UD 2

Programación Modular

ÍNDICE

1. Metodología de programación

Características de un buen programa:

- **Correcto**: Hace lo que tiene que hacer.
- **Completo**: Debe controlar todos los casos y prever cualquier reacción del usuario. No deben producirse salidas "descontroladas" del programa.
- **Sencillo**: Fácil de entender y usar para el usuario sin muchos conocimientos de informática. Preciso en la entrada y salida de datos.
- **Claro y legible:**
  - Nombres significativos.
  - Tabulaciones y comentarios aclaratorios.
- **Flexible** (fácil de modificar)**.**
- **Correctamente documentado.**
- **Eficiente:**
  - En tiempo.
  - En espacio (memoria utilizada).
- **Transportable** (fácil cambiar el programa a otra plataforma)**.**

Programación convencional

- Aquella en la que nos ponemos a programar directamente después de hablar con el usuario o de leer un enunciado, sin pensar en ninguna estructuración o algoritmo previo.

- Inconvenientes:
  - No se cuida la declaración de constantes.
  - Se utilizan poco los funciones auxiliares.
  - Se utiliza mucho la sentencia GOTO, y aunque a veces es la forma más sencilla de resolver un problema, dificulta la legibilidad del módulo debido a los continuos saltos en el código.
  - No se incluyen los comentarios necesarios.

Programación estructurada

Hace uso de los siguientes elementos:

- Estructuras básicas de programación repetitivas y selectivas.
    - Este punto viene resumido en el **Teorema de Estructura**, que dice "*Cualquier programa o algoritmo, si es resoluble, se puede resolver siempre utilizando las tres estructuras básicas: secuencial, selectiva e iterativa*".
- Utilización de subprogramas o funciones auxiliares (librerías) para hacer operaciones que se van a repetir.
- Diseño del programa **TOP-DOWN** (descendente), es decir, se analiza el problema subdividiéndolo en problemas más pequeños, a su vez cada subproblema se divide en otros más pequeños, y así sucesivamente hasta llegar a tareas básicas fáciles de resolver.
- Ausencia de saltos incondicionales (GOTO).
- Elaboración de documentación y comentarios que ilustren el código.

Consecuencias de la programación estructurada

- Hace más fácil la verificación de la corrección del programa por el autor o por otras personas. Dicha verificación se puede realizar antes de codificación si se desea.
- Facilita igualmente la depuración del programa para eliminar posibles errores tanto semánticos como sintácticos.
- Permite probar el programa de una manera más rigurosa y completa.

¿Qué es un módulo?

- Un módulo es un conjunto de instrucciones que resuelve una **tarea concreta**.
- Un programa es un conjunto de módulos interrelacionados que resuelven un problema.
- Se caracteriza por:
  - Tiene un nombre que se utiliza para identificarlo e invocarlo.
  - Recibe una serie de datos de entrada y puede devolver datos de salida.
  - Puede ser llamado (invocado) desde cualquier otro punto del programa.
- En algunos lenguajes estructurados (como C) los módulos se implementan como funciones.

2. Programación modular

- **“Divide y vencerás”**
  Dividir el problema original en otros más simples y fáciles de resolver. Posteriormente se integrarán cada una de las partes (módulos) para resolver el problema original.

- **Diseño de programa descendente (TOP-DOWN)**
  Se descompone el programa en módulos y en cada nivel los módulos deben ser de menor complejidad.

- **Abstracción funcional:**
  Reconocer los diferentes “subproblemas” dentro del problema original, sin preocuparse por el momento de cómo se van a resolver.

- **Organización de componentes interrelacionados**
  Cada componente pueda ser pensado, verificado y probado independientemente. En consecuencia, es viable utilizar componentes o módulos preconstruidos, reutilizar módulos ya diseñados y construidos por el mismo programados o por otros.

Dividir en módulos ¿hasta cuando?

- No hay una regla fija.

- Dependerá de:
  - Tamaño (líneas de código, aprox. máximo 40).
  - Complejidad lógica (nº de sentencias de control).
  - Número de parámetros (debe ser mínimo).

- Se deberá:
  - Maximizar la cohesión de cada módulo.
  - Minimizar el acoplamiento entre módulos.

Cohesión y Acoplamiento

- Cohesión: evalúa el grado de interrelación entre los elementos que constituyen un módulo.
- Acoplamiento: evalúa el grado de interconexión entre los módulos.

- Un buen diseño debe **minimizar el acoplamiento**, aumentando la independencia de cada módulo respecto a los demás, y **maximizar la cohesión** en cada módulo, haciendo que todos sus elementos estén fuertemente relacionados entre sí.
  - Estas características se dan cuando se utilizan funciones que devuelven un solo valor al punto de programa donde se les llamó, ya que esto también implica que la función ha resuelto un solo problema básico, con lo que todos sus elementos están muy relacionados entre sí.
  - Un caso muy común que aumenta el acoplamiento y por lo tanto hace que los módulos dependan unos de otros es el uso de variables globales, ya que las modificaciones sobre una de estas variables en un módulo puede afectar al buen funcionamiento de otro módulo, por lo que se debe minimizar en lo posible el uso de este tipo de variables.

Fases a la hora de resolver un problema

1. Establecer la descomposición modular general del problema.
2. Especificar las interfaces o transferencias de datos entre los módulos de manera que sean mínimas.
3. Especificar lo que hace cada módulo mediante pseudocódigo (deseable) utilizando solo las estructuras básicas.
4. Si el grado de complejidad de un módulo es demasiado alto, fraccionarlo o descomponerlo expandiendo el diagrama de estructura.

3. Funciones

- Es un módulo que toma una serie de entradas (parámetros formales) realiza una tarea concreta y produce una salida llamada valor de retorno.
- Puede que no tenga parámetros de entrada ni valor de retorno.

```
tipo nombre_funcion (lista_param)
{
   /* variables locales a la función/
   /* instrucciones de la función*/
}
```

La lista de parámetros formales es de la forma

tipo nombre_param1, tipo nombre_param2, …

Más sobre funciones

`return valor;`

- En las funciones que devuelvan algo siempre debe aparecer (una sola vez).
- Provoca una salida inmediata de la función.

Llamada a una función:

- Una función puede llamarse desde cualquier punto del programa o desde otra función.

  valor_devuelto = nombre_funcion (lista_param_reales);

- La lista de parámetros reales es de la siguiente forma:

  nombre_param1, nombre_param2, …

4. Parámetros. Tipos

- Los parámetros son los variables de enlace entre los distintos módulos.
- Se clasifican en:
  - Parámetros reales: variables locales pertenecientes al módulo que realiza la llamada y cuyo valor es enviado al módulo que se llama (módulo invocado).
  - Parámetros formales: Son variables locales pertenecientes al módulo.

Paso de parámetros

2 tipos:

1. Por valor:

   Cuando se llama a una función se reserva memoria para los parámetros formales de la función y se realiza una copia de los parámetros reales en los formales.

2. Por referencia:

   A la función se le pasa una referencia a los parámetros reales de forma que los cambios que hagan en ellos se mantendrán al terminar la función.

6. Sentencia `return`

```
return valor;
```

- El control vuelve inmediatamente al lugar donde se invocó la función.
- En todas las funciones que devuelvan un valor debe aparecer la sentencia `return` (si no el valor devuelto queda indeterminado).
- Mal uso de `return;` para provocar la salida inmediata de la función.

Observaciones sobre `return`

- La sentencia `return` tiene dos usos importantes. Primero, fuerza a una salida inmediata de la función, esto es, no espera a que se llegue a la última sentencia de la función para acabar. Segundo, se puede utilizar para devolver un valor.

- `return` no es una función sino una palabra clave del C, por lo tanto no necesita paréntesis como las funciones, aunque también es correcto:

```
return (expresión);
```

  pero teniendo en cuenta que los paréntesis forman parte de la expresión, no representan una llamada a una función.