Indice

Contenido

Unidad 3
Descomposición de problemas: Modularización

Temario: Funciones de biblioteca, uso. Módulos. Concepto. Clasificación. Ámbito de identificadores. Transferencia de información a y desde procedimientos: los parámetros, tipos de parámetros. Conceptos de acoplamiento y cohesión.

Bibliografía:

[1] Luis Joyanes Aguilar e Ignacio Martínez - Programación en C. Metodología, algoritmos y estructuras de datos.
[2] Cairó, Osvaldo. Fundamentos de Programación. Piensa en C.
[3] Andrés Marzal e Isabel Gracia - Introducción a la programación con C .

1Módulos

1.1Introducción

Para resolver problemas complejos y/o de gran tamaño podemos utilizar uno de los métodos fundamentales que consiste en dividir el problema en problemas más pequeños, llamados subproblemas. Esta técnica de dividir el programa se denomina, Divide y Vencerás. El método de diseño se denomina Diseño Descendente o Top-Down.

La resolución de un problema comienza con una descomposición modular y luego nuevas descomposiciones de cada módulo hasta que el problema original queda reducido a un conjunto de actividades básicas, que no se pueden o no conviene volver a descomponer. Es decir se realiza un proceso de Refinamiento Sucesivo (Stepwise). Luego cada módulo se resuelve y las subsoluciones se fusionan en la solución del problema original.

El lenguaje de programación C fue diseñado como un lenguaje de programación estructurado y modular. Por esta razón, la solución de un problema en C se expresa por medio de un programa que quedará formado por una serie de módulos, cada uno de los cuales realiza una tarea concreta de la tarea total. Todo programa en C cuenta con un módulo indispensable, la función main. Esta función es la primera en ejecutarse y desde ella se invocan los demás módulos. El diseño modular conlleva una serie de beneficios: producir programas más fáciles de escribir, más fáciles de mantener, crear módulos independientes por sus parámetros y realizar pruebas independientes.

1.2Concepto

Un módulo es cada una de las partes de un programa que resuelve uno de los subproblemas en que se divide el problema complejo original. Cada uno de estos módulos tiene una tarea bien definida y algunos necesitan de otros para poder operar. En caso de que un módulo necesite de otro, puede comunicarse con éste mediante una interfaz de comunicación que también debe estar bien definida.

1.3Clasificación

Los módulos básicos se clasifican en Procedimientos y Funciones. Toda función o procedimiento realiza una tarea definida y concreta, sin embargo, la diferencia entre una función y un procedimiento es que, la función retorna un valor en el nombre de dicha función, mientras que el procedimiento no, es decir, el nombre de la función tiene asociado un espacio de memoria capaz de contener un dato, mientras que el

nombre del procedimiento es sólo una referencia a las sentencias vinculadas al módulo y no tiene la posibilidad de contener datos.

Los módulos en C no se pueden anidar. Esto significa que un módulo no se puede declarar dentro de otro módulo. La razón para esto es permitir un acceso muy eficiente a los datos. Los módulos en C tienen un ámbito global, es decir, pueden ser llamados desde cualquier punto del programa.

Los módulos en C pueden ser funciones de bibliotecas o módulos definidos por el programador.

1.3Funciones de biblioteca

1.3.1Introducción

Todas las versiones del lenguaje **C** ofrecen bibliotecas estándares de funciones en tiempo de ejecución que proporcionan soporte para operaciones utilizadas con más frecuencia. Estas funciones permiten realizar una operación con sólo una llamada a la función (sin necesidad de escribir su código fuente).

Las **funciones estándar o predefinidas**, como así se denominan las funciones pertenecientes a la biblioteca estándar, se dividen en grupos; todas las funciones que pertenecen al mismo grupo se declaran en el mismo **archivo de cabecera**.

Algunos de los nombres de los archivos de cabecera estándar más utilizados se muestran a continuación, encerrados entre corchetes tipo ángulo:

`<assert.h>`	`<ctype.h>`	`<errno.h>`	`<float..h>`
`<limits.h>`	`<math.h>`	`<setjmp.h>`	`<signal.h>`
`<stdarg.h>`	`<stdef.h>`	`<stdio.h>`	`<string.h>`
`<time.h>`	`<stdlib.h>`		

En los módulos de programa se pueden incluir líneas `#include`con los archivos de cabecera correspondientes en cualquier orden, y estas líneas pueden aparecer más de una vez.

Para utilizar una función o una macro, se debe conocer el número y tipo de datos de los argumentos y el tipo de dato de su valor de retorno. Esta información se proporciona en el prototipo de la función. La sentencia `#include`mezcla el archivo de cabecera en su programa.

Se pueden incluir tantos archivos de cabecera como sean necesarios en el archivo de programa, incluyendo los propios archivos de cabecera que define el programador con las propias funciones.

1.3.2Funciones matemáticas

Para utilizar estas funciones debe incluirse `#include<math.h>`.

Las funciones matemáticas usuales en la biblioteca estándar son:

Función y Prototipo	Descripción	Ejemplo

Ceil
doubleceil(double x);
	Calcula el valor entero más chico que sea mayor o igual a x.	#include<stdio.h>
		#include<math.h>
		Intmain()
		{
		double x = 6.54321;
		printf("ceil(%f)= %f\n", x, ceil(x));
		return 0;
		}

fabs
doublefabs(double x);
	Devuelve el valor absoluto de x (un valor positivo).	#include<stdio.h>
		#include<math.h>
		Intmain()
		{
		double x = -6.54321;
		printf("fabs(%f)= %f\n", x, fabs(x)
		);
		return 0;
		}

floor
doublefloor(double x);
	Calcula el valor entero más grande que no sea mayor de x.	#include<stdio.h>
		#include<math.h>
		Intmain()
		{
		double x = 6.54321;
		printf("floor(%f)= %f\n", x, floor(x));
		return 0;
		}

fmod
double fmod(double x,
double y);
	Calcula el resto f en coma flotante para la división x/y**,** de modo que x = i*y+f, donde i es un entero, f tiene el mismo signo que x y el valor absoluto de f es menor que el valor absoluto de y.	#include<stdio.h>
		#include<math.h>
		Intmain()
		{
		double x = -6.54321, y = 1.23456;
		printf("fmod(%f) = %f\n", x,fmod(x));
		return 0;
		}

pow
doublepow(double x, double y);
	Calcula x elevado a la potencia de y. Puede producirse un error de dominio si x es negativo e y no es un valor entero. También se produce un error de dominio si el resultado no se puede representar cuando x es cero e y es menor o igual que cero.	#include<stdio.h>
		#include<math.h>
		intmain()
		{
		double x = 6.54321, y = 0.56789;
		printf("pow(%f, %f)= %f\n", x, y, pow(x,y));
		return 0;
		}

sqrt
doublesqrt(double x);
	Devuelve la raíz cuadrada de x; x debe ser mayor oigual a cero.	#include<stdio.h>
		#include<math.h>
		intmain()
		{
		double x = 6.54321;
		printf("sqrt(%f)= %f\n", x, sqrt(x));
		return 0;
		}

1.3.3Funciones trigonométricas

La biblioteca de C incluye una serie de funciones que sirven para realizar cálculos trigonométricos. Para utilizar cualquiera de estas funciones es necesario incluir en el programa, el archivo de cabecera <math.h>.

Función y Prototipo	Descripción	Ejemplo

cos
doublecos(double x);
	Calcula el coseno del ángulo x; x se expresa en radianes.	#include<stdio.h>
		#include<math.h>
		Intmain()
		{
		double x = 3.1416/3.0; printf("cos(%f) = %f\n", x, cos(x));
		return 0;
		}

sin
double sin(double x);
	Calcula el seno del ángulo x; x se expresa en radianes.	#include<stdio.h>
		#include<math.h>
		Intmain()
		{
		double x = 3.1416/3.0;
		printf("sin(%f) = %f\n", x, sin(x));
		return 0;
		}

tan
double tan(double x);
	Devuelve la tangente del ángulo x; x se expresa en radianes.	#include<stdio.h>
		#include<math.h>
		Intmain()
		{
		double x = 3.1416/3.0;
		printf("tan(%f) = %f\n", x, tan(x));
		return 0;
		}

Para pasar un ángulo expresado en grados a radianes, se multiplica los grados por pi y se divide por180.
Pi = 3.14159.

1.3.4Funciones logarítmicas y exponenciales

Las funciones logarítmicas y exponenciales suelen ser utilizadas con frecuencia no sólo en matemáticas, sino también en el mundo de las empresas y los negocios. Estas funciones requieren que se incluya en el programa, el archivo <math.h>.

Función y Prototipo	Descripción	Ejemplo

exp
doubleexp(double x);
	Calcula la función exponencial de x.	#include<stdio.h>
		#include<math.h>
		Intmain()
		{
		double x = -5.567;
		printf("exp(%f) = %f\n", x, exp(x));
		return 0;
		}

log
double log(double x);
	Calcula el logaritmo natural del argumento x	#include<stdio.h>
		#include<math.h>
		Intmain()
		{
		double x = 6.54321;
		printf("log(%f) = %f\n", x, log(x));
		return 0;
		}

1.3.5Funciones aleatorias

Los números aleatorios son de gran utilidad en numerosas aplicaciones y requieren un trato especial en cualquier lenguaje de programación. C no es una excepción, la mayoría de los compiladores incorporan funciones que generan números aleatorios. Estas funciones requieren también el archivo de inclusión <stdlib.h>.

Función y Prototipo	Descripción	Ejemplo

rand
int rand(void);
	Genera un número aleatorio. El número calculado por rand varía en el rango entero de 0 a RAND-MAX. La constante RAND-MAX se define en el archivo stdlib.h en forma hexadecimal (por ejemplo, 7FFF, que equivale a 32767).	#include<stdio.h>
		#include<stdlib.h>
		Intmain()
		{
		unsignedint i;
		printf("30 numerosgeneradosaleatoriamente: \n\n");
		for(i=1; i<30; i++)
		printf("%d\t", rand());
		return 0;
		}

srand
voidsrand(unsignedint semilla);
	La función srandinicializa el generador de números aleatorios. Usa el argumento como una semilla para una secuencia nueva de números pseudo aleatorios para ser retornados por llamadas posteriores a rand. Si srand es entonces llamada con el mismo valor semilla, la secuencia de números pseudo aleatorios será repetida. Si rand es llamada antes de que se hayan hecho cualquier llamada a srand, la misma secuencia será generada como cuando srand fue llamada la primera vez con un valor semilla de 1.	#include<stdio.h>
		#include<time.h>
		#include<stdlib.h>
		Intmain()
		{
		unsignedint i;
		srand(time (NULL)); /* Cada vez que se ejecute el programa, se usa una semilla de acuerdo al reloj de la PC */
		printf("30 numeros generados aleatoriamente: \n\n");
		for(i=1; i<30; i++)
		{
		printf("%d, ", rand());
		}
		return 0;
		}

Siempre que se necesite generar numero aleatorios, se utiliza la función `rand()`. Se debe invocar al procedimiento `srand()` una vez por cada ejecución de programa para obtener un conjunto diferente de números aleatorios en cada corrida.

Si se quiere obtener un número aleatorio entre un subrango determinado [R1…R2], se puede utilizar la función `rand`, y posteriormente modificar estos valores convenientemente

```
y = rand() % (R2 – R1 + 1) + R1
```

1.3.6Funciones de utilidad

C incluyen una serie de funciones de utilidad que se encuentran en el archivo de cabecera stdlib.h y que se listan a continuación.

Función	Descripción	Ejemplo

abs
`intabs(intnum);`
	Devuelve el valor absoluto de num, en formato entero.	`#include<stdlib.h>`
		`#include<stdio.h>`
		`Intmain()`
		`{`
		`Intnum;`
		`printf( "Escriba un numero entero:" );`
		`scanf( "%d", &num );`
		`printf("abs(%d)= %d\n", num, abs(num) );`
		`return 0;`
		`}`

labs
`longintlabs(longintnum);`
	Devuelve el valor absoluto de un número en formato entero largo	`#include<stdlib.h>`
		`#include<stdio.h>`
		`Intmain()`
		`{`
		`Longintnum;`
		`printf("Escriba un numero entero:");`
		`scanf("%d", &num );`
		`printf("labs(%d)= %d\n",num, labs(num));`
		`return 0;`
		`}`

1.4 Módulos definidos por el programador

El usuario o programador puede diseñar e implementar sus propios módulos, los cuales pueden formar parte de su programa o de una librería no estándar. Estos módulos son básicamente funciones o procedimientos.

1.4.1Funciones definidas por el programador

Como se dijo anteriormente, una función es un grupo de sentencias que realizan una tarea concreta y como resultado de ella retorna un único valor como respuesta. Las funciones se pueden llamar desde cualquier parte de un programa,pero se debe tener en cuenta que su declaración tiene que hacerse antes de que sea llamado por otro módulo, por ello, es conveniente realizar la declaración de los prototipos al inicio del programa, antes de la función main.

Al invocarse una función, se ejecutan las sentencias y retorna un valor en el nombre de la función. Es decir, que conceptualmente el nombre de la función tiene asignado un espacio de memoria con capacidad para representar un dato. El dato que representa debe ser compatible con el tipo de datos asociado al nombre de la función

Por ejemplo, es posible representar una función matemática mediante una función de programación.

Sea G(x,y)=x + y, entonces para G(3,-5)=-2. Es decir si la función g recibe los parámetros 3 y -5 retornaría como resultado -2.

Declaración o Prototipo

```
tipo_de_retornonombre_Funcion(lista_de_parámetros);
```

En la declaración del prototipo se puede definir en la lista de parámetros sólo el tipo de datos, sin incluir un identificador para el nombre de cada parámetro.

Ejemplo de declaración de prototipo de función G

```
int G(int, int);
```

Definición:Estructura

```
tipo_de_retornonombre_Funcion (lista_de_parámetros)
{
  /*cuerpo de la función*/
return (expresión);
}
```

Ejemplo de definición de función G

```
int G(int num1, int num2)
{ int resultado;
   resultado= num1+num2;
   return resultado;
}
```

Los aspectos más sobresalientes en el diseño de una función son:

- Tipo de retorno. Es el tipo de dato que devuelve la función C y aparece antes del nombre de la función. En C, cada función devuelve valores de un único tipo. Si no se especifica ningún tipo, se asume un resultado entero. El tipo de retorno puede ser cualquier tipo de dato excepto una función, una cadena de caracteres (string) o un arreglo (array), y debe coincidir siempre con el Valor devuelto por la función.
- Nombre Función. Identificador o nombre de la función.
- Lista de parámetros. Es una lista de parámetros tipificados (con tipos). Cada declaración de parámetro indica tanto el tipo del mismo como su identificador. Se utiliza el siguiente formato: `tipo1 parámetro1, tipo2 parámetro2,...`
- Cuerpo de la función. Se encierra entre llaves de apertura ({) y cierre (}).
- No se pueden declarar funciones anidadas, es decir, no se puede declarar ni definir una función dentro del cuerpo de otra función.
- Declaración local. Las constantes, tipos de datos y variables declaradas dentro de la función son locales a la misma y no perduran fuera de ella.
- Valor devuelto por la función. Una función devuelve un único valor. Este valor debe coincidir con el Tipo de retorno. El resultado se muestra con una sentencia `return` que generalmente se escribe al final de la función. En el ejemplo: `return(expresión);`

Las funciones pueden ser invocadas desde sentencias de asignación, sentencias de salida o como valores que conformen una expresión.

Para llamar a una función con parámetros es importante respetar el orden y el tipo de los parámetros que ella recibe. El primer valor pasado corresponde al primer parámetro de entrada; el segundo valor, al segundo parámetro; y así sucesivamente si hubiera más.

Ejemplo 1:
A continuación se presenta un programa ejemplo donde se utilizan dos funciones definidas por el usuario, una para el ingreso de un entero y otra para la suma de dos enteros.

```
1.      #include<stdio.h>

2.      /*Prototipos*/
3.      int ingreso(void);
4.      int suma(int,int);

5.      /*Función principal*/
6.      Intmain (){
7.      Inta,b,s;
8.      a=ingreso();
9.      b=ingreso();
10.     s=suma(a,b);
11.     printf("Resultado de la suma: %d", s);
12.     return 0;
13.     }
14.     /*Definición de funciones*/
15.     int ingreso(void){
16.     intnum;
17.     printf("Ingrese un número entero: ");
18.     scanf("%d",&num);
19.     returnnum;
20.     }

21.     intsuma(int x, int y){
22.     return(x+y);
23.     }
```

1.4.2Procedimientos definidos por el programador

Un procedimiento es un grupo de sentencias que realizan una tarea concreta y como resultado de ella puede modificar variables, presentar información en pantalla, generar nuevos valores, etc.

Para invocar un procedimiento, es decir, para hacer que se ejecute, basta con escribir su nombre en el cuerpo de otro procedimiento o en el programa principal. Al igual que con las funciones, la declaración de cada procedimiento tiene que hacerse antes de que sea llamado por otro módulo, por ello es conveniente realizar la declaración de los prototipos al inicio del programa.

Los programas escritos en C se componeníntegramente de funciones, no existen los procedimientos en C, por lo que sesimula la definición de un procedimiento utilizando como tipo de retorno el tipo de datos`void`.

Declaración o Prototipo

```
Voidnombre_Procedimiento(lista_de_parámetros[1]);
```

Estructura

```
Voidnombre_Procedimiento(lista_de_parámetros)
{
   /*cuerpo del Procedimiento*/
}
```

[1] En la declaración es posible tipar los parámetros indicando solo el tipo y no su nombre.

Ejemplo

```
Voidtitulo ()
{ printf("\nPROGRAMACION\n");
}
```

Los aspectos más sobresalientes en el diseño de un procedimiento son:

- Tipo de retorno. Los procedimientos no retornan resultado en su nombre, y como en C los módulos son funciones, se simula el comportamiento de los procedimientos declarando funciones de tipo `void`.
- Nombre Procedimiento. Identificador o nombre del procedimiento.
- Lista de parámetros. Es una lista de parámetros tipificados (con tipos) que utilizan el formato siguiente: `tipo1 parámetro1, tipo2 parámetro2,...`
- Cuerpo del procedimiento. Se encierra entre llaves de apertura ({) y cierre (}).
- No se pueden declarar procedimientos anidados.
- Declaración local. Las constantes, tipos de datos y variables declaradas dentro del procedimiento son locales al mismo y no perduran fuera de él.
- Valor devuelto por el procedimiento. Los procedimientos devuelven cero, uno o más valores y lo hacen en los parámetros.

Ejemplo 2: Procedimiento

```
1.      #include<stdio.h>
2.      /*Prototipos*/
3.      Voidtitulo ();
4.
5.      Intmain(){
6.      printf("Se invoca el procedimiento titulo\n");
7.      titulo(); /**Invocación al procedimiento**/
8.      printf("\nSegunda llamada\n");
9.      titulo(); /**se vuelve a invocar**/
10.     return 0;
11.     }
12.
13.     /*Definición del procedimiento*/
14.     Voidtitulo()
15.     {
16.     printf("\nPROGRAMACION\n");
17.     }
```

1.4.3Identificador o nombre de un módulo

El nombre de un módulo comienza con una letra o un subrayado (_) y puede contener tantas letras, números o subrayados como desee. El compilador ignora, sin embargo, a partir de una cantidad dada, que depende del compilador que se esté utilizando. C es sensible a mayúsculas, lo que significa que las letras mayúsculas y minúsculas son distintas a efectos del nombre dela función. Generalmente en la comunidad

de programadores de C, las funciones que comienzan con subrayado (_) son funciones de sistema o funciones definidas en librerías estándar.

1.4.4Declaración o Prototipos

La declaración de una función se denomina prototipo. Los prototipos de un módulo sólo contienen la cabecera del módulo y terminan con un punto y coma.

Específicamente un prototipo consta de los siguientes elementos: nombre del módulo, una lista deargumentos, también llamados parámetros, encerrados entre paréntesis y un punto y coma. Los prototipos de los módulos llamados en un programa se incluyen en la cabecera del programa, antes de la función main, para que sean reconocidos en todo el programa.

El compilador utiliza los prototipos para validar que el número y tipos de datos de los argumentos reales de la llamada a la función sean los mismos que el número y tipo de argumentos declarados en la función. Si se detecta una inconsistencia, se visualiza un mensaje de error.

2Ámbito de identificadores

Cada identificador tiene un campo de acción (ámbito), sólo dentro de éste campo de acción es posible utilizarlo. El ámbito es la zona de un programa en la que es visible una variable.

Los ejemplos más claros son las variables, que pueden ser globales o locales.

Existen cuatro tipos de ámbitos: **programa**, **archivo fuente**, **función** y **bloque**. En este curso se trabajará con los tres primeros ámbitos. Se puede designar una variable para que esté asociada a uno de estos ámbitos. Tal variable es invisible fuera de su ámbito y sólo se puede acceder a ella en su ámbito.

Por otro lado, existen modificadores de tipo o clases de almacenamiento que permiten modificar el ámbito y la permanencia de una variable dentro de un programa. Estos modificadores son cuatro: **automático**, **externo**, **estático** y **registro**, que se corresponden con las palabras reservadas **`auto`**, **`extern`**, **`static`**y**`register`**, respectivamente. En este curso se trabajará sólo con el modificador `auto`.[2]

`auto`(automática o dinámica)

Una variable se considera automática porque cuando se accede a la función se le asigna espacio en la memoria automáticamente y se libera dicho espacio tan pronto finaliza la ejecución de la función.

La utilización de la palabra reservada auto es opcional, ya que es el modificador por defecto y normalmente no se utiliza.Por ejemplo:
`autoint i;` esequivalente a `int i;`

extern (externa)

Se usa cuando una variable que se creó (se definió) en otro módulo (anterior a la definición de la variable o ubicado en un archivo fuente diferente) y se quiere usar en el actual.

El compilador queda advertido de que la variable ya existe en otro módulo, por lo que no tiene que crearla, sino simplemente usarla.

A este tipo de proceso se le llama declaración de tipo de variable. Una diferencia muy importante entre una definición y una declaración es que en la declaración no se reserva espacio en la memoria para la variable y en la definición si se crea.

[2] Para ampliar la lectura sobre Modificadores de tipo consultar a: Programación en C. Metodología, algoritmos y estructuras de datos. Capítulo7 Funciones. Inciso 7.7

`static`(estática)

Cuando a una variable local se le antepone el modificador `static` pasa de ser dinámica, a ser estática. Su valor es recordado incluso si la función donde está definida finaliza su ejecución y se la vuelve a invocar más tarde.

Si se antepone el modificador `static`a una variable global, definida fuera de una función, se modifica su alcance: pasa de tener alcance global a todos los archivos del programa a ser sólo accesible por las funciones del archivo en el que se la crea.

register(registro)

Indica al compilador que almacene la variable en un registro de la máquina, que es el lugar más eficiente para guardar las variables. Esto se hace porque el trabajo con los registros del procesador es mucho más rápido que el trabajo con la memoria central.

Hay dos detalles importantes:normalmente no hay muchos registros libres en el procesador y muchos compiladores realizan trabajos de optimización, que son modificaciones en el código que generan, para hacerlo más eficiente, aun así suele usarse en situaciones especiales que requieran de mucha eficiencia en el código.

Sólo se puede aplicar a variables locales y a los parámetros formales de una función. Son ideales para el control de bucles.

2.1Variables locales y globales

Cada función puede definir sus propias variables locales definiéndolas en su cuerpo. C permite, además, definir variables fuera del cuerpo de cualquier función: son las variables globales.

2.1.1Variables locales

Las variables locales son aquellas que se declaran tanto dentro dela función principal (`main`) como en las demás funciones, su alcance está limitado solamente ala función en el cual está definida. Por ejemplo, si el programa principal invoca una función “AA” cada variable local definida en una “AA” comienza a existir sólo cuando se llama esa función y desaparece cuando el control regresa al programa principal.

Este programa, por ejemplo, declara dos variables locales para calcular la sumatoria: $\sum_{i=a}^{b}$

Ejemplo 3:

```
1.  #include<stdio.h>
2.  /*Prototipos*/
3.  int sumatoria(int, int);
4.        intmain()
5.        {
6.        Int i; /* Variable local a main */
7.        for(i=1; i<=10; i++)
8.        printf("Sumatoria de los %d primeros numeros naturales: %d\n", i, sumatoria(1, i));
9.        return 0;
10.       }
11.       /*Definición*/
12.       Intsumatoria(int a, int b)
13.       {
14.       Int i, s; /* Variables locales a sumatoria */
15.       s = 0;
```

```
16.     for (i=a; i<=b; i++)
17.     s += i;
18.     return s;
19.     }
```

En el ejemplo se puede observar que la variable local de la función `sumatoria()` con identificador `i` nada tiene que ver con la variable del mismo nombre que es local a la función `main()`.

Las variables locales `i` y `s` de `sumatoria` sólo "viven" durante las llamadas a `sumatoria`.

La zona en la que es visible una variable es su ámbito. Las variables locales sólo son visibles en el cuerpo de la función en la que se declaran; ése es su ámbito.

Se debe tener en cuenta que cualquier variable declarada dentro de una función se considera como una variable automática (`auto`) a menos que se utilice algún modificador de tipo.

2.1.2Variables globales

Las variables globales están definidas fuera de cualquier función y pueden ser utilizadas por cualquier parte del programa. Es decir, si se define una variable al principio del programa, cualquier módulo que forme parte de éste podrá utilizarla simplemente haciendo uso de su nombre.

Es importante recordar que sólo se puede inicializar una variable global en su definición. El valor inicial que se le asigne a la variable global debe expresarse como una constante y no como una expresión. En caso de no asignar un valor inicial a la variable global, automáticamente se le asigna el valor cero (0). De esta manera, las variables globales siempre cuentan con un valor inicial.

Es posible darle el mismo nombre a una variable local y a una global en el mismo programa. En este caso el módulo no puede utilizar la variable global ya que le da preferencia a las locales sobre las globales.

El siguiente ejemplo utiliza una variable local y una variable global.

Ejemplo 4:

```
1.      #include<stdio.h>
2.      int i = 1; /* Variable global i */
3.      int doble(void); /*prototipo*/
4.      intmain(void)
5.      {
6.      Int i; /* Variable local i */
7.      for (i=0; i<5; i++) /*Referencias a la variable local i */
8.          printf ("%d\n", doble()); /*Cuidado, el valor mostrado
            corresponde a la variable global i*/
9.      return 0;
10.     }
11.     int doble(void)
12.     {
13.     i *= 2;    /* Referencia a la variable global i*/
```

```
14.     return i; /* Referencia a la variable global i */
15.     }
```

Se puede observar la pérdida de legibilidad que supone el uso del identificador i en diferentes puntos del programa ya que siempre está la pregunta latente, si la variable que se utiliza, en el ejemplo anterior, la variable i, corresponde a la variable local o global. No se aconseja el uso generalizado de variables globales,esto aumenta la probabilidad de cometer errores al intentar acceder o modificar una variable. Por otro lado, el uso de variables globales dificulta la re-utilización de las funciones en otros programas.

2.2Transferencia de información a y desde procedimientos

La comunicación entre funciones y el programa principal o bien entre las mismas funciones se lleva a cabo entre variables globales, parámetros por valor y parámetros por referencia.

2.2.1Parámetros formales y actuales

Parámetros formales

Estos se incluyen en la declaración de una función. El término "formal" hace mención a la forma que toman dichos parámetros, y el referirse a la forma indica la cantidad de parámetros, el tipo de datos asociados a cada uno de ellos, y el orden en que se presentan dichos parámetros.

Por ejemplo: En la función`intsumatoria(int a, int b)`, "`a`" y "`b`" son parámetros formales de tipo entero de la función`sumatoria`. Estos sirven para contener los valores de los parámetros actuales cuando se invoca el procedimiento.

Parámetros actuales

Estos se incluyen en las sentencias de llamada a una función. El término "actual" se refiere a los valores o a las variables que intervienen como parámetros en cada llamada a la función. Estos parámetros actuales deben coincidir en cantidad, tipo de datos y orden con los parámetros formales.

Por ejemplo: En la función `resultado = sumatoria(1, i)`,`1` e `i`sonparámetros actuales de tipo entero de la función `sumatoria`. Los parámetros actuales`1`e`i`tienen que tener los valores que se pasan a la función `sumatoria`.

2.2.2Parámetros pasados por valor y por referencia

Parámetros por valor

El paso de parámetros por valor consiste en copiar el contenido del parámetro actual que se quiere pasar en una variable dentro del ámbito local de la subrutina, consiste en copiar el contenido de la memoria del argumento que se quiere pasar a otra dirección de memoria, correspondiente al argumento dentro del ámbito de dicha subrutina. Se tendrán dos valores duplicados e independientes, con lo que la modificación de uno no afecta al otro.

El parámetro actual pasado puede ser una variable, una contaste, una expresión aritmética, lógica o una llamada a otra función. Si el parámetro actual es una variable y la función que lo recibe lo modifica, la variable original no se verá afectada.

Parámetros por referencia

El paso de parámetros por referencia consiste en proporcionar al módulo invocado, la dirección de memoria donde se aloja del dato. En este caso se tiene un único valor referenciado (o apuntado) desde dos

puntos diferentes, por ejemplo el programa principal y la subrutina a la que se le pasa el argumento, por lo que cualquier acción sobre el parámetro se realiza sobre la misma posición de memoria. Es decir en el parámetro se recibe la dirección de memoria de la variable original. Si la función que lo recibe lo modifica, la variable original también se ve afectada. En C los parámetros por referencia se realizan con punteros. Se utiliza el operador * para indicar indirección y el operador & para indicar la dirección.

Un ejemplo de uso de parámetros por referencia es,

```
/* Definición (parámetro formal y por referencia) */
IntmiFuncion (char *unaVariable)
{
return (*unaVariable+=5);
}
...
/*Invocación a la función (parámetro actual y por referencia)*/
resultado = miFuncion(&otraVariable);
```

Diferencias entre paso de variables por valor y por referencia

Parámetropasado por valor	**Parámetro pasado por referencia**
Los parámetros pasados por valor **reciben copias** de los valores de los argumentos que se le pasa.	Los parámetros formales pasados por referencia son declarados con *, **reciben la dirección** de los argumentos pasados; los parámetros actuales deben ser precedidospor el operador &. Se exceptúan de esta regla los arrays;
La asignación a un parámetro pasado por valor **nunca cambia el valor del argumento original.**	La asignación a un parámetro por referencia (punteros) **cambia el valor del argumento original**.

Se puede definir una función sin parámetros dejando la palabra `void` como contenido de la lista de parámetros. Para invocarlase debe añadir los paréntesis a la derecha del identificador, aunque no tenga parámetros.