Estructura de Datos y Algoritmos

# Diseño de Programas, Módulos de Programa

## 1. Diseño de Programas.

En una primera etapa analizamos el problema. En esta etapa determinamos *que* hace el programa. Nos preguntamos: que entradas se nos ofrecen, que salida debemos generar, que método debemos usar para llegar a la solución deseada.

Luego diseñamos el algoritmo. Si en el análisis determinamos *qué* hace el programa, en esta etapa determinamos *cómo* lo hace. Para ello se divide el problema en varios subproblemas que se solucionan de manera independiente, lo que se denomina **diseño modular**.

## 2. Modularidad

La modularidad es la capacidad que tiene un sistema de ser estudiado, visto o entendido como la unión de varias partes que interactúan entre sí y que trabajan para alcanzar un objetivo común, realizando cada una de estas partes, una tarea necesaria e independiente para conseguir dicho objetivo.

Cada una de las partes en que se encuentra dividido el sistema recibe el nombre de módulo.

Idealmente un módulo debe de ser independiente del resto de los módulos y puede comunicarse con algunos de estos a través de entradas y/o salidas bien definidas.

Cada uno de los módulos de un programa debe cumplir las siguientes características:

- tamaño pequeño : facilita aislar el impacto que puede tener la realización de un cambio en el programa ya sea para corregir un error o por rediseño del algoritmo correspondiente.
- independencia modular : cuanto más independientes son los módulos entre si más facilmente se trabajará con ellos. Esto implica que para desarrollar un módulo no es necesario conocer detalles internos de otros módulos. Le llamamos también *encapsulamiento*

Se persigue crear programas modulares (que cumplen o tienen la característica de **modularidad**. Con esto se consigue desarrollar programas a partir de un conjunto de módulos cada uno de los cuales desempeña una tarea necesaria para el correcto funcionamiento del programa global. En el caso de que el conjunto de módulos implicado esté sometido a una jerarquia estariamos hablando de un sistema por capas, en la que cada capa representaría un nivel en la jerarquia de los módulos.

# 3. Implementación de módulos

La experiencia ha demostrado que la mejor forma de desarrollar y mantener un programa grande es construirlo a partir de partes o componentes más pequeños los cuales son más faciles de manejar que el programa en forma completa. Esta técnica se llama **divide y conquista**. Veremos características claves de *C* que facilitarán el diseñ0, implementación, operación y mantenimiento de programas grandes.

Los módulos en *C* se llaman *funciones*. Los programas en *C* se escriben combinando funciones nuevas que el programador escribe con las funciones "preempacadas"disponibles en la biblioteca estándar de *C*.

La biblioteca estándar de *C* contiene un amplio conjunto de funciones para realizar cálculos matemáticos comunes, manipulación de cadenas, de caracteres, entrada/salida, comprobación de errores y muchas otras operaciones útiles. Esto simplifica el trabajo del programador.

El programador puede escribir funciones para definir ciertas tareas específicas que podrán utilizarse en muchos puntos del programa. Éstas a veces se conocen como *funciones definidas por el programador*. Las instrucciones reales que definen la función sólo se escriben una vez y se ocultan de las demás funciones.

Se puede invocar una función mediante una *llamada de función*. Esta especifica el nombre de la función y le proporciona información (en forma de argumentos) que necesita para trabajar.

## 3.1. Funciones

Las funciones le permiten al programador modularizar sus programas. Todas las variables declaradas en las definiciones de función son variables locales (sólo se conocen en la función en la que se definieron).

La mayoría de las funciones tienen una lista de parámetros, que proporcionan el medio para la comunicación de información entre las funciones. Los parámetros de una función también son variables locales.

Hay varias razones para "funcionalizar"los programas. La estrategia de divide y conquista hace más manejable el desarrollo de programas. Otra razón es la *reutilización del software*, es decir la utilización de funciones ya existentes como bloques de construcción de nuevos programas. La reutilización de software es un factor medular en la programación orientada a objetos (C++). Un tercer aliciente es evitar la repetición del código dentro de un programa. Cuando el código se empaca como función es posible ejecutarlo desde varios puntos del programa, simplemente llamando a la función.

**Observación 1:** Cada función debe limitarse a efectuar una sola tarea bien definida y el nombre de la función debe expresar claramente la tarea. Esto promueve la reutilización del software.

**Observación 2:** Si no puede encontrar un nombre consiso que exprese lo que hace la función es posible que ésta haga demasiadas tareas. Generalmente es mejor dividir la función en varias partes más pequeñas.

### 3.1.1. Definiciones de funciones

Los programas en *C* consisten de una función *main* que llama a funciones de la biblioteca estándar y funciones definidas por el usuario.

El usuario debe declarar *prototipos de función* para las funciones que define, por ejemplo:

```
int cuadrado (int y);
```

es el prototipo correspondiente a la función cuadrado (observar finaliza con ";"). No es necesario el prototipo de función si la función se define antes de ser utilizada.

El formato de una definición de función es:

```
tipo-de-valor-devuelto nombre-de-funcion (lista-de-parametros)
{
        declaraciones e instrucciones
}
```

El tipo-de-valor-devuelto es el tipo de datos del resultado que la función devuelve al invocador. En caso de que la función no devuelva ningún valor ponemos como dicho tipo "void".

### 3.1.2. Estructuras y funciones

Las unicas operaciones legales sobre una estructura son copiarla o asignarla como unidad, tomar su dirección con & y tener acceso a sus miembros. La copia y la asignación incluyen pasarlas como argumentos a funciones y también regresar valores de funciones.

Las estructuras no se pueden comparar. Una estructura se puede inicializar con una lista de valores constantes de miembro; una estructura automática también se puede inicializar con una asignación.

Ejemplos son:

```
struct punto {
        int x;
        int y;
};
```

podemos declarar e inicializar en la forma:

```
struct punto maxpt = {320,200};
```

Accedemos a los miembros de una estructura con el operador punto ".". Este operador conecta el nombre de la estructura con el nombre del miembro. Por ejemplo podemos imprimir las coordenadas del punto con la instrucción:

```
printf(" %d, %d", pt.x, pt.y);
```

Definamos ahora un rectangulo dando un par de puntos que denotan las esquinas diagonalmente opuestas:

```
struct rect {
        struct punto pt1;
        struct punto pt2;
};
```

En lo que respecta a funciones que reciban y/o retornen estructuras consideremos las siguientes:

```
/* construir_punto : crea un punto con las componentes x e y */
struct punto construir_punto(int x, int y)
{
        struct punto temp;

        temp.x = x;
        temp.y = y;
        return temp;
}


/* sumo_puntos : suma dos puntos */
struct punto sumo_puntos(struct punto p1, struct punto p2)
{
        p1.x += p2.x;
        p1.y += p2.y;
        return p1;
}


/* pt_en_rect : regresa 1 si el punto p esta en el rectangulo r y 0 si no esta*/
/* suponemos las coordenadas de p1 son menores que las de p2. */
int pt_en_rect(struct punto p, struct rect r) {
    return p.x >= r.pt1.x && p.x <= r.pt2.x
            &&
            p.y >= r.pt1.y && p.y <= r.pt2.y;
}
```

Si una estructura grande va a ser pasada a una función es más eficiente pasar un apuntador que copiar la estructura completa. Los apuntadores a estructuras son como los apuntadores a variables ordinarias. La declaración:

```
struct punto *pp;
```

dice que pp es un apuntador a una estructura de tipo struct punto. Hacemos referencia a las partes del punto escribiendo (*pp).x e (*pp).y. Otra forma de referenciar a las partes es con : pp− >x y pp− >y.