LENGUAJES DE PROGRAMACIÓN

CODE UD2 – Tema 1

IES Plurilingüe Antón Losada Diéguez

Adrián Fernández González

Tabla de contenido

Lenguajes de programación

1. Introducción

Para que un ordenador realice algo, es necesario indicarle qué hacer mediante un programa. Un programa es un conjunto de sentencias o instrucciones finitas que el ordenador entiende y que le indican, paso a paso, las acciones a realizar. Estos programas se construyen en base a una serie de símbolos y reglas estipulados por un lenguaje de programación.

Un lenguaje de programación posee un conjunto de símbolos permitidos denominado léxico, una serie de reglas que indican cómo realizar las instrucciones denominada sintaxis, y unas reglas que determinan el significado de dichas instrucciones denominado semántica. Cada lenguaje tiene su propio léxico, sintaxis y semántica

Existen multitud de lenguajes de todo tipo, más o menos cercanos a un lenguaje natural o a un lenguaje máquina, cada uno con sus peculiaridades, sus ventajas e inconvenientes. Todos ellos tienen sus propias reglas y símbolos que han de ser traducidos a lenguaje máquina para que el ordenador los entienda.

2. Tipos y características

Aunque existen infinidad de lenguajes, todos pueden clasificarse teniendo en cuenta tres características fundamentales, su nivel de abstracción, su proceso de traducción al lenguaje máquina y su paradigma de programación.

2.1 Según nivel de abstracción

El nivel de abstracción es lo mucho que se diferencia del lenguaje máquina, es decir, lo mucho que abstrae al programador de la máquina que va a ejecutar el programa.

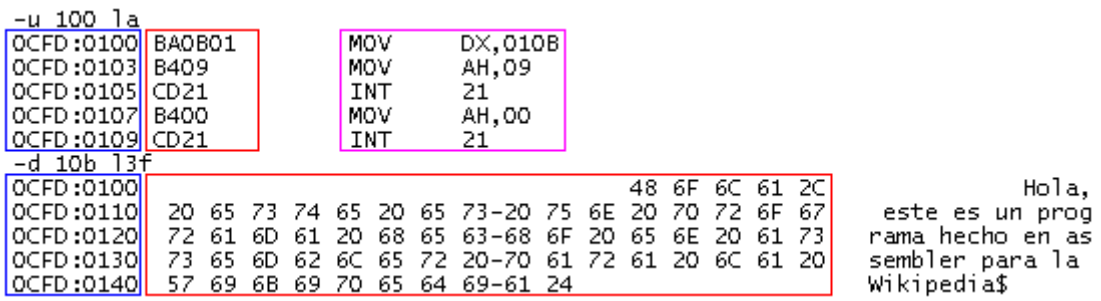
2.1.1 Lenguaje máquina o bajo nivel

Es el lenguaje que entiende el ordenador y está diseñado para los circuitos de la máquina en cuestión.

Su programación es muy tediosa, alejada del lenguaje natural y limitada, puesto que se realiza en binario y con unas instrucciones muy rígidas.

La memoria es controlada por el programador directamente, sin nombres de variables, accediendo a las posiciones de memoria directamente.

Tal como se comentó antes, está diseñado para esa CPU en concreto, por lo que el programa no funcionará en otra CPU distinta.



2.1.2 Lenguaje ensamblador o nivel medio

El lenguaje ensamblador fue la primera aproximación para facilitar la programación y acercarlo al lenguaje natural, humano.

Establece nombres significativos a las distintas operaciones, lo que facilita enormemente su uso y poder recordarlas con mayor facilidad. Algunos más avanzados, agrupan un conjunto de operaciones bajo una única instrucción, reduciendo el código a escribir.

El lenguaje ensamblador introdujo las variables, nombrar los datos para facilitar su uso y aumentar la legibilidad del código al no tener que referenciar continuamente una posición de memoria.

También añade la posibilidad de poner comentarios en el código, permitiendo explicarlo, aumentando enormemente su legibilidad.

Pese a todas estas mejoras, su uso sigue siendo limitado y tedioso y dependiente de la CPU.

Este tipo de lenguajes se utilizan para afinar partes importantes de un programa, ya que te da el control total de la máquina y su memoria, agilizado y optimizando al máximo dicho programa.

```
ORG    8030H
include
T05SEG:
   SETB TR0
   JNB uSEG,T05SEG ;esta subrutina es utilizada
   CLR TR0 ;para realizar una cuenta de
   CPL uSEG ;0,5 segundos mediante la
   MOV R1,DPL ;interrupción del timer 0.
       INVOKE
   MOV R2,DPH
   CJNE R2,#07H,T05SEG
   CJNE R1,#78H,T05SEG
   MOV DPTR,#0
RET
```

2.1.3 Lenguaje de alto nivel

Los lenguajes de alto nivel se abstraen del propio lenguaje de la máquina y su arquitectura, usando un lenguaje más cercano al humano.

Son portables, es decir, un mismo programa puede ser usado en distintas máquinas con arquitecturas diferentes.

Añaden multitud de instrucciones muy potentes que realizan diversas operaciones con una simple línea de código.

No solo permiten establecer nombres a las variables para facilitar su manejo y legibilidad, también permiten agrupar una serie de instrucciones bajo un mismo identificador para ser usadas múltiples veces a lo largo del programa.

Pueden incluirse comentarios para una mayor legibilidad del código.

Puesto que se alejan del lenguaje que entiende la máquina, los programas han de ser traducidos a un lenguaje de bajo nivel para que la máquina los entienda.

```
public class NodeList {
    //El propio elemento
    private int elem;
    //La referencia al siguiente elemento
    private NodeList next = null;

    public NodeList(int elemento)
    {
        this.elem = elemento;
    }

    //Getters
    public int getElem()
    {
        return this.elem;
    }
    public NodeList getNext()
    {
        return this.next;
    }

    //Setters
    public void setElem(int elemento)
    {
        this.elem = elemento;
    }
    public void setNext(NodeList siguiente)
    {
        this.next = siguiente;
    }
}
```

2.2 Según proceso de traducción

2.2.1 Compilado

Los lenguajes compilados son aquellos que traducen todo el código fuente a un código objeto que es el que entiende el ordenador para luego ejecutarlo.

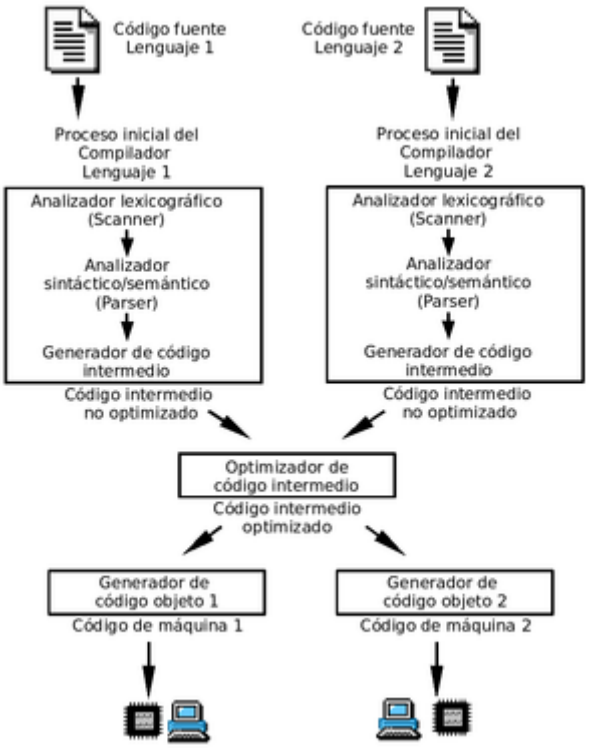
Este proceso se divide en dos etapas fundamentales:

Primero se analiza el código buscando posibles errores y se eliminan los comentarios, si todo está correcto, se genera un código intermedio.

Luego, se optimiza este código intermedio y si todo está correcto se genera el código objeto.

El código objeto generado es en lenguaje máquina, por lo que depende de la arquitectura y ha de ser generado en la máquina a utilizar, pero una vez hecho, no es necesario volver a compilarlo.

Para poder ejecutar el programa, detectar errores o probar el código es necesario compilar todo el programa antes, lo que retrasa el proceso, pero una vez está compilado, su ejecución es inmediata, eficiente y rápida.



2.2.2 Interpretado

Los lenguajes interpretados son traducidos a lenguaje máquina línea a línea a la par que son ejecutados.

Este tipo de lenguajes permiten modificar dinámicamente el código, ejecutarlo, probarlo y detectar errores rápidamente.

Tienen el problema de que son más lentos, puesto que necesitan traducir línea a línea y ejecutar todo tal cual, sin optimización de bloques de código.

Son más inseguros al no realizar análisis en bloque y por tanto se desconoce lo que hacen hasta que se ejecuta.

En cambio, permiten una mayor portabilidad, al pasar el código fuente y no una versión traducida, cada máquina se lo traduce al ejecutarlo.

2.2.3 Mixto o Just In Time (JIT)

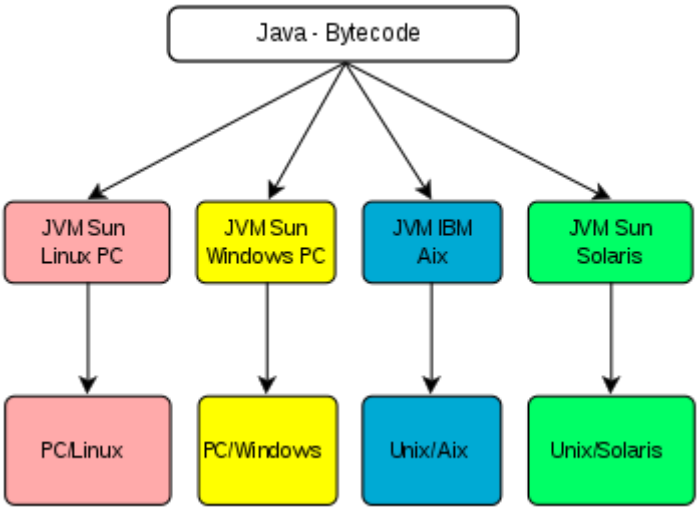
Son una alternativa intermedia entre un lenguaje compilado y uno interpretado.

Se basan en traducir dinámicamente el código fuente a un código intermedio denominado bytecode mientras se va programando. De esta forma, ofrecen las ventajas de la compilación

al analizar y optimizar el código, pero reduciendo el tiempo de espera para su ejecución y pruebas.

El bytecode, al ser intermedio, es portable, puede ser utilizado por cualquier CPU y al estar previamente analizado y optimizado, el proceso de traducción es menor que uno interpretado.

Dado que se realiza todo dinámicamente, tanto la traducción a bytecode mientras se programa como la traducción de este a máquina mientras se ejecuta, consumen una mayor cantidad de memoria.



2.3 Según el paradigma de programación

Un paradigma de programación es un marco conceptual para crear el programa. Un conjunto de ideas a seguir como el tipo de operaciones, la división en bloques de código o la forma en la que se estipula cada instrucción.

2.3.1 Imperativos

Son los lenguajes más habituales, en los que se estipulan una serie de instrucciones que indican al ordenador qué hacer y cómo.

2.3.1.1 Estructurados

Son uno de los primeros paradigmas y uno de los más básicos y sencillos, en los cuales se usan bloques de código, sentencias alterativas e iterativas.

Cualquier programa puede realizarse siguiendo este paradigma, pues es el paradigma base sobre el que se crearon los otros.

Puesto que utiliza estructuras básicas, carece de saltos, por lo que puede verse el flujo del programa de forma sencilla, al ser completamente secuencial.

Tienen el inconveniente de que cuanto mayor es el sistema, más difícil se hace seguir el código, puesto que carece de divisiones, de bloques de código asociados a una parte del sistema ni separación por ficheros.

Otro problema es que carecen de estructuras de repetición de código, que permitan reutilizar partes en distintas secciones de nuestro sistema, por lo que hay que volver a escribir una y otra vez lo mismo.

2.3.1.2 Modulares

Es una evolución del paradigma anterior que permite la división del código en bloques aumentando la legibilidad y la capacidad para reutilizar código. Se basa en la idea del divide y vencerás, la división de un problema en problemas más pequeños.

Fomenta el uso de los procedimientos y funciones. Estos reciben una serie de datos denominados parámetros, realizan una serie de operaciones y, en caso de las funciones, devuelven unos datos.

Este paradigma también defiende la idea de dividir el sistema en varios archivos con una funcionalidad propia y agruparlos en paquetes.

Este tipo de lenguaje es mucho más legible, puesto que divide la funcionalidad en diferentes partes y facilita la creación de sistemas más grandes.

2.3.1.3 Orientados a Objetos

Este paradigma se basa en la construcción de unos elementos denominados objetos que representan un ente del mundo real con sus características (atributos) y su comportamiento (métodos). Estos objetos tienen una identidad, es decir, cada uno tiene sus características propias aunque compartan comportamiento.

Este paradigma es un paso más en la modulación, puesto que no solo divide el sistema, también abstrae a las distintas partes del funcionamiento de las otras, aumentando la legibilidad y la reutilización de código.

Incluso, mediante relaciones jerárquicas entre los objetos, pueden crearse objetos que realizan las mismas operaciones de diferentes formas en función de las características del sistema. Esto permite, utilizando un mismo nombre de función (método) pero de diferente objeto, adaptarnos, por ejemplo, a diferentes sistemas de obtención de datos sin que todo el sistema se tenga que adaptar.

2.3.2 Declarativos

Los lenguajes declarativos son más cercanos al lenguaje natural, estipulan qué queremos, pero no cómo.

El más representativo es el SQL en el que indicamos qué valores queremos de dónde los queremos y con qué restricción sin estipular en ningún momento cómo se va a realizar dicha operación.

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

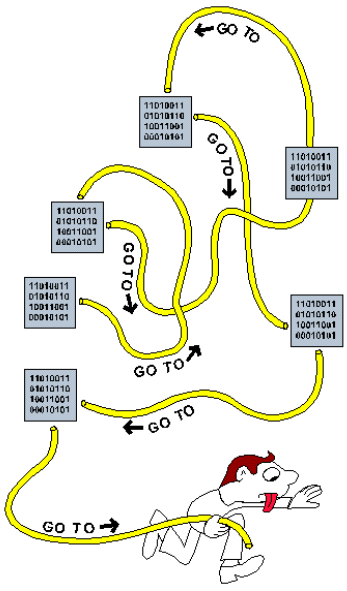
3. Programación estructurada

Con la aparición de los primeros lenguajes de alto nivel, se hizo más fácil programar, pero se empezó a descuidar mucho la eficiencia y seguía siendo difícil su seguimiento.

Una década más tarde, la situación es insostenible y varios expertos se reúnen para crear lo que posteriormente se denominaría ingeniería del software y la programación estructurada.

Este paradigma de programación nació con la idea de resolver el problema de la legibilidad de código, que este fuese fácil de seguir, leer y programar.

El principal problema que había hasta la fecha era que el flujo del programa era muy difícil de seguir, ya que se abusaba de instrucciones GOTO en las que se saltaba de un sitio a otro, por lo que había que estar saltando de un lugar a otro del código para entenderlo. Unido a esto, las referencias al código iban por líneas, por lo que cuando saltábamos, teníamos que ver la línea en concreto, no había un identificador o nombre asociado, lo que dificultaba todavía más la lectura.



La base de la programación estructurada es crear programas secuenciales que permitan leerse más claramente, sin saltos hechos con la instrucción GOTO, ampliamente utilizada hasta ese momento.

Además, estableció que cualquier programa con una entrada y una salida podía ser creado como programa estructurado obteniendo igual resultado.

3.1 Ventajas

La principal ventaja de la programación estructurada viene de la idea base, mejorar la legibilidad del código. Esto acarreó otra serie de ventajas:

- La lectura se hace secuencial, sin saltos. Podemos leer todo el código de arriba abajo línea a línea.
- Se hace más fácil probar, mantener y extender, ya que puede ser entendido por terceros de forma más sencilla. Esto ahorra tiempo y dinero.
- La documentación es más sencilla, puesto que el código es mucho más entendible, ahorrando tener que dar explicaciones antes necesarias.

Todo esto hace que su programación sea más sencilla, rápida y cueste menos.

3.2 Inconvenientes

El principal problema es que todo es un único bloque de código, no hay separación de lógica en ficheros. Esto hace que, a mayor sistema, mayor es el fichero y, por tanto, nos obliga a buscar una sección de código entre miles de líneas.

3.3 Estructuras básicas

La programación estructurada defiende el uso de tres estructuras básicas.

3.3.1 Secuencial

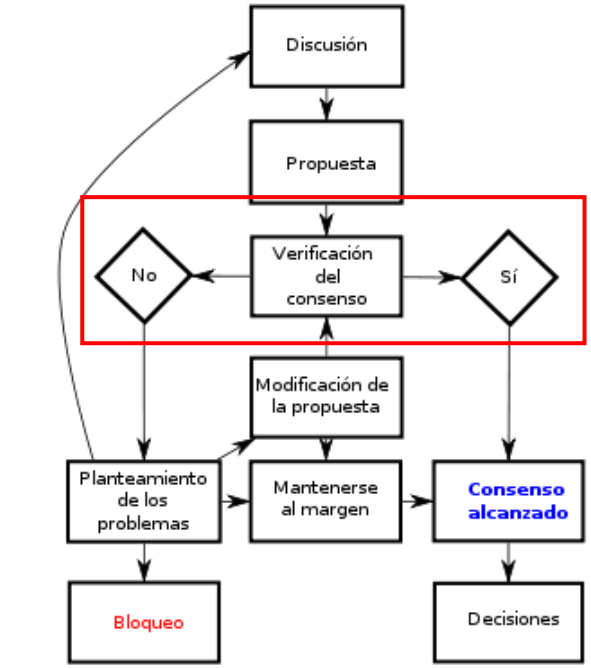
Código que se ejecuta línea a línea, una detrás de la otra sin ningún tipo de alteración.

3.3.2 Alterativa

Modifican la ejecución del flujo del programa mediante condiciones que establecen si un grupo de instrucciones son ejecutadas o no en función del valor de unas variables.

Existen tres tipos:

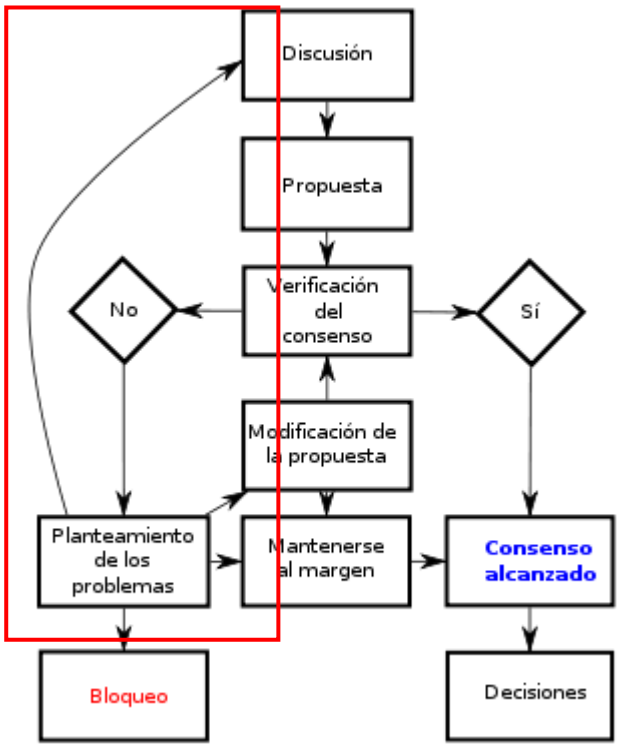
- **Simples:** Si se cumple la condición, se ejecuta un grupo de instrucciones, si no, se ignoran. Por ejemplo, el if.
- **Dobles:** Si se cumple la condición se ejecuta un grupo de instrucciones, si no, se ejecutan otras. Por ejemplo, el if else.
- **Múltiples:** En función del valor de una variable, se ejecuta un grupo de instrucciones, varios o ninguno. Por ejemplo, el switch.



3.3 Iterativa

Modifican la ejecución del flujo mediante la repetición de un mismo conjunto de instrucciones mientras se cumpla una condición. Por ejemplo, el for, foreach, while o do while.

Dicha condición puede ser un número de veces establecido a priori o indeterminado, dependiendo de que se cumpla una condición dentro del conjunto de operaciones del bucle. Por ejemplo, si buscamos una línea determinada de un fichero, el bucle terminará cuando la encontremos, aunque no sabemos a priori cuántas veces se repetirá.



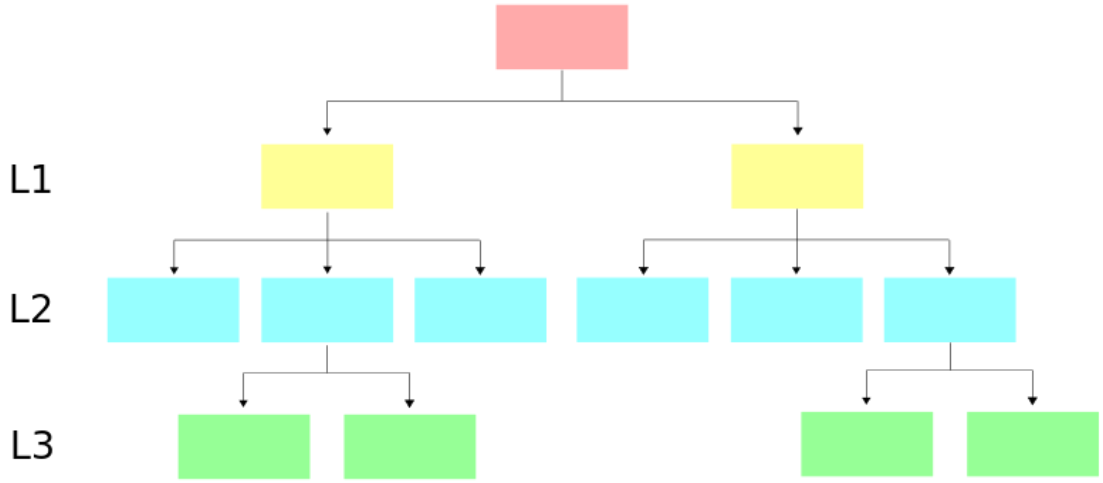
4. Programación modular

La programación estructurada había resuelto el problema de los saltos, la poca legibilidad del código y permitía cierto nivel de reutilización, pero seguía teniendo problemas, pues los sistemas estaban hechos en un archivo y a medida que aumentaba la complejidad del sistema, esto se hacía inviable.

Por tanto, poco tiempo después de la aparición de la programación estructurada, nació la programación modular, una evolución de la anterior que se basa en la idea de separar las distintas partes del sistema en bloques con significado propio.

Se basa en la idea del divide y vencerás, la división de un problema en problemas más pequeños. De esta forma, si resolvemos cada uno de estos subproblemas, se resuelve el problema inicial.

Con esta idea, se crea una estructura jerárquica descendente, (top-down en inglés) en el que cada elemento representa un problema a resolver que se divide, a su vez, en problemas más sencillas hasta llegar a un problema unitario. De esta forma, cada elemento forma una parte del sistema final. Estos elementos se denominan módulos.



Este paradigma también defiende la idea de dividir el sistema en varios archivos, uno por módulo.

Este tipo de lenguaje es mucho más legible y reutilizable, puesto que divide la funcionalidad en diferentes partes facilitando la creación de sistemas más grandes y permite usar partes del sistema en otros sistemas gracias a la utilización de archivos separados.

Además, puesto que está formado por diferentes archivos, la creación de estos puede ser efectuada por distintos miembros de un equipo, lo que aumenta la velocidad a la hora de su creación.

4.1 Módulo

Un módulo es un conjunto de instrucciones agrupadas bajo un nombre identificador que confeccionan las funcionalidades de una parte del sistema. Usualmente se encuentran separados cada uno en un archivo propio.

4.1.1 Características

-Entidad propia: todo módulo tiene un significado por sí mismo, y se conectan con otros módulos para formar un todo, el sistema final.

-Caja negra: admiten una serie de datos, realizan una serie de operaciones transparentes para el sistema y devuelven unos datos en función a esas entradas.

-Tamaño adecuado: el módulo ha de ser lo suficientemente pequeño para mejorar la legibilidad, pero lo suficientemente grande para no aumentar la complejidad del sistema.

4.2 Ventajas

Cada módulo efectúa una función en concreto dentro del sistema, por lo que se necesitamos realizar cualquier cambio en esa parte o analizarla, tan solo necesitamos leer ese código.

Dada su alto nivel de independencia, los módulos suelen localizarse en ficheros independientes, aumentando la legibilidad y su programación, centrándose únicamente en el subproblema en concreto y no en el todo.

Facilita la programación por un grupo de gente, ya que cada programador puede dedicarse a un módulo distinto en un fichero distinto.

Permite la reutilización y el intercambio de los módulos. Si son suficientemente independientes, puede utilizarse un mismo módulo para sistemas distintos o intercambiarse uno por otro.

4.3 Inconvenientes

Podemos caer en error de simplificar in extremis un problema creando un número excesivo de módulos. Estos carecerán de sentido propio, requerirán de otros módulos para tenerlo, creando una red de módulos difíciles de seguir y gestionar.

Por otro lado, podemos caer en el caso opuesto, crear módulos excesivamente grandes que sean difíciles de manejar y que no sean realmente independientes, pues forman parte de varias secciones del sistema.

4.2 Subrutinas

Las funciones, también denominadas subrutinas, son segmentos de código con nombre que reciben, generalmente, unos datos, realizan una serie de operaciones y devuelven, o no, un resultado.

Dado que tienen un nombre identificativo, pueden ser llamadas desde distintas partes del programa, lo que permiten, entre otras cosas, mayor legibilidad y reutilización de código dentro del programa.

4.2.1 Procedimientos

Los procedimientos son la subrutina más simple, reciben una serie de datos sobre los que realizan una serie de operaciones pero no devuelven ningún resultado.

Por ejemplo, una subrutina para escribir una serie de datos en un fichero.

4.2.2 Funciones

Las funciones son subrutinas que reciben datos, realizan una serie de operaciones en base a ellos y devuelven un resultado al programa.

Son el tipo de subrutina más utilizado.

4.2.3 Gestión de memoria

Cuando un programa llama a una subrutina, el estado actual del programa se guarda en memoria en forma de pila. Luego, sobre esta se añaden los parámetros y las variables que vayamos creando.

Cuando las variables de la subrutina se van utilizando, se van quitando de la parte superior de la pila, de tal modo que cuando termine la subrutina, lo que queda es lo que había antes de su llamada.

4.3. Librerías

Las librerías son un conjunto de funciones y procedimientos agrupados en un archivo que sirve como un único módulo. No funcionan como un programa o parte de ningún sistema, ofrecen una serie de utilidades genéricas para ser utilizadas en diversos sistemas.

Es muy habitual que un programador o un equipo creen sus propias librerías (de usuario) que utilizarán en la creación de varios sistemas, ya que ahorra mucho tiempo de programación y se asegura de que el código de dichas librerías funciona de forma óptima.

A su vez, los fabricantes suelen crear librerías (estándar) que faciliten el uso de su hardware y así abstraer a los programadores de la arquitectura del hardware en cuestión.

Existen dos tipos fundamentales, las estáticas, las cuales se incluyen dentro del propio programa y son transparentes para el usuario y las dinámicas, que son externas al programa y el usuario tiene que tener en su equipo.