Universidad Autónoma del Estado de México

Facultad de Ingeniería

Programación Modular

Tania Lilia Chávez Soto

Octubre 2017

Contenido

Ubicación del material

Unidad de Aprendizaje:

- **Programación Básica**

Clave:

- **L41205, L41304**

Horas teóricas:

- **2 horas**

Horas prácticas:

- **2 horas**

Créditos:

- **8 créditos**

Tipo de curso:

- **Curso presencial**

Núcleo de formación:

- **Sustantivo**

Unidad de aprendizaje antecedente:

- **Ninguna**

Unidad de aprendizaje consecuente:

- **Ingeniería Civil: Ninguna**
- **Ingeniería Mecánica : Programación Avanzada**

Programas educativos o espacios académicos en los que se imparte:

- **Licenciatura en Ingeniería Civil, Licenciatura en Ingeniería Mecánica**

Programación modular

Introducción

La programación modular está presente desde los primeros tiempos de la programación como disciplina. De hecho existía programación modular antes de existiera programación estructurada para poder abordar problemas complejos.

Cuando se requieren resolver problemas más complejos, en los que el número de instrucciones aumenta, existen técnicas que permiten resolverla de forma eficiente estos problemas, facilitando al programador la construcción de la solución.

Se puede definir a la programación modular como aquella que usa el concepto de dividir un problema complejo en subproblemas más pequeños, hasta que estos sean fáciles de tratar y resolver por separado. Así la solución de los subproblemas en conjunto dan como resultado la solución del problema completo (López Román, 2003).

Aplicando este principio a la hora de hacer un programa, entonces habría que dividir el programa en "subprogramas" que realicen tareas específicas. Es importante considerar que para poner en práctica la modularización, es necesario un mecanismo que permita aplicarla en los lenguajes de programación; este mecanismo existe prácticamente en todos los lenguajes y consiste en la posibilidad de definir tareas específicas como módulos de código independientes del programa principal. En consecuencia estos módulos de código independientes del programa deben poder invocarse desde el programa o módulo principal para que empiecen a trabajar y deben acabar devolviendo el control al programa o módulo principal cuando terminen de ejecutarse.

La programación modular permite:

a) Dividir la complejidad de un problema convirtiendo problemas complejos en un conjunto de problemas más simples y por tanto más sencillos de implementar.
b) Reutilizar el código de un programa en cualquier momento de la ejecución del mismo.

Evidentemente la división de un problema en módulos no tiene por qué ser ni única ni obligatoria, pero si es claramente aconsejable a la hora de abordar un problema para lo cuál que se aplican diversos conceptos y técnicas entre las que se encuentran el diseño descendente, el proceso de modularización y el paso de parámetros, por mencionar algunos. A continuación se abordaran cada uno de ellos.

1. Diseño descendente.

Es una técnica que permite diseñar la solución de un problema con base en la modularización o segmentación dándole un enfoque de arriba hacia abajo (Top Down Design). Esta solución permite dividir el problema en módulos que se estructuran e integran jerárquicamente (ver figura 1), como si fuera el organigrama de una empresa (López Román, 2003).

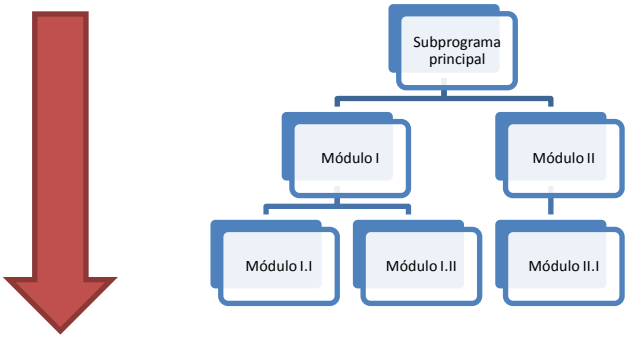


Figura 1. Concepción de módulos.

1.1 Módulo.

Es un segmento, rutina, subrutina, subalgoritmo o procedimiento que puede ser definido dentro de un algoritmo mayor con el propósito de ejecutar una tarea específica, pudiendo ser llamado o invocado desde el módulo principal cuando se requiera.

Este enfoque de segmentación o modularización es útil:

- ✓ Cuando existe un grupo de instrucciones o una tarea específica que debe ejecutarse en más de una ocasión.
- ✓ Cuando un problema es complejo o extenso, entonces la solución se "divide" o "segmenta" en módulos que ejecutan "partes" o tareas específicas.

Dicha solución se organiza y divide en partes más pequeñas que sean fácilmente manejables y que, lógicamente, puedan ser separadas; así cada una de estas partes se dedica a ejecutar una determinada tarea, lo que redundará en una mayor concentración, entendimiento y capacidad de solución a la hora de diseñar la lógica de cada una de estas.

Las partes en las que se divide una empresa (funcionalmente separadas) son el equivalente a los módulos o segmentos del algoritmo, algunos de ellos son módulos directivos o de control, que son los que se encargarán de distribuir el

trabajo a los demás módulos, de tal forma que se puede diseñar una especie de organigrama que indique la estructura general de un algoritmo (López Román, 2003).

1.2 Diseño descendente con pseudocódigo.

Cuando el enfoque de diseño descendente se utiliza de manera conjunta con la técnica del pseudocódigo, se logra una herramienta de diseño de algoritmos (programas) muy poderosa. A continuación se explica el procedimiento para hacerlo:

1. Se tiene un problema ya analizado.
2. Se diseña la estructura general del algoritmo utilizando el diseño descendente, es decir, se definen los módulos que integrarán el algoritmo.
3. Se toma cada módulo y se diseña su lógica utilizando pseudocódigo.

La llamada de un módulo en pseudocódigo se hace siguiendo la siguiente sintaxis:

Llamar NombreModulo

Donde:

Llamar: identifica la acción de llamar o invocar a un módulo

NombreModulo: es el nombre del módulo que se está llamando

Formato general de un algoritmo con módulos:

Parte 1. Se tiene el encabezado del algoritmo. Es obligatorio.

Parte 2. Se tiene una parte de declaraciones globales, donde se pueden declarar

- ✓ Constantes
- ✓ Tipos
- ✓ Variables

Que son globales, es decir, se pueden utilizar en cualquier módulo del algoritmo. Es opcional; es decir, puede estar presente o no.

Parte 3. Se tiene el módulo principal, es el paso 1 del algoritmo. Es obligatorio.

Parte 4. Se tiene uno o más módulos o funciones subordinados, son opcionales (ver figura 2).

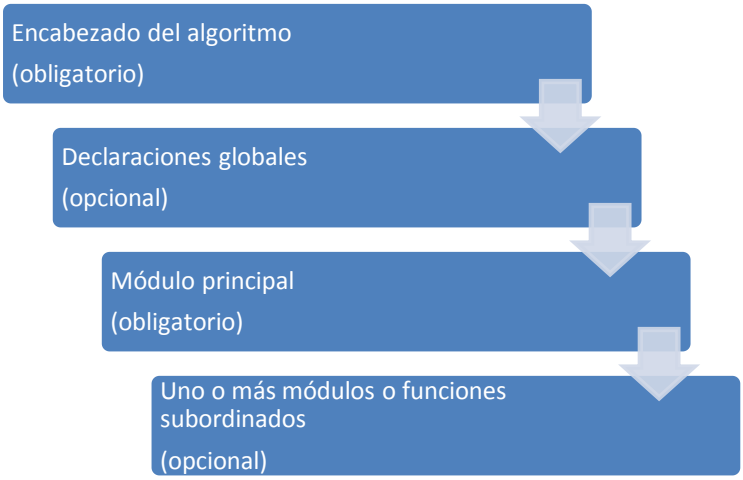


Figura 2. Algoritmo con módulos.

2. Proceso de modularización o segmentación.

El proceso de segmentación consiste en hacer una abstracción del problema, del cual se tiene inicialmente el panorama general. Enseguida se procede a "descomponer" o "dividir" el problema en partes pequeñas y simples (ver figura 3).

I. Se forma un primer módulo enunciando el problema en términos de la solución de éste.

II. Se toma este módulo y se busca la forma de dividirlo en otros módulos más pequeños, que ejecuten tareas o funciones específicas. Normalmente serán las mismas funciones que se desea que ejecute el algoritmo, lo que permite de una forma simple definir los módulos, y de esta forma dividir el problema en partes más manejables.

III. Se repite el paso anterior para cada módulo nuevo definido, hasta llegar a un nivel de detalle adecuado, es decir, hasta hacer que cada módulo ejecute una tarea específica, claramente definida y que el diseño y la codificación de la lógica del mismo resulte fácil (Ver anexo I).

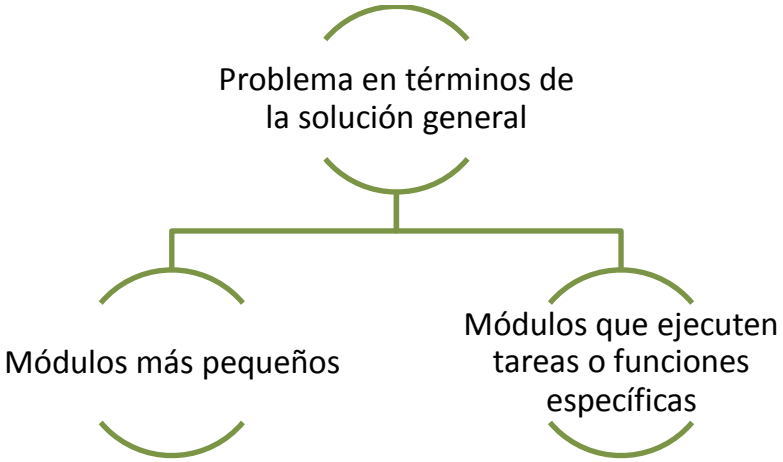


Figura 3. Descomposición de módulos.

2.1 Modularización funcional.

Los bloques de código independientes del programa principal que se han definido como módulos tienen siempre una connotación funcional. Esto quiere decir que estos segmentos realizan alguna tarea específica por lo que cuando se definen estos subprogramas, lo que se hace es identificar actividades concretas y desarrollar el subprograma para resolverlas.

Una ventaja al dar solución a los problemas mediante la modularización es que se pueden "encapsular" las distintas tareas que se quiere realice el programa, logrando así colocar como una unidad todas las instrucciones relevantes para una sección específica del programa, lo que también permite que cuando existe la necesidad de realizar algún cambio en dichas instrucciones se tenga claridad a donde dirigirse para hacer ese cambio y no haya necesidad de analizar de manera completa todo el desarrollo, lo que hace que los programas sean fáciles de probar y mantener.

Lo anterior permite que si se logra tener un programa que resuelva una tarea específica y este proceso está suficientemente probado, entonces una vez que está programado, es posible usarlo tantas veces como sea necesario con la única condición de conocer los datos que lo alimentan y los resultados que arroja. Por otro lado, esta condición evita que se repita el código muchas veces y que si se comete un error de programación en este segmento del código solo se tenga que corregir una vez.

Estos bloques de código o subprogramas reciben distintas denominaciones en los diferentes lenguajes, por ejemplo en los no orientados a objetos la división tradicional distingue entre:

- Subprogramas, subrutinas o procedimientos, si pueden devolver más de un valor.
- Funciones, si devuelven un único valor a través de su identificador.

La estructura del programa en función de sus módulos se puede representar con los denominados diagramas de estructura, que son herramientas típicas del diseño estructurado y forman parte de una completa metodología de trabajo. Estos diagramas son una representación en forma de árbol jerárquico de la estructura del software, donde los diferentes módulos se representan como cajas rectangulares. La relación de invocaciones de unos módulos a otros se representan en estos diagramas por medio de flechas y el sentido de las flechas representa el orden de invocación de los módulos.

3. Cohesión y acoplamiento.

Los módulos deben cumplir dos características básicas para ser candidatos a una buena división del problema.

- Alta cohesión: las instrucciones contenidas dentro de un módulo deben contribuir a la ejecución de la misma tarea.
- Bajo acoplamiento: a dependencia entre módulos debe ser mínima. Lo que implica que no debe haber dantos compartidos entre los módulos (ver figura 4).

Por lo general se espera que cada módulo de código realice una sola tarea; es decir, todos los componentes de un módulo (sentencias) deben ir dirigidos a resolver un y sólo un problema. A esta propiedad se le denomina cohesión y se dice que los módulos deben ser cohesivos.

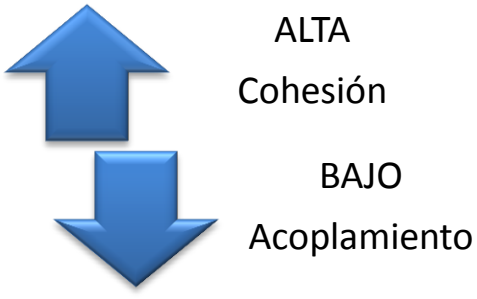


Figura 4. Características de programas modulares.

El objetivo en el diseño modular es conseguir módulos tan cohesivos como sea posible. Muchas veces esto no es posible al cien por ciento pero siempre debe pretenderse desarrollar módulos con alta cohesión.

La cohesión tiene que ver con que cada módulo del sistema se refiera a un único proceso o entidad. A mayor cohesión el módulo en cuestión será más sencillo de diseñar, programar, probar y mantener.

En el diseño estructurado, se logra alta cohesión cuando cada módulo (función o procedimiento) realiza una única tarea trabajando sobre una sola estructura de datos. Una prueba que se recomienda hacer a los módulos funcionales para ver si son cohesivos es analizarlos y describirlos con una oración simple que contenga un solo verbo activo. Si existe más de un verbo presente en la descripción del procedimiento o función, se debía plantear más de un módulo, y volver a hacer la prueba con los módulos resultantes (ver figura 5).

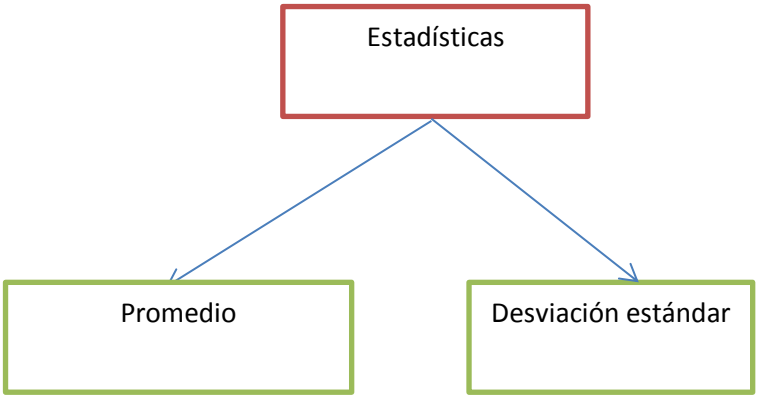


Figura 5. Problema dividido en módulos.

Otro aspecto que destaca al usar modularidad es el acoplamiento que mide el grado de relación de un módulo con los demás. Con un menor acoplamiento el módulo es más sencillo de diseñar, programar, probar y mantener.

En el diseño estructurado, se logra bajo acoplamiento reduciendo las interacciones entre procedimientos o funciones, reduciendo la cantidad y complejidad de los parámetros y disminuyendo al mínimo los parámetros por referencia.

4. Alcance de las variables.

Cuando se consideran varios módulos de software y no un solo programa principal, se pueden declarar variables tanto en el contexto global del algoritmo, como de manera local en cada módulo a lo que se le conoce como alcance de las variables, en otras palabras esto permite identificar la zona del programa donde una variable es accesible “conocida”. Desde este punto de vista, las variables pueden ser de dos tipos: locales o globales.

4.1 Variables globales.

Son las que son accesibles desde cualquier punto del programa y se pueden usar desde cualquier módulo o subprograma, esto lleva a considerar que la variable puede usarse en cualquier parte del programa y su valor se puede alterar incontroladamente, y si posteriormente es necesario usar la variable en otra parte del programa con su valor original, se tendrá un error. El punto donde se da el error es fácil de localizar, pero no lo es tanto el origen del mismo. Este tipo de efectos colaterales produce errores cuyo origen es difícil de trazar y localizar: La programación modular sugiere que se evite el uso de variables globales (ver figura 6).

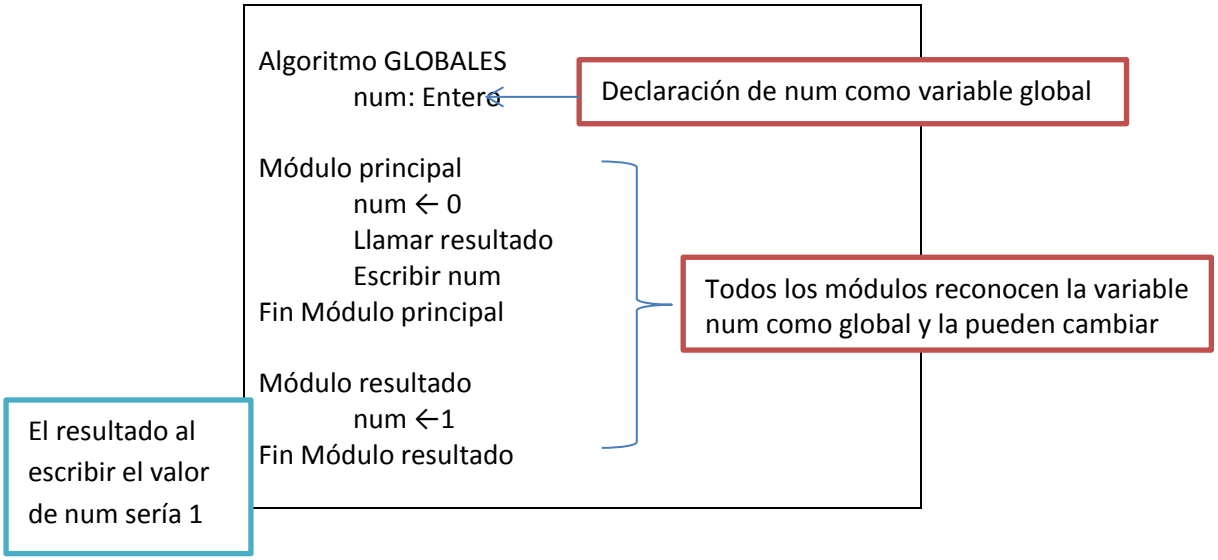


Figura 6. Variables globales.

4.2 Variables locales.

Las variables locales sólo existen en un ámbito determinado del programa, por ejemplo en un subprograma o en un bloque de sentencias. Solo pueden ser utilizadas en el módulo donde fueron definidas (ver figura 7).

Entonces si son analizados los conceptos de variables globales y locales, es posible resumir que cuando se usan variables locales, éstas son independientes de las globales y de las de otros módulos.

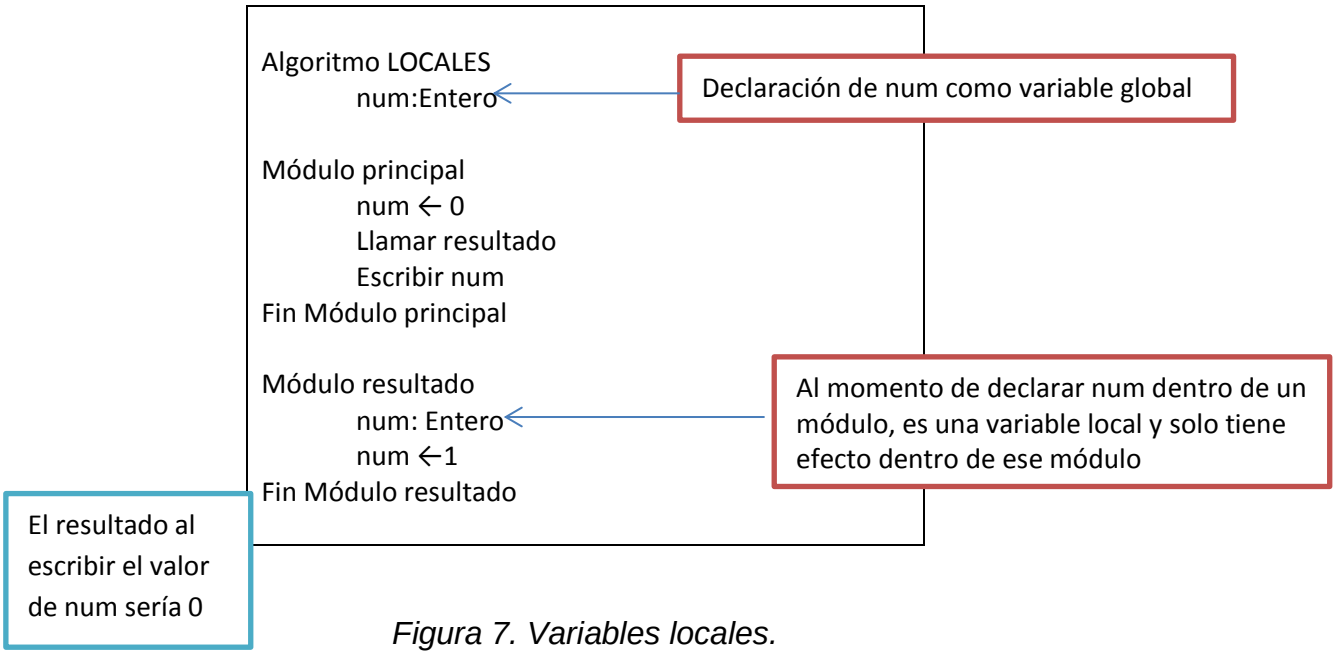


Figura 7. Variables locales.

4.3 Parámetros.

Entre los módulos que componen la estructura completa de un problema, se establece una serie de comunicaciones a través de determinadas interfaces (llamadas a los módulos) que permiten el paso de información de un módulo a otro así como la trasferencia del control de la ejecución del programa.

Un método puede aceptar información para usarla en su cuerpo de sentencias. Esta información es suministrada en forma de literales, variables pasadas al método a través de su cabecera y cada uno de estos elementos de información se denomina parámetro.

De manera más formal se puede definir un parámetro como un valor que se pasa al método cuando éste es invocado. La lista de parámetros en la cabecera (al inicio) de un método especifica los tipos de los valores que se pasan y los nombres por los cuales el método se referirá a los parámetros en la definición del método.

- ✓ En la definición del método los nombres de los parámetros aceptados se denominan *parámetros formales*.
- ✓ En las invocaciones al método desde algún punto del programa, los valores que se pasan al método se denominan *parámetros actuales (reales)*.

Los parámetros actuales y los parámetros formales no necesitan tener los mismos nombres, ya que se corresponden por tipo y posición.

Cuando se invoca un método, los parámetros se colocan entre paréntesis, si no se necesitan parámetros se usan paréntesis vacíos.

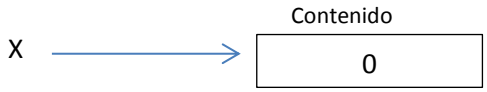
En relación con los parámetros, un aspecto importante, que es necesario conocer en cualquier lenguaje, es como se realiza el paso de los parámetros actuales a los formales, a lo que se le conoce comúnmente como el problema del paso de parámetros. Hay dos posibles formas por valor o por referencia.

a) *Paso por valor:*

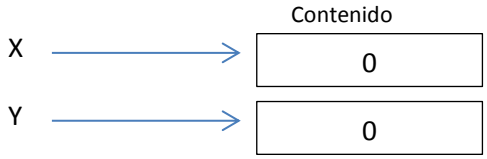
En este caso el método recibe una copia del valor que se le pasa. La variable original (parámetro actual) no cambia de valor, independientemente de que en el método se cambie el contenido del parámetro formal.

Explicación.

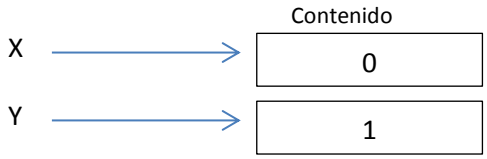
Se define la variable X en módulo Principal, la cual tiene su contenido en la memoria, se le coloca 0



Cuando se llama al Módulo cambiar, se conectan X y Y vía parámetro por valor, lo que hace que el valor de X se le asigne a Y; de ahí en adelante X y Y no tienen ninguna relación.



Es decir X tiene su propio contenido en la memoria y Y tiene el suyo. Cuando a Y se le coloca 1 ¿qué le pasa a X)?



A X no le pasa nada; es decir, a Y se le coloca 1 en su contenido, y X ni cuenta se da, porque tiene su propio contenido independientemente de Y (López Román, 2003).

b) *Paso por referencia:*

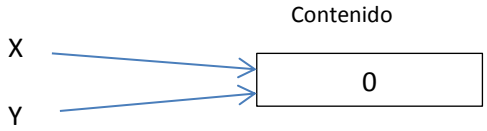
En el paso por referencia no se pasa una copia del valor sino la identificación de la zona de memoria donde se almacena dicho valor. Por esta razón al trabajar dentro

del módulo con la entidad pasada por referencia se está manipulando el mismo valor que se utiliza fuera. Para efectos prácticos si se realiza una modificación de ese valor dentro del módulo, la modificación se mantiene al salir del mismo y la diferencia con el paso por valor, es que el que el módulo maneja una copia del valor original, y las modificaciones realizadas dentro del método no afectan a la variable externa.

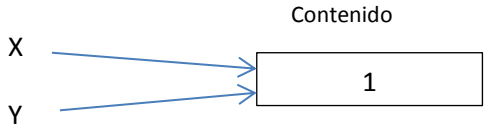
Explicación:

Se define a la variable X en el módulo principal, la cual tiene su contenido en la memoria, y se le coloca 0.

Cuando se llama al Módulo Cambiar, se conecta X y Y vía parámetro por referencia, lo que hace que sean sinónimos, es decir, que Y utilice la misma dirección de memoria que X.



Cuando Y se le colca 1 ¿qué le pasa a X?



A X le pasa lo mismo que le sucede a Y; es decir, a Y se le coloca 1 e indirectamente le sucede lo mismo a X, porque manejan el mismo contenido en la memoria (López Román, 2003).

5. Funciones definidas por el usuario.

Cuando existe un cálculo que será usado de manera repetida a lo largo del programa en distintos momentos se implementan funciones que el usuario define. Una función es muy parecida a los módulos, con la diferencia de que sólo devuelve un valor de un tipo de dato simple.

El nombre de la función puede estar seguido por uno o más parámetros encerrados entre paréntesis, a continuación se detalla como son definidas:

Sintaxis:

Función Nom_Función (Parámetros): Tipo de dato

Declaraciones

Acción 1

…..

Acción N

Fin Función Nom_Función

Donde:

Función: Indica que se está definiendo una función.

Declaraciones: se pueden definir constantes y variables.

Nom_Función: Es el nombre de la función que será manejada como cualquier variable de tipo simple.

Parámetros: Es la lista de variables de entrada que sirven como base para hacer los cálculos en la función.

Tipo de dato: Es el tipo de dato que regresa la función.

Para llamar una función se indica el nombre de la función y entre paréntesis el parámetro o parámetros que se envían (ver Anexo II).

Nom_Función (lista de parámetro(s))

6. Módulos de programa en C.

En C la técnica de programación modular se implementa mediante la utilización de funciones. Una función permite agrupar un conjunto de instrucciones en un bloque que típicamente realizara una tarea elemental. Por lo general en C, los programas se escriben combinando nuevas funciones que el programador diseña, con funciones “preempacadas”, disponibles en la biblioteca estándar de C (Deitel & Deitel, 2004).

Como se indicó anteriormente para dar solución a un problema particular es necesario identificar previamente las tareas elementales que debe realizar el problema, y posteriormente estas se implementarán como funciones.

Las funciones permiten modularizar un programa y se invocan mediante una llamada de función que específica el nombre de la misma y proporciona

información (en forma de argumentos) que la función llamada necesita a fin de llevar a cabo la tarea que tiene designada.

6.1 Función principal.

Todo programa en C contiene una función principal o función *main*, la cual es la encargada de llevar el control de ejecución del programa (secuencia de ejecución de instrucciones) y de llamar a ejecución a las funciones correspondientes en los puntos del programa donde se necesiten.

Un programa siempre empieza a ejecutarse desde la función *main*, denominándose a este el módulo o función principal del programa (raíz de la estructura jerárquica). Para que una función (excepto la función *main*) se ejecute, debe ser siempre llamada por otra función. A una función que llama a otra se le denomina función invocadora (o también llamada llamadora) y a la función llamada se le denomina función invocada.

Cuando una función invocadora realiza una llamada a otra función, el control se transfiere de la primera a la segunda, lo que quiere decir que, la función que pasa ahora a ejecutarse es la segunda. Adicionalmente la función invocadora puede trasferir ciertos datos a la función invocada en el momento de realizar la llamada a estos datos se les denomina argumentos de entrada de la función.

Cuando la función invocada concluye su ejecución debe devolver el controla a la función que la ha llamado y adicionalmente la función invocada puede devolver un resultado o dato a la invocadora, a este dato se le denomina salida de la función.

6.2 Funciones.

Como ya se ha mencionado anteriormente existen varias ventajas al modularizar e implementar funciones:

- ✓ Los programas son más sencillos de implementar y corregir, puesto que son tareas básicas compuestas de unas pocas líneas de instrucciones.
- ✓ Fomenta la reutilización.
- ✓ Es posible crear programas partiendo de funciones estandarizadas que ejecuten tareas específicas, en vez de desarrollarlos usando código personalizado.
- ✓ Evitar la repetición de código, ya que agrupar código en forma de funciones permite que se ejecute dicho código desde distintas posiciones en un programa, simplemente llamando dicha función.
- ✓ Se puede llamar a una función desde diferentes partes de un programa.

- ✓ Se pueden trasferir un conjunto de datos distinto cada vez que se llama a la función (entorno de ejecución de una función), obteniendo resultados diferentes en cada uno de los casos.

La estructura de una función en C se divide en:

- ✓ **Cabecera o prototipo de la función:** Se define el nombre de la función y el modo en el que se va a realizar la trasferencia de información con dicha función (argumentos de entrada y tipo de dato de salida).
- ✓ **Cuerpo de la función**: Se encuentra encerrado entre llaves y esta formado por dos partes, la primera contiene la declaración de las variables que se van a emplear en la función y la segunda un conjunto de instrucciones que realizan la tarea para la cual se diseño la función, empleando los datos en los argumentos formales de la cabecera. La declaración de variables (si existe) en el cuerpo de la función debe proceder siempre a las instrucciones del mismo. El control de la ejecución de programa se pasará automáticamente a la función invocadora una vez que se haya terminado de ejecutar las instrucciones que forman el cuerpo de la función invocada.

Finalmente las funciones se utilizan normalmente en un programa, escribiendo el nombre de la función seguido por un paréntesis izquierdo y a continuación por el argumento (o una lista de argumentos separados por comas) de la función, seguida por un paréntesis derecho (ver figura 8).

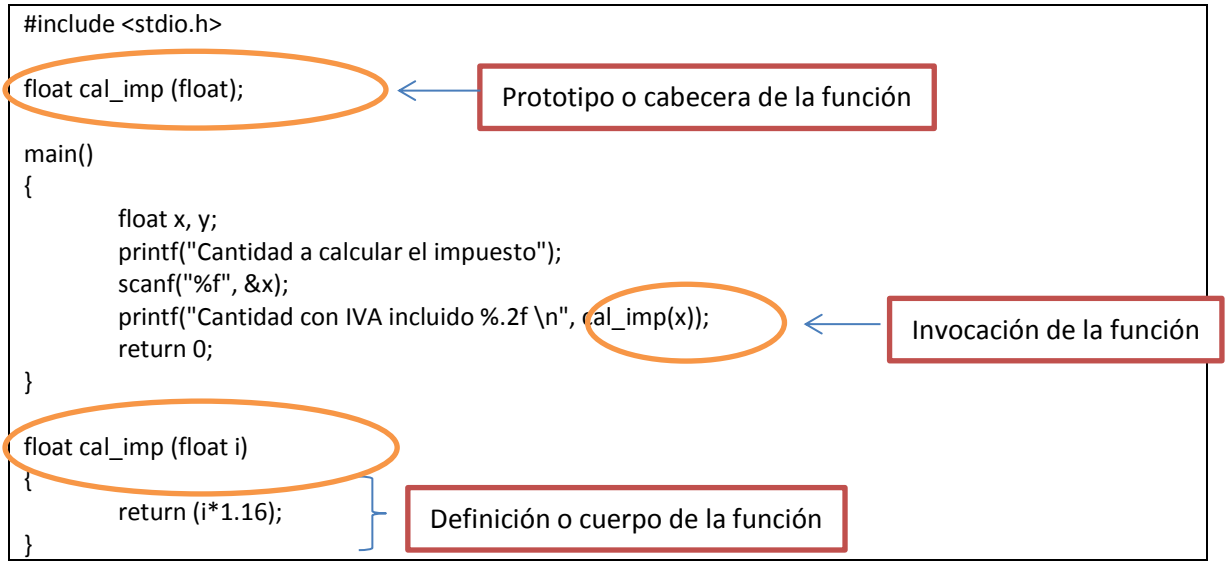


Figura 8. Funciones en un programa en C

Prototipo de la función.

Un prototipo de función le indica al compilador el tipo de dato que regresa la función, el número de parámetros que la función espera recibir, los tipos de dichos parámetros, y el orden en el cual se esperan dichos parámetros. El compilador utiliza los prototipos de funciones para verificar las llamadas de función.

Cuerpo de la función.

El formato de una definición de función como se ha mostrado en el ejemplo anterior es:

```
tipo_de_valor_de_regreso nombre_la_función (lista de parámetros)
{
    Declaraciones
    Enunciados
}
```

Donde:

tipo_de_valor_de_regreso: El tipo de valor de regreso el valor que una función devolverá.[1]

nombre_de_la_función: es cualquier identificador valido[2].

lista_de_parámetros: lista separada por comas que contiene las declaraciones de los parámetros recibidas por la función al ser llamada[3].

Declaraciones y Enunciados: forman el cuerpo de la función. El cuerpo de la función también se conoce como un bloque.

Un bloque es un enunciado compuesto que incluye declaraciones. Las variables pueden ser declaras en cualquier bloque y los bloques pueden estar anidados. Bajo ninguna circunstancia puede ser defina una función en el interior de otra función.

---

[1] Un tipo de valor de regreso no especificado será siempre supuesto por el compilador como *int*.

[2] Un identificador válido es cualquier nombre formado por letras y números, excluyendo palabras reservadas del propio lenguaje como son por ejemplo *if, while, void, int*, etc.

[3] Si una función no recibe ningún valor, la lista de parámetros es *void*. Para cada parámetro deberá ser enlistado de forma explícita un tipo, a menos de que el parámetro sea del tipo *int*.

Existen varias formas de regresar el control al punto desde el cual se invocó a una función.

a) Si la función no regresa un resultado, el control sólo se devuelve cuando se llega a la llave derecha que termina la función, o al ejecutar el enunciado

return;

b) Si la función regresa un resultado, el enunciado es:

return valor;

Todas las variables declaradas en las definiciones de funciones son variables locales lo que significa que son conocidas solo en la función en la cual están definidas (Deitel & Deitel, 2004).

La mayor parte de las funciones tienen una lista de parámetros que proporcionan la forma de comunicar información entre funciones. Los parámetros de una función también son variables locales.

6.3 Llamadas de función.

Las formas de invocar funciones en muchos lenguajes de programación son la llamada por valor y la llamada por referencia. Para el caso de que los argumentos se pasan en llamada por valor, se efectúa una copia del valor del argumento y ésta se pasa a la función llamada. Las modificaciones a la copia no afectan al valor original de la variable de la función invocadora. –cuando un argumento es pasado en llamada por referencia, el llamador de hecho permite que la función llamada modifique el valor original de la variable (Deitel & Deitel, 2004).

Llamadas por valor.

Todas las funciones disponen de una zona de memoria donde se almacenan las variables definidas dentro del cuerpo de una función, las variables que actúan como argumentos formales, así como distintos resultados que se vayan produciendo a lo largo de ejecución de la misma.

En el momento que una función comienza a ejecutarse se realiza una reserva de memoria para almacenar todos los datos mencionados, cuando la función finaliza su ejecución se libera dicha zona de memoria y por consiguiente se pierden los datos almacenados en ella.

La llamada por valor debería se utilizada siempre que la función llamada no necesite modificar el valor de la variable original de la función invocadora. Esto

evita efectos colaterales accidentales que obstaculizan de forma importante el desarrolla de sistemas correctos y confiables de software. La llamada por referencia debe ser utilizada solo en funciones llamadas confiables, que necesitan modificar la variable original (Deitel & Deitel, 2004).

Al realizar llamadas por valor se realiza una copia del valor de la variable y esta copia se pasa a la función llamada. De tal forma que los cambios a la copia de la función llamada no afectan el valor contenido en la variable original (ver anexo III).

En C, todas las llamadas son llamadas por valor. Sin embargo es posible simular la llamada por referencia mediante la utilización de operadores de dirección y de indirección (Deitel & Deitel, 2004).

Llamadas por referencia.

En muchos casos cuando se requiere modificar una o más variables del llamador o bien es necesario pasar un objeto de datos grande y evitar así la sobrecarga de pasar un objeto en llamada por valor, C permite "simular" las llamadas por referencia.

Es frecuente emplear apuntadores y el operador de indirección[4] para simular las llamadas por referencia[5] (Deitel & Deitel, 2004).

C implementa esta funcionalidad con el uso de referencias. Una referencia es un alias a una localización de memoria, es decir permite referirse a la misma zona de almacenamiento que otro valor. El paso de valores por referencia permite tener parámetros que se refieran a las mismas posiciones de memoria que los parámetros reales usados en al llamada. Por esta razón, toda modificación que se realice sobre los parámetros formales pasados por referencia se aplicará a los parámetros reales. Para indicar que un determinado parámetro se pasa por referencia, el nombre del parámetro debe ir precedido del símbolo &.

Al simular estas llamadas lo que se hace es pasar a la función las direcciones de los argumentos mediante el operador de dirección "&" a la variable que modificara el valor (ver anexo IV).

Diferencias entre paso por valor y paso por referencia.

- ✓ En el paso por valor se realiza una copia del valor del parámetro en el momento de la llamada. Para datos de gran tamaño esto puede representar un problema de eficiencia.

[4] Referirse a un valor a través de un apuntador se conoce como indirección.

[5] Los apuntadores son variables que contienen direcciones de memoria como sus valores. Los nombres de las variables se refieren directamente a un valor y un apuntador se refiere indirectamente a un valor.

- ✓ En los parámetros pasados por valor, si se hacen cambios a los parámetros formales, los parámetros originales no son modificados.
- ✓ En los parámetros pasados por referencia, la llamada debe usar como parámetros reales valores de otra forma no podrán ser modificados.
- ✓ En el paso por referencia lo que se copia es una dirección aunque el parámetro en cuestión se refiera a un tipo estructurado (Fatos, Martin Prat, Vazquez, Gómez, & Molinero, 2006)

7. Bibliografía

Deitel, H., & Deitel, P. (2004). *Como programar en C/C++ y Java.* México: Prentice Hall.

Fatos, X., Martin Prat, A., Vazquez, P.-P., Gómez, J., & Molinero, X. (2006). *Programación en C++ para ingenieros.* Madrid: Thomson.

GCC, the GNU Compiler Collection. (s.f.). Recuperado junio de 2012, de http://gcc.gnu.org/

Gottfried, B. (1998). *Programación en C.* México: McGraw Hill.

Joyanes Aguilar, L., & Zahonero Martínez, I. (2001). *Programación en C: metodología, algoritmos, estructuras de datos y objetos.* Mc Graw Hill.

López Román, L. (2003). *Programación Estructurada. Un enfoque algoritmico.* México: Alfaomega.

Muñoz Caro, C., Niño Ramos, A., & Vizcaíno Barceló, A. (2002). *Introducción a la Programación con Orientación a Objetos.* Madrid: Prentice Hall.

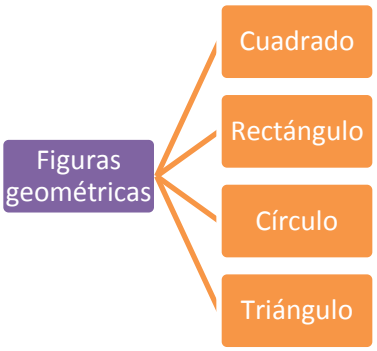
8. ANEXOS

8.1 Anexo I

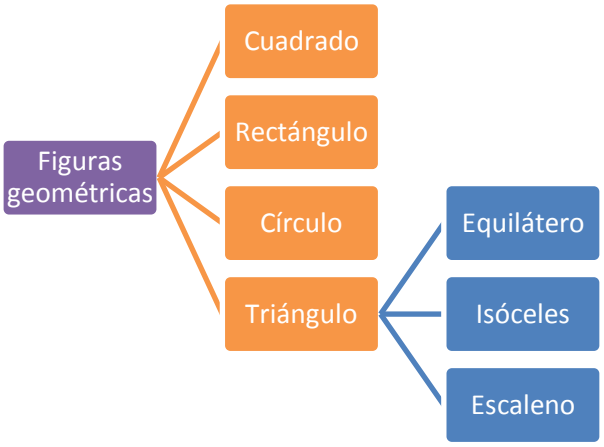
Se desea calcular el área y perímetro de distintas figuras geométricas (cuadrado, rectángulo, circulo y triangulo). Es claro el objetivo, sin embargo dar respuesta a cada uno de los planteamientos requiere la instrumentación de tareas de manera específica.

Aplicando la técnica de modularización es claro que se tiene identificado el conjunto de fórmulas asociado para realizar los cálculos correspondientes de acuerdo a la figura de la que se trate.

Entonces es posible plantear el siguiente esquema para abordar el problema:



Así cada módulo hace una tarea simple, clara y específica y entonces se puede diseñar fácilmente. Y para el caso que algún módulo requiere de otros subordinados estos se van agregando en esta estructura jerárquica.



El siguiente paso es diseñar la estructura lógica de cada módulo planteado utilizando pseudocódigo.

8.2 Anexo II

Implementación de una función en pseudocódigo

Elaborar un algoritmo que solicite dos enteros y evalué la función

$$f = \frac{x!}{y! * (x - y)!}$$

Donde los valores de x y y deben ser mayores a cero

```
Algoritmo CALCULA_F
Módulo Principal
        Declaraciones
                Variables
                        X, Y tipo entero
                        F tipo real
        Escribir "Cuales son los valores de x, y"
        Leer X, Y
        Si (X>0 y Y>0 y X>=Y) entonces
                F=Factorial(X) / (Factorial(Y) * Factorial(X-Y))
                Escribir "El valor de F es: ", F
        Otro
                Escribir "Números incorrectos"
        Fin Si
Fin Módulo Principal

Función Factorial (Val N tipo entero) : Entero
        Declaraciones
                Variables
                        I, Fact=1 tipo entero
        Fact=1
        Si N<>O entonces
                Para I=1, I<=N, inc 1
                        Fact=Fact * I
                Fin Para
        Fin Si
        Factorial=Fact
Fin Función Factorial
```

8.3 Anexo III

Implementación de una función en C

Elaborar un programa que solicte al usuario dos números enteros y evalué la función

$$f = \frac{x!}{y! * (x-y)!}$$

Donde los valores de x y y deben ser mayores a cero

```
#include <stdio.h>

int factorial(int);

main(){
	int x, y;
	float f;

	printf ("Cuales son los valores de x y de y");
	scanf ("%d %d", &x, &y);
	if (x>0 && y>0 && x>y){
		f=factorial(x) / (factorial (y) * factorial (x-y));
		printf("El valor de F es: %.2f", f);
	}
	else
	{
		printf("Números incorrectos \n");
	}
}

int factorial( int n)
{
	int i, fact;

	fact=1;
	if (n!=0)
	{
		for (i=1; i<=n; i++)
			fact=fact*i;
	}
	return fact;
}
```

8.4 Anexo IV

Ejemplo de función con paso de parámetros por valor

```
#include <stdio.h>

int potencia (int);

main ()
{
        int num=5;
        printf("El valor original del numero es: %d\n",num);
        num=potencia(num);
        printf("El nuevo valor de numero aplicada la función con paso por valor es: %d\n", num);
        return 0;
}

int potencia(int x)
{
        return x*x*x;
}
```

Ejemplo de función con paso de parámetros por referencia

```
#include <stdio.h>

int potencia (int *);

main ()
{
        int num=5;
        printf("El valor original del numero es: %d\n",num);
        num=potencia_ref (&num);
        printf("El nuevo valor de numero aplicada la función con paso por referencia es: %d\n", num);
        return 0;
}

int potencia_ref (int *xPtr)
{
        *xPtr=*xPtr * *xPtr * *xPtr;
}
```