Tema 6 – Modularización y funciones

1. Modularización

1.1. Importancia de la modularización

En el desarrollo de algoritmos complejos, la descomposición de problemas y la modularización son fundamentales. Utilizando por ejemplo un diseño descendente (Top-Down), se logra dividir un problema en un conjunto de tareas más sencillas de resolver.

Cada tarea recibe el nombre de **subprograma**, **módulo** o **rutina**, y cada una puede desarrollarse de manera independiente del resto.

El concepto de **modularización** se asocia al concepto de **reuso**. La reutilización consiste en aprovechar el trabajo realizado en desarrollos anteriores, de manera tal que no sea necesario codificar soluciones que ya fueron probadas.

Modularizar significa descomponer un problema en partes funcionalmente independientes. Cada una de estas partes se denomina **módulo**.

La descomposición busca lograr:

Alta Cohesión	Medida del grado de identificación de un módulo con una función concreta.
Bajo Acoplamiento	Medida de la interacción de los módulos que constituyen un programa.

Un módulo o subprograma puede realizar las mismas acciones que un programa (aceptar datos, realizar cálculos y devolver resultados).La diferencia está en que el módulo se ejecuta cada vez que el programa principal lo invoca. Además, un módulo puede invocar a otros.

Cada módulo debe tener un nombre que lo identifique y, en algunos casos, una serie de parámetros asociados, que se explicarán más adelante. El nombre del módulo se utiliza para su *invocación.*

Si un programa o subprograma necesita ejecutar las acciones que corresponden a algún módulo, debe invocarlo usando su nombre. La invocación pasa el control al módulo, una vez ejecutada la última instrucción del módulo, el control retorna al programa que lo llamó, tal como se muestra en la figura 6.1. Cada lenguaje de programación podrá utilizar una sintaxis diferente para este proceso.

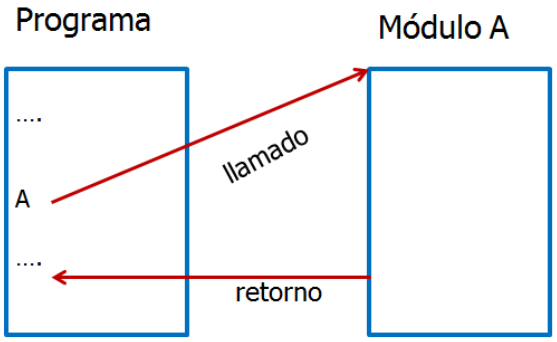


Figura 6.1. Invocación del módulo A

Los módulos representan tareas específicas bien definidas que deben comunicarse entre sí adecuadamente y cooperar para conseguir un objetivo común.

- Cada módulo contiene acciones, tareas ó funciones
- Cada módulo debe representar los objetos relevantes del subproblema a resolver

Las cuestiones que se presentan en la modularización son las siguientes:

¿Qué es un módulo en nuestras soluciones?

¿Cómo debe ser un módulo de software en nuestros programas?

¿Existe una metodología para trabajar?

La metodología que se adecua es el diseño descendente (Top Down). Es una técnica para resolver problemas que comienza descomponiendo el problema principal en subproblemas, para luego diseñar las soluciones específicas para cada uno de ellos.

TOP DOWN
Ir de lo general a lo particular
Dividir … conectar … y verificar

Ejemplo 1: Realizar una aplicación (algoritmo) que ayude a los niños a comprender las operaciones aritméticas fundamentales: Suma, Resta, Multiplicación, División. Ver Fig. 6.2.

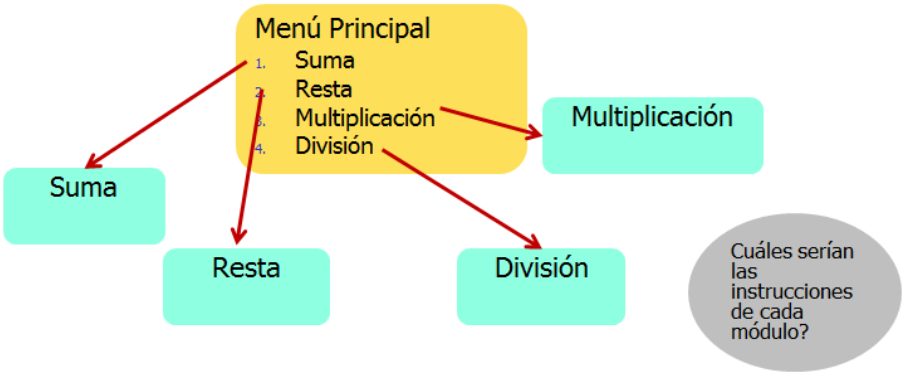


Figura 6.2. Modularización de las operaciones aritméticas

Ejemplo 2: Sistema de gestión hotelera, los distintos módulos atienden cuestiones específicas: Clientes, Reservas y Pagos. Todos ellos contribuyen a los objetivos del sistema. Si fuera necesario incorporar o modificar alguna funcionalidad en la gestión de pagos, no es necesario afectar a los otros módulos. Ver Fig. 6.3.

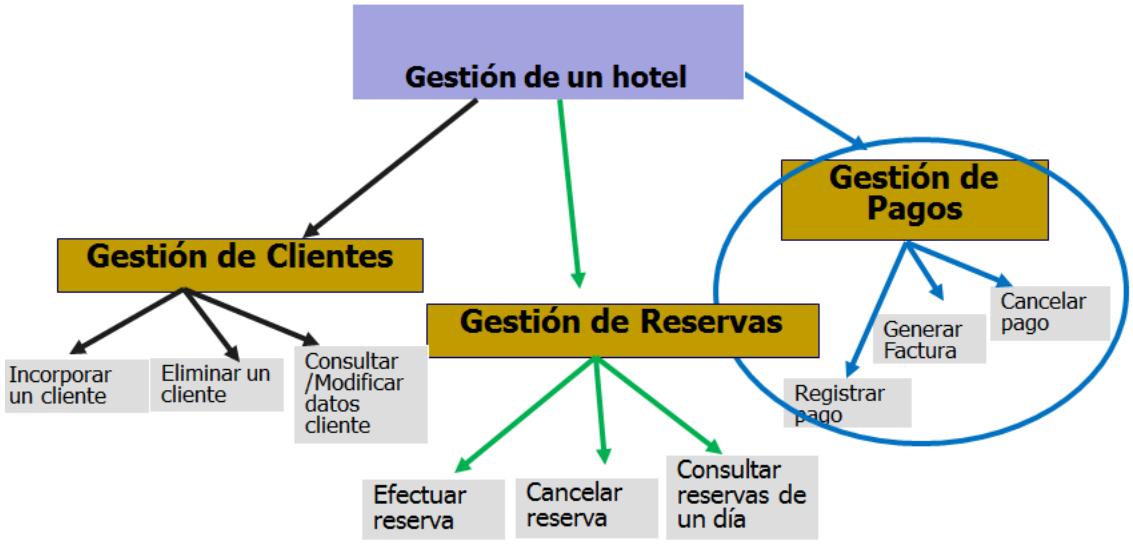


Figura 6.3. Modularización en un sistema de gestión hotelera

1.2. Ventajas de la modularización

- **Mayor productividad:** Al dividir un sistema de software en módulos funcionalmente independientes, un equipo de desarrollo puede trabajar simultáneamente en varios módulos, incrementando la productividad (es decir reduciendo el tiempo de desarrollo global del sistema).
- **Reusabilidad:** Un objetivo fundamental de la Ingeniería de Software es la *reusabilidad*, es decir la posibilidad de utilizar repetidamente el producto de software desarrollado. La descomposición funcional que ofrece la modularización *favorece el reuso.*

- **Facilidades de mantenimiento correctivo:** La división lógica de un sistema en módulos permite aislar los errores que se producen con mayor facilidad. Esto significa poder corregir los errores en menor tiempo y disminuir los costos de mantenimiento de los sistemas.
- **Facilidades de evolución del sistema:** Los sistemas de software reales crecen (es decir aparecen con el tiempo nuevos requerimientos del usuario). La modularización permite disminuir los riesgos y costos de incorporar nuevas prestaciones a un sistema en funcionamiento.
- **Mayor Legibilidad:** Un efecto de la modularización es una mayor claridad para leer y comprender el código fuente. El ser humano maneja y comprende con mayor facilidad un número limitado de instrucciones directamente relacionadas.

Los distintos lenguajes de programación utilizan diferentes recursos para especificar la modularización. Por ejemplo:

- Subroutine
- Module
- Procedure
- Function
- Package
- Class, etc.

Pascal utiliza procedimientos y funciones. El procedimiento ejecuta un conjunto de acciones y no devuelve un valor. Por el contrario, las funciones, ejecutan un conjunto de acciones y siempre devuelven un valor.

En C, los subprogramas se denominan **funciones**, y pueden o no devolver un valor. Cuando no devuelven un valor, son equivalentes a los **procedimientos** en Pascal y otros lenguajes.

2. Las funciones en C

Una función realiza una tarea específica agrupando un conjunto de instrucciones con un nombre. Para que se ejecuten las instrucciones contenidas en la función ésta se debe invocar o llamar mediante su nombre desde otra función, la cual puede ser **main**.

Cada función puede ser diseñada, verificada y depurada de manera independiente; esto permite que cada función se pueda probar sin tener que esperar a que estén programadas todas las demás funciones que también se usarán en el programa. Sin embargo, las funciones no pueden actuar de manera autónoma, es decir, no pueden ejecutarse por sí mismas: su ejecución depende de que sea invocada por otra función y esto a su vez debe estar relacionado siempre con la ejecución de una función **main**.

En el lenguaje C se pueden distinguir dos tipos de funciones: las **predefinidas** y las **definidas por el programador.** Las funciones predefinidas se encuentran en las bibliotecas estándar de C, y es necesario hacer uso de la directiva #include para invocarlas. Ejemplo de funciones predefinidas: **printf**, **scanf**, **pow**.

Con respecto a las definidas por el programador, las diseña el programador según sus necesidades.

2.1. Función main ()

Es la función principal y puede contener de pocas a muchas líneas; su papel es coordinar a las otras funciones mediante llamadas o invocaciones. El siguiente diagrama muestra la jerarquía que existe en un programa modular en lenguaje C, en el cual se puede ver que siempre debe existir una función **main** y ésta puede hacer uso de cualquier cantidad de funciones, ya sean creadas por el usuario o predefinidas en el lenguaje. Las funciones invocadas por **main** pueden llamar a su vez otras funciones.

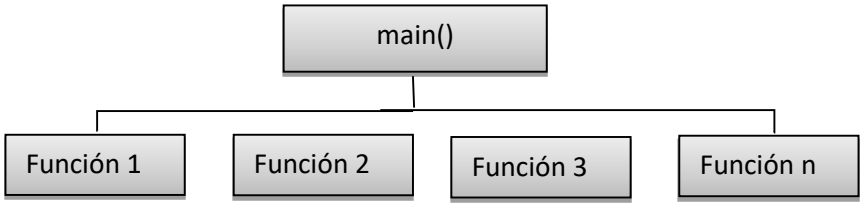


Figura 6.4 Jerarquía de funciones en C

2.2. Estructura de una función

Una función se compone de una **cabecera** y un **cuerpo**. En la cabecera se especifican los parámetros con los que la función va a trabajar y el tipo de datos del valor de retorno. El cuerpo es la codificación del algoritmo de la función.

```
Tipo_de_retorno nombreFuncion(lista de parámetros) {
      cuerpo de la función;
      return expresión;
}
```

Tipo_de_retorno	Es el tipo de dato del valor que devuelve la función. Si se omite el tipo de dato, asume entero.
nombreFuncion	Es el nombre asignado a la función.
Lista de parámetros	Lista de variables con sus respectivos tipos de datos: tipo1 parametro1, tipo2 parametro2,..., Cuando existen, éstos son los datos que debe recibir la función cuando se le invoque.
Cuerpo de la función	Son las sentencias o instrucciones que ejecutará la función cada vez que sea invocada.
Expresión	Valor que devuelve la función (cuando devuelve valor).

a) Prototipo de una función

La declaración de una función se denomina “prototipo”. El prototipo describe la lista de argumentos que la función espera recibir y el tipo de datos de su valor de retorno. Contiene la cabecera de la función, siendo opcional incluir los nombres de variables, y termina con punto y coma para indicar al compilador que es una instrucción.

Ejemplo: Función que retorna el valor absoluto de un número, su prototipo será el siguiente:

```
floatvalorAbsoluto (float);
```

Describe una función llamada valorAbsoluto que recibe un valor de tipo float y devuelve un valor del mismo tipo.

Los prototipos se sitúan normalmente al principio del programa, antes de la función main. El compilador utiliza los prototipos para validar que el número y los tipos de datos de los parámetros reales son los mismos que el número y los tipos de datos de los parámetros formales de la función llamada. En caso de inconsistencia se visualiza un error.

b) Invocar a una función

Si desde el programa principal se desea invocar a una función, previamente debe estar *prototipada*. El **prototipo** se escribe antes del programa principal main(), como se muestra en la Fig, 6.4.

c) Desarrollo de una función

Para desarrollar una función, se define su cabecera y su cuerpo. La cabecera es idéntica al prototipo pero le pone nombre a los argumentos, que constituyen los parámetros de la función.

En el ejemplo de la figura 6.4 se muestra el código que corresponde al desarrollo de la función que se ubica después del cuerpo de la función principal **main().**

```c
#include <stdio.h>
float  valorAbsoluto (float); /* prototipo de la función */
int main() {
    float v, a;
    printf("Ingrese un valor numérico: ");
    scanf("%f", &v);
    /* invoco a la función */
    a = valorAbsoluto(v);
    printf("El valor absoluto de %f es %f", v, a);
    return 0;
}
/* desarrollo de la función */
float valorAbsoluto (float d) {
    float valorDevuelto;
    valorDevuelto = d;
        if (d < 0) {
            valorDevuelto = -valorDevuelto;
        }
    return valorDevuelto;
}
```

Figura 6.4. Prototipo, invocación y desarrollo de una función

2.3. Formas de implementación

Como ya se mencionó, cada función se diseña de manera independiente. Según el propósito específico, las funciones pueden ser diseñadas o definidas, de las siguientes maneras:

a) **Funciones sin paso de parámetros.** Son subprogramas que no requieren información adicional de su entorno, pues simplemente ejecutan una acción cada vez que son invocadas.
b) **Funciones con paso de parámetros.** Para la ejecución de estos subprogramas se requiere además de su invocación, que se le pase información adicional de su entorno.
c) **Funciones que no regresan valor.** Subprogramas que luego de su ejecución no devuelven al entorno algún valor como resultado de su ejecución.
d) **Funciones que regresan valor.** Funciones que luego de su ejecución generan un valor como resultado y "entregan" ese valor a su entorno.

Estas funciones se pueden combinar, es decir, se puede diseñar una función con parámetros que regresen valor o que no lo hagan, si así se requiere; o bien diseñar una función sin parámetros que regrese valor en un mismo programa. Esto dependerá del programador, y de cómo decida que es más conveniente el diseño de la función.

El **entorno** de cualquier función es la función por la que es invocada. Por ejemplo, si **main()** invoca a una función diseñada con paso de parámetros y que regresa un valor, entonces **main()** es el entorno de ésta, y será **main()** la que le proporcione la información (parámetros) que dicha función requiera y la que reciba el valor que dicha función devuelva.

2.4. Funciones con paso de parámetros

Una función con paso de parámetros es aquella que además de ser invocada requiere información por parte del subprograma que la llama. Esta información se refiere a los datos de entrada que se necesitan para que la función trabaje.

La forma general de este tipo de función es la siguiente:

```
tipo_dato identificador (tipo_dato parámetro ){
      declaración de variables locales; cuerpo de la función;
      return (valor);
}
```

Lo nuevo en el diseño de la misma es el contenido de los paréntesis:

```
      #include<stdio.h>
      float sumaNumeros (float, float); /* prototipo de la función */
```

```
int main( ) {
      float n1, n2, c;
      printf("Ingreso valor 1 ");
      scanf("%f", &n1);
      printf("Ingreso valor 2 ");
      scanf("%f", &n2);
      c=sumaNumeros (n1,n2); /* invoca a la función sumaNumeros */
      printf("La suma de numeros es %f", c);
}
/* define la función */
float sumaNumeros (float a, float b) {
return a + b;
}
```

Observe cómo la función **sumaNumeros** se declara con dos parámetros de tipo *formal* (**a** y **b**) los cuáles son una especie de "molde" en los cuales la función recibe los valores que necesita para trabajar.

Las acciones de main, al ejecutarse, son las siguientes: solicitar los datos al usuario y almacenarlos en **n1** y **n2** respectivamente, e invocar a **suma** con los valores almacenados en **n1** y **n2** (los contenidos de n1 y n2, entran a la función a través de los parámetros a y b en el orden en que se enviaron, es decir, el valor de n1 ingresa a través de a y n2 entra a través de b. La función **suma** devuelve el valor de la suma de a + b a main que se almacena en la variable c. Finalmente main imprime el valor de c.

Parámetros formales y argumentos

Los parámetros **formales** (también denominados parámetros *ficticios*) son las variables que se declaran dentro del paréntesis junto con la función; en este caso **a** y **b**. Por otro lado, los valores con que se invoca a la función se denominan **argumentos o parámetros reales o parámetros actuales**. En el ejemplo tales argumentos son representados por **n1** y **n2**.

Frecuentemente, el término parámetro se utiliza indistintamente tanto para referirse a los ficticios como a los reales; sin embargo, aquí se denominarán **parámetros** *a los ficticios* y **argumentos** a los *reales*. Dicho de otra manera: son **parámetros** los que se declaran en la definición de la función y **argumentos** los valores con que se invocan a la misma.

Cabe señalar también que los parámetros son variables siempre, mientras que los argumentos pueden ser variables, constantes, expresiones aritméticas e incluso lo que devuelva la llamada a otra función, siempre y cuando los argumentos coincidan en cantidad, tipo de dato y orden con los parámetros, tal como se hayan declarado en la función.

En el ejemplo, **sumaNumeros** fue definida para recibir dos parámetros de tipo real (a y b), por lo tanto, cuando en main se invoca suma, a esta variable se le entregan dos argumentos (n1 y n2) también de tipo real.

2.5. Paso de parámetros por valor y por referencia

En programación existen dos formas de pasar variables a una función: **paso por valor** y **paso por referencia**. En la primera forma, la función recibirá una copia de la variable que se pasa y, cualquier modificación que se realice, afectará a dicha copia. En la segunda forma, la función recibe prácticamente la variable original, es decir, si se realiza algún cambio en el parámetro de la función, esto equivale actuar directamente sobre la variable original.

2.5.1. Funciones con parámetros por valor

Los parámetros por valor reciben una copia de lo que valen los argumentos; su manipulación es independiente, es decir, una vez finalizada la función, los argumentos continúan con el valor que tenían antes.

Ejemplo de parámetros por valor:

```
#include<stdio.h>
void intercambio(int, int);
int main() {
      int n1, n2;
      scanf("%d %d", &n1, &n2);
      intercambio(n1,n2);
      printf("Los valores son n1 = %d y n2 = %d", n1, n2);
      return 0;
}
void intercambio(int x, int y) {
      int aux;
      aux = x;
      x = y;
      y = aux;
}
```

En este ejemplo, la función main solicita dos números al usuario y los recibe en **n1** y **n2**. Si en tiempo de ejecución el usuario ingresa los valores 9 y 10, la función main pasa como argumentos a n1 y n2 a la función *intercambio*, la cual al ejecutarse recibe el 9 a través de **x** y el 10 a través de **y**, y los intercambia. Al terminar de ejecutarse la función *intercambio*, main imprime los valores de n1 y n2. En pantalla aparecería:

Los valores son: n1 = 9 y n2 = 10

En este ejemplo es posible observar lo que se explicó antes: la función *intercambio* no tiene el poder de modificar los valores de los argumentos, sólo los utiliza. Esto se debe a que cuando un parámetro ingresa **por valor** a una función, ésta tiene acceso sólo a una copia del valor del argumento, pero sin acceder a la localidad de memoria donde está almacenado el dato. En el ejemplo, la función *intercambio* recibió sólo copia de los valores de **n1** y **n2**, y aunque intercambió los parámetros, los argumentos permanecieron con su mismo valor una vez finalizada la función.

2.5.2. Funciones con parámetros por referencia

a) Punteros y direcciones de memoria.

Para entender este concepto, se requiere conocer previamente sobre punteros y direcciones de memoria.

Un puntero es un **tipo especial de variable que almacena una dirección de memoria**.

Cuando se utiliza la función scanf() para el ingreso de datos, se usa el operador de dirección **&** (léase ampersand) explicando que al anteponer este operador al identificador de una variable, hacemos referencia a su dirección de memoria. Ejemplo: Si **a** es una variable, entonces **&a** es la dirección de memoria donde se aloja el valor que contiene la variable **a**.

Las direcciones de memoria son valores de un tipo de datos particular, llamado "**puntero**".

Si la variable **a** es de tipo **int** entonces el tipo de dato de su dirección de memoria es "puntero a int" o simplemente int* (léase int asterisco).
Ejemplo:

```
#include<stdio.h>
intmain() {
      /* se define la variable a */
      int a = 0;
      /* se define la variable p de tipo "puntero a int" */
      int* p;
      /* se asigna a p la dirección de a */
      p = &a;
      /* se asigna el valor 12 al contenido de p */
      *p = 12;
      /* se visualiza el valor de la variable a */
      printf("valor de a = %d\n", a);
}
```

La salida de este programa (lo pueden probar) es **a = 12.**

En el código ejemplo se define la variable **a** de tipo **int** y la variable **p** de tipo **int*.** Como **p** tiene la capacidad de contener una dirección de memoria se le asigna la dirección de la variable **a** que se obtiene anteponiendo a esta variable el operador **&.**

Luego se accede al espacio de memoria direccionado por **p** a través de ***p** y se asigna allí el valor 12. Como este espacio de memoria corresponde a la variable **a**, indirectamente estamos asignando el valor 12 a esta variable.

En el lenguaje C el asterisco se utiliza para definir variables de tipo puntero, pero también se utiliza para hacer referencia al contenido del espacio de memoria direccionado por un puntero.

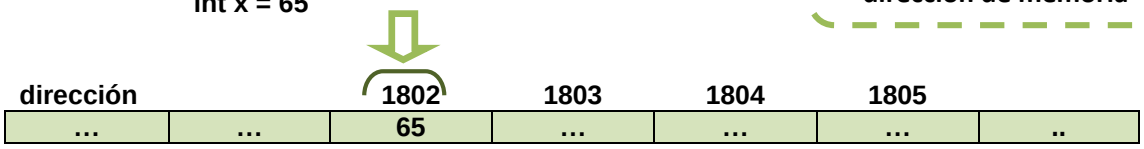
El operador de indirección * (asterisco)
Este es el operador inverso de &. Aplicando el operador & a una variable obtenemos su dirección de memoria. En cambio, aplicando "*" (asterisco) a un puntero obtenemos el contenido almacenado en esa dirección de memoria. Se lo conoce también operador de "desreferencia" .
Ejemplo:

```
#include<stdio.h>
int main() {
      int a = 0;
      *(&a) = 10;
      printf("valor de a = %d\n", a);
      return 0;
}
```

En este código se asigna 10 al contenido de la variable **a**, que es lo mismo que asignar 10 a esta variable. El resultado del printf es **Valor de = 10.**

Un **puntero** es una variable que contiene una **dirección de memoria**

Una **dirección** de memoria y su **contenido**no son la misma cosa.



La **dirección** de la variable **x** es 1802, el **contenido** de la variable **x** es 65

b) Paso por referencia

El paso de parámetros por referencia implica utilizar los operadores **&** y *****.

El operador & se antepone a una variable, dando con ello acceso a su direcciónde memoria asignada. Se utiliza en los argumentos para pasar por referencia dicha variable. El ejemplo clásico de una función de este tipo es **scanf,** donde los valores introducidos por medio del teclado son almacenados por referencia en la variable o variables indicadas; cuando se utiliza la función, se le llama con el operador & antes del identificador de cada variable.

```
scanf (&a, &b, &c);
```

Luego de recibir las variables argumento, lo que se almacena en ellas permanece incluso después de finalizada la función, dado que pasaron por referencia.

El operador * es un apuntador que "apunta" a la dirección de la variable pasada como argumento. Se utiliza tanto en la declaración de los parámetros formales de la función como en el cuerpo de la misma. Debe aparecer antes del nombre de un parámetro formal en la cabecera para indicar que dicho parámetro será pasado por referencia, y debe aparecer en el cuerpo de la función antepuesto al nombre de un parámetro formal para acceder al valor de la variable externa a la función y referenciada por el parámetro formal.

Observe el siguiente ejemplo, que es una versión del programa anterior. Aquí se utiliza el paso por referencia, lo cual luego de la ejecución presenta resultados diferentes a los de la versión con paso de parámetros por valor:

```
#include<stdio.h>
void intercambio(int *, int *);
int main() {
```

```
        int n1, n2;
        scanf("%d %d", &n1, &n2);
        intercambio(&n1,&n2);
        printf("Los valores son n1 = %d y n2 = %d", n1, n2);
        return 0;
}
void intercambio(int *x, int *y) {
        int aux;
        aux = *x;
        *x = *y;
        *y = aux;
}
```

Si en tiempo de ejecución se ingresan los valores n1=5 y n=8, estas variables son "entregadas" a la función intercambio, y como pasan por referencia, de ese punto en adelante lo que suceda con **x** e **y** dentro de la función afectará directamente a los argumentos, es decir, a n1 y n2, por lo que al finalizar la función el mensaje que se mostrará en pantalla será:

Los nuevos valores son n1 = 8 y n2 = 5

Observe que los valores se invirtieron: ahora **n1** aparece con 8 y **n2** con 5.

3. Alcance de variables

Una característica de las variables de un programa es el "alcance" o ámbito de referencia de las mismas. El alcance determina la zona del programa donde la variable es definida y conocida. De acuerdo a este alcance, las variables utilizadas en los programas y/o módulos se clasifican en: **locales** y **globales**.

3.1. Variables locales

Las variables locales tienen las siguientes características:

- Sólo pueden ser reconocidas y utilizadas por la función en la que están declaradas; ninguna otra, ni siquiera main, que es el programa principal, tiene acceso a las variables locales declaradas en otra función.
- Los espacios reservados en memoria para variables locales están disponibles sólo en el momento en que se está ejecutando la función donde fueron declaradas, una vez que se termina la ejecución de la función desaparecen de la memoria.

El uso de variables locales presenta la ventaja importante de construir módulos altamente independientes, donde las comunicaciones necesarias desde los programas que los invoquen deben hacerse necesariamente mediante el pasaje de parámetros. De esta manera se evitan posibles efectos colaterales producidos por operaciones sobre datos globales dentro de un módulo.

Conceptualmente puede decirse que desarrollar módulos con independencia funcional, variables locales y mínima comunicación externa mediante parámetros es una buena práctica de programación, que favorece la reutilización y el mantenimiento del software.

3.2. Variables globales

Se declara fuera de todos los módulos del programa y puede ser usada por el programa y todos los módulos del mismo.
Ejemplo:

```
#include<stdio.h>
void incrementar ();
int i = 0;  /* i es una variable global */
int main() {
        while (i< 10) {
                incrementar();
                printf("valor de i = %d\n", i);
        }
        return 0;
}
void incrementar() {
        i = i + 1;
}
```

En este ejemplo se puede ver que la variable i se define fuera de la función principal main, y es utilizada dentro del main y dentro de la función incrementar sin necesidad de que sea definida en ninguna de estas funciones. El uso de variables globales es una mala práctica de programación que puede generar problemas de mantenibilidad y legibilidad del código.

4. Funciones predefinidas en C

Todas las versiones del lenguaje C ofrecen una biblioteca estándar de funciones que proporciona soporte para las operaciones más frecuentes. Las funciones *estándar o predefinida,* se dividen en grupos*,* todas las funciones que corresponden al mismo grupo se declaran en el mismo *archivo de cabecera.*

Algunos de los grupos son:

<stdio.h> - funciones de entrada y salida

<math.h> - funciones matemáticas

<ctype.h> - funciones de manejo de caracteres

<string.h> - funciones de manejo de cadenas

<time.h> - funciones de fecha y hora

Para utilizar una función se debe conocer la cantidad y tipo de argumentos y tipo de valor de retorno.
Como ejemplo se muestran algunas funciones matemáticas de uso más habitual:

Funciones	Significado	Funciones	Significado
fabs (x)	Valor absoluto de x	**ceil (x)**	Redondeo entero superior
pow (x, y)	Potencia de x elevado a y	**floor (x)**	Redondeo entero inferior
sqrt (x)	Raíz cuadrada de x	**exp (x)**	Exponenciación (número e elevado a
sin (x)	Seno de x	**log (x)**	Logaritmo neperiano de x
cos (x)	Coseno de x	**log10 (x)**	Logaritmo decimal de x
tan (x)	Tangente de x	**atan (x)**	Arcotangente de x

Ejemplo de uso de funciones matemáticas en C;

```
#include <stdio.h>
#include <math.h>
float n;
int main () {
    printf("Ingrese n: ");
    scanf("%f", &n);
    printf("\nRedondear hacia arriba (CEIL) = %.1f\n", ceil(n));
    printf("\nRedondear hacia abajo (FLOOR) = %.1f\n", floor(n));
    printf("\nValor absoluto de %.2f (FABS) = %.2f\n", n, fabs(n));
    printf("\nRaiz cuadrada de %.2f (SQRT) = %.2f\n", n, sqrt(n));
    printf("\n%.2f elevado al cuadrado (POW) = %.4f\n", n, pow(n,2));
    return 0;
}
```

Videos: Funciones en C

Javier Salmerón – Prototipo de funciones:
https://www.youtube.com/watch?v=9P-oIbzuZB8&index=3&list=PLq5lWpKgT2w73Eh0uuVgRztnb1KjmX8vl

Javier Salmerón – Llamada a funciones:
https://www.youtube.com/watch?v=_cHFLBGyLZ8&list=PLq5lWpKgT2w73Eh0uuVgRztnb1KjmX8vl&index=4

Javier Salmerón - Cuerpo de funciones:
https://www.youtube.com/watch?v=8SF30WcSQWQ&index=11&list=PLq5lWpKgT2w73Eh0uuVgRztnb1KjmX8vl