de cohesión, que se describen de mejor a peor a continuación:
  - *Cohesión funcional*. Ocurre cuando un módulo ejecuta una tarea funcional en un programa y requiere poca interacción con el resto de módulos.
  - *Cohesión secuencial*. Se produce cuando un módulo realiza distintas tareas en secuencia, de forma que las entradas de cada tarea son las salidas de la anterior.
  - *Cohesión de comunicación*. Ocurre cuando un módulo desempeña varias tareas paralelas usando los mismos datos de entrada y de salida.
  - *Cohesión procedimental*. Se produce cuando un módulo contiene operaciones que se realizan en un orden concreto aunque no tengan nada que ver entre sí.
  - *Cohesión lógica*. Ocurre cuando un módulo contiene operaciones cuya ejecución depende de forma condicional del valor de un parámetro.
  - *Cohesión coincidental*. Se produce cuando las operaciones de un módulo no guardan ninguna relación observable entre ellas, mezclando elementos asociados con distintas tareas.

4. Recursividad.

4.1. Concepto. Casos base y recursivo. Tipos.

- La recursividad es la posibilidad de que un algoritmo se invoque a sí mismo. Es útil cuando la solución de un problema se puede expresar en términos de la resolución de un problema de la misma naturaleza, aunque de menor complejidad.
- Un módulo se denomina recursivo cuando en el cuerpo del módulo existe una llamada a sí mismo. Todo módulo recursivo necesita una condición de parada para evitar una recursividad infinita. La condición de parada se denomina caso base y la llamada a sí mismo se denomina caso recursivo. Ni el caso base ni el caso recursivo son necesariamente únicos.
- Existen los siguientes tipos de módulos recursivos:
  - *Lineales*. Aquellos en los que cada llamada recursiva genera sólo otra llamada recursiva.
  - *Múltiples*. Aquellos en los que cada llamada recursiva genera varias llamadas recursivas.
  - *Finales*. Aquellos cuyo valor de retorno es el resultado de lo que se haya obtenido de la llamada recursiva, sin combinación de resultados parciales.
  - *Indirectos*. Son aquellos que llaman a otros y éstos a su vez a los primeros.

4.2. Diseño de módulos recursivos.

- En primer lugar, se debe obtener la solución al problema para los casos base sin emplear recursividad. Siempre debe existir algún caso base.
- En segundo lugar, se debe descomponer el problema de forma que se pueda obtener fácilmente la solución al caso general a partir de cálculos realizados con resultados obtenidos por el propio módulo, avanzando siempre hacia el caso base. Las llamadas recursivas simplifican el problema, y en última instancia los casos base sirven para obtener la solución. Se supondrá que las llamadas recursivas satisfacen la especificación del propio módulo que se está construyendo.

4.3. Recursividad e iteración. Consideraciones.

- La recursividad es un método algorítmico alternativo a la iteración cuando se trata de repetir cálculos. Cualquier estructura iterativa es posible sustituirla por un módulo recursivo y viceversa, aunque en ocasiones esta conversión puede ser bastante complicada.
- Normalmente un módulo iterativo es más eficiente que uno recursivo, pero la versión recursiva puede ser más recomendable por claridad del código. Cuando existe recursividad múltiple puede ocurrir que muchas llamadas se repitan, por lo que la versión recursiva no resulta recomendable.
- Es necesario considerar una serie de aspectos para decidir una implementación u otra:
  - El tiempo de CPU y el espacio en memoria asociados a las llamadas recursivas.
  - Algunas soluciones recursivas resuelven un problema en repetidas ocasiones.
  - La complejidad de la solución iterativa.
  - La concisión, legibilidad y elegancia del código resultante de la solución recursiva.

4.4. Ejemplos.

- Un ejemplo de programas recursivos en C, el factorial y la potencia:

```
int factorial (int n) {
   int resultado;
   if (n==0) */Caso base/*
          resultado = 1;
   else */Caso general/*
          resultado = n*factorial(n-1);
   return (resultado);
}

int potencia (int base, int exp) {
   int resultado;
   if (exp==0) */Caso base/*
          resultado = 1;
   else */Caso general/*
        resultado = base * potencia(base,exp-1);
   return (resultado);
}
```

- Casi todos los algoritmos basados en los esquemas de vuelta atrás (búsqueda en árboles) y divide y vencerás (búsqueda binaria, quicksort) son recursivos.

5. Librerías.

5.1. Concepto. Características.

- Una librería es un archivo en el que se encuentra almacenado el código correspondiente a un conjunto de módulos independientes entre sí, con el fin de ser utilizadas por otros programas.
- Se encargan de proporcionar una colección básica de estructuras de datos, funciones y procedimientos independientes del tipo de aplicación. Esta colección debe ser suficiente para cubrir las necesidades de la mayoría de las aplicaciones en los lenguajes que permitan su uso.
- Las librerías tienen las siguientes características:
  - No tienen estructura de programa ejecutable y tampoco son ejecutables de modo independiente, sino que la ejecución de sus funciones y procedimientos se realiza mediante la llamada de un programa.

- Cuando en un programa se necesita utilizar un módulo de librería, sólo será necesario realizar la llamada al módulo, independientemente del lugar donde se encuentre definido.
- Las librerías facilitan la reutilización de código y liberan al programador de tareas de codificación tediosas o de bajo nivel. También facilitan la portabilidad del software cuando diferentes sistemas utilizan un mismo estándar de codificación.

5.2. Enlace estático y enlace dinámico.

- Se denomina enlace al acceso por parte de un programa al código de un módulo de una librería.
- En función del tipo de enlace, existen dos tipos de librerías:
  - *Las estáticas, o de enlace estático*. Son aquellas en las que el código de los módulos se incorpora en el enlazado del código objeto durante la compilación. Esto da como resultado un archivo ejecutable que incluye todos los símbolos y módulos respectivos.
  - *Las compartidas, o de enlace dinámico*. Son aquellas en las que el código de los módulos se incorpora en tiempo de ejecución. Esto permite que, a partir de una única copia en memoria por cada librería, los programas puedan compartir las funciones incluidas en ellas.
- En Windows, los archivos de librerías dinámicas poseen extensión .DLL (Dynamic Link Library), mientras que las estáticas generalmente terminan en .LIB. En Unix y Linux, las librerías dinámicas tienen extensión .so (Shared Object) y las estáticas .a (archive).

5.3. Tipos de librería.

- En función de quién las crea, las librerías pueden ser:
  - *Librerías de usuario*. Son librerías creadas por el usuario para incluir un conjunto de funciones y procedimientos utilizados frecuentemente en distintos programas.
  - *Librerías estándar*. Son librerías creadas por los fabricantes de compiladores que incluyen funciones para realizar tareas de bajo nivel, como la entrada y salida de datos.
- En función de cómo han sido creadas, las librerías pueden ser:
  - *Librerías en código fuente*. Son aquellas cuyo código se incorpora a los programas fuente durante la compilación. El resultado es un código objeto que contiene la totalidad del código de las librerías, se utilicen todas sus funciones o no.
  - *Librerías precompiladas*. Son módulos objeto generados a partir del código fuente de la declaración y definición de las funciones. Tienen las siguientes características:
    - Su código se incorpora en el enlazado del código objeto durante la compilación. El enlazador extrae de estas librerías sólo el código correspondiente a las funciones utilizadas, añadiéndolo al programa ejecutable.
    - No es necesaria la presencia de las librerías para que el programa funcione.
    - El inconveniente es que se generan programas de gran tamaño. Este sistema suele ser utilizado para programas que no han de compartir recursos con otras aplicaciones.
  - *Librerías de enlace dinámico (DLL).* Son ficheros con código ejecutable que mantienen su independencia física del programa principal. Tienen las siguientes características:
    - La extracción del código correspondiente a una función se realiza en tiempo de ejecución. Cuando un programa realiza una llamada a una función de la librería, la dirección real de enlace es calculada y la función se enlaza dinámicamente con el

programa. Sólo se carga en memoria el código de la función llamada, liberándose el espacio ocupado por ésta cuando se retorna al programa que la llamó.

- Es imprescindible la presencia de estas librerías para que el programa funcione.
- El inconveniente principal es la pérdida de velocidad en la ejecución. La utilidad principal de estas librerías es evitar programas ejecutables de tamaño excesivo.

5.4. Gestor de bibliotecas estáticas ar.

- El gestor de bibliotecas estáticas de Linux es *ar*, y sus funciones son crear bibliotecas (cuyos ficheros tienen extensión *.a*) permitiendo añadir y extraer los ficheros objeto de su interior.
- La sintaxis es `ar [opción] biblioteca [ficheros]` donde `opción` puede ser:
  - `r`. Reemplazar ficheros en la biblioteca.
  - `c`. Crea la biblioteca si no existe.
  - `s`. Crea o actualiza el índice de ficheros de la biblioteca.
  - `x`. Extraer ficheros de la biblioteca.
  - `d`. Borrar ficheros de la biblioteca.
  - `t`. Mostrar una tabla con el contenido de la biblioteca.

5.5. Ejemplos.

- **Librería estándar de C**. ANSI C define sólo unas pocas palabras reservadas, por lo cual el conjunto de funciones que realizan tareas de bajo nivel y determinadas operaciones con datos están disponibles en el entorno de programación de C. Las funciones se declaran en ficheros de cabecera o .h. Las cabeceras contienen única y exclusivamente los prototipos de las funciones. El código o cuerpo de las funciones se incluye en archivos objeto que son realmente la librería.
- **Application Programming Interfaces (APIs)**. Son un conjunto de librerías de programación, que elaboran y publican los fabricantes de elementos tales como sistemas operativos o dispositivos hardware, para permitir a los programadores de aplicaciones utilizar los servicios y posibilidades de dichos elementos. Por extensión, se denomina API a cualquier grupo de funciones que son parte de ciertas aplicaciones pero que son utilizables desde otras aplicaciones.