Lenguaje C

Universidad Nacional de Rosario
Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Franco Di Pietro
dipietro@fceia.unr.edu.ar

2018

1. Introducción

La resolución de problemas mediante el uso de computadoras consta de una serie de etapas. En primera instancia, se debe analizar y comprender la totalidad del problema para poder abordarlo correctamente, una falla en esta instancia causará indefectiblemente resultados erróneos. Como resultado de esto, se determinará cuáles son los datos del problema, cuáles los resultados que se desean obtener, y se empezará a diagramar el camino para obtenerlos partiendo de dichos datos. De este modo, se obtendrá un *algoritmo* solución del problema en cuestión, que se escribirá en pseudo-código *pseudo-código* [1].

Sin embargo, un algoritmo no puede ser comprendido por una computadora, pues la misma no es capaz de interpretar su sintaxis. Para ello, este debe ser *codificado* en un lenguaje de programación, obteniendo así un *programa*, el cual podrá ser ejecutado por la computadora. Surge entonces el interrogante de cuál es el lenguaje que debemos utilizar para realizar dicha codificación, y esta pregunta puede tener más de una respuesta.

Del mismo modo que una misma idea puede transmitirse en diferentes idiomas, existe una multiplicidad de lenguajes en los cuales se puede codificar un mismo algoritmo [2, 3], los hay compilados, interpretados, orientados a objetos, etc. Volviendo al caso de los idiomas, veamos el siguiente ejemplo:

```
Hola, hoy vamos a estudiar el lenguaje C.
Hello, today we are studying the C language.
Bonjour, aujourd'hui on va ètudier le langage C.
你们好，今天我们要学习语言C.
```

Estás oraciones dicen lo mismo en diferentes idiomas, es decir todas ellas solucionan el mismo problema: transmitir una idea, que en este caso es saludar y comunicar qué es lo que haremos el día de hoy. La elección de un idioma dependerá en este caso, entre otras cosas, de la lengua que maneje quien quiera transmitir la idea, así como a quién esté dirigido ese mensaje. Análogamente, la elección de uno u otro lenguaje de programación dependerá de varios factores, como ser la naturaleza del problema a resolver, la plataforma que ejecutará el programa final y lo familiarizado que el programador esté con algún lenguaje en particular, entre otros.

En el caso de la asignatura *Informática*, para las carreras de ingeniería dictadas en la FCEIA, se utilizará el lenguaje C. Este lenguaje originalmente desarrollado por Dennis Ritchie a comienzos de la década del 70, es apreciado por la eficiencia del código que produce y es el lenguaje de programación más popular para crear software de sistemas, aunque también se lo utiliza para crear aplicaciones [4].

2. Codificación en lenguaje C

El lenguaje C trabaja utilizando funciones, concepto que será abordado con mayor detalle más adelante. Un programa en C consiste, en su forma más básica, sólo de una función principal cuyo nombre es `main`. Esta función principal es análoga al algoritmo principal diseñado para resolver un problema. El `main` es el punto de partida de ejecución de todo programa.

A continuación se muestra el código de un programa en C.

```
1 int main()
2 {
3   // acciones
4
5   return 0;
6 }
```

En la primera linea se observa la declaración de la función principal. En esta declaración se pueden distinguir tres partes: el tipo de retorno de la función, que en este caso es de tipo entero (`int`); el nombre de la función (`main`); y finalmente la lista de argumentos que recibe dicha función entre paréntesis (en este caso, no tiene argumentos). Luego, el cuerpo del `main` está delimitado por llaves, en donde deberán declararse variables y codificarse las acciones a llevar a cabo. Finalmente, la sentencia `return 0;` devuelve el control al sistema operativo que esté ejecutando el programa en cuestión. Aquí podemos ver que esta sentencia, al igual que toda sentencia de sólo una linea, debe concluir con el carácter punto y coma (`;`).

Tipos de datos

A la hora de resolver un problema, surge la necesidad de utilizar variables de diverso tipo. Los tipos de datos presentes de forma nativa en C son los siguientes:

- `int`: entero
- `float`: real en simple precisión
- `char`: carácter
- `double`: real en doble precisión (mayor capacidad de representación)

Existen además variantes de estos tipos de datos nativos que se obtienen utilizando *modificadores de tipo de dato*, como ser: `short`, `long`, `signed` y `unsigned`.

Operadores

Las operaciones válidas en *pseudo-código* tienen su equivalencia en cada lenguaje de programación. En el caso de C, el operador de asignación no es una flecha sino el signo igual. Por ejemplo para asignar el valor 1 a la variable `x` se utiliza `x = 1;`. A continuación se listan los operadores más comunes.

Operadores matemáticos	
suma	`+`
resta	`-`
multiplicación	`*`
división	`/`
módulo	`%`

Operadores relacionales	
menor	`<`
menor o igual	`<=`
mayor	`>`
mayor o igual	`>=`
igual	`==`
distinto	`!=`

Operadores lógicos	
and	`&&`
or	`||`
not	`!`

El operador matemático módulo obtiene el resto de la división entera entre sus dos operando, por ejemplo: `5% 2`, dará como resultado `1`.

Además de los operadores matemáticos mostrados, también existen los operadores *incremento* y *decremento*. El operador incremento (++) suma 1 a su operando, en tanto que el operador decremento (−−) le resta 1. Ambos pueden ser usados como prefijo(`++x`) y como sufijo(`x++`), la diferencia entre ambos usos escapa a los objetivos del curso, por lo cual simplemente se dirá que lo utilizaremos como sufijo, usualmente para incrementar o decrementar índices.

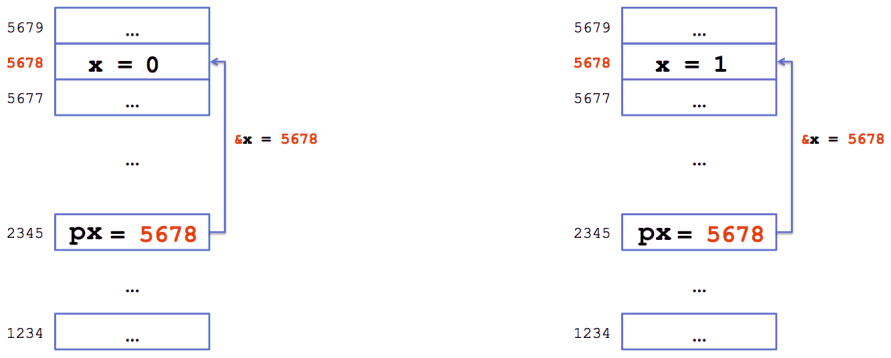
Una explicación en mayor detalle tanto de tipos de datos como de operadores puede ser encontrada en [5, Cap 2].

Algunos conceptos importantes

Antes de seguir avanzando, es conveniente entender algunos conceptos que se utilizan en el lenguaje C. Como lo son el de los operadores de *dirección*(&) e *indirección*(*) y el concepto de *puntero*. Entender esto posibilitará una mejor comprensión del lenguaje y facilitará su utilización considerablemente. Para ello, partiremos de las siguientes lineas de un programa:

```
1 int x;
2 x = 0;
3 int *px;
4 px = &x;
5 *px = 1;
```

Supongamos que se tiene una variable de tipo entero cuyo nombre es `x` (linea 1). Inicialmente se le asigna a dicha variable el valor 0 (linea 2). Como resultado de aplicar el operador dirección a esta variable (`&x`) se obtiene la dirección en memoria de la misma. Esta dirección puede ser almacenada en un tipo de variable que recibe el nombre de *puntero*. Un puntero es una variable que apunta a otra (es decir que contiene su dirección). Por lo tanto, se dice que un puntero es una *referencia* a otra variable. Como toda variable, esta debe ser declarada, y para hacerlo se debe definir el tipo de variable a la que apuntará y el nombre del puntero precedido por el carácter *. En la linea 3 puede verse la declaración del puntero de nombre `px`, que apuntará a una variable de tipo entero. A continuación, en la linea 4, se le asigna al puntero `px` la dirección de la variable `x`. Finalmente, utilizando el operador de indirección (*), se puede modificar el contenido de la variable apuntada por un puntero. En la linea 5, se está diciendo que en la variable apuntada por `px`, es decir en `x`, se debe guardar el valor 1. Esta modificación del valor en la variable `x` utilizando el puntero `px` puede verse ilustrada en la figura 1, donde el número que aparece a la izquierda de cada celda de memoria representa su dirección.



(a) Antes de la linea 5 (b) Luego de la linea 5

Figura 1: Ejemplo del estado de la mermoria principal en el uso de punteros.

Aquí se ha utilizado el símbolo * para dos usos distintos que no deben confundirse. En la linea 3 se lo utiliza para la declaración de un puntero, es decir para decir que `px` será una referencia. Por otro lado, en la linea 5, se lo utiliza como operador de indirección para modificar

el valor de la variable referenciada por el puntero, en este caso el resultado sería equivalente a escribir `x = 1`.

2.1. Entrada/salida: leer y escribir

Para que un programa pueda interactuar con el usuario, debe valerse de capacidades de entrada y salida. Recordando que C es un lenguaje que trabaja con funciones, para dotar a un programa de tal capacidad se deben utilizar funciones específicas para estas tareas. Estas funciones están incluidas en la biblioteca `stdio` (del inglés *Standard Input Output*). Para incluir esta biblioteca se debe colocar la siguiente sentencia al comienzo del programa (fuera del main).

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

Salida: escribir

La función encargada de implementar el *escribir* es `printf`. Esta función recibe al menos un parámetro, que es una cadena de caracteres a ser mostrada en pantalla (delimitada por comillas). Por ejemplo, el programa para mostrar un texto en pantalla es el siguiente:

```
1 #include <stdio.h>
2
3 int main()
4 {
5   printf("Hola mundo!");
6   return 0;
7 }
```

Y como resultado de su ejecución se obtendría:

```
Hola mundo!
```

Ahora bien, para mostrar el valor de una variable se debe especificar en primera instancia el tipo de variable que se mostrará. Para ellos se utilizan *especificadores de formato* que se escriben en la cadena de caracteres de la función. Estos son:

- `%d` para mostrar un entero
- `%f` para mostrar un real
- `%c` para mostrar un carácter
- `%s` para mostrar una cadena de caracteres

Además puede utilizarse `%g` para mostrar reales de manera más prolija (sin cifras no significativas o utilizando notación científica según convenga).

Por ejemplo si se quiere mostrar el valor de la variable entera `x` cuyo valor es 5 y el de la variable real `y` que vale 2.3, se debe escribir:

```
1 printf("El valor de x es: %d\nEl valor de y es: %g", x, y);
```

Aquí se ve que además del texto a mostrar, se ha incluido el comando `\n`. Esto quiere decir que debe cambiar de linea, sería el equivalente a un *"enter"* en un procesador de texto. (también pueden incluirse tabulaciones con `\t`). Así, el resultado obtenido es:

```
El valor de x es: 5
El valor de y es: 2.3
```

Entrada: leer

La función encargada de implementar el *leer* es `scanf`. Esta función recibe al menos dos parámetros, una cadena de caracteres (delimitada por comillas) en donde se utilizan especificadores de formato para indicar el tipo de valor a ser leído, y luego, una referencia a la variable donde se desea guardar el valor ingresado por teclado. Para obtener una referencia a una variable en particular basta con utilizar el operador de dirección &. Por ejemplo, el programa para ingresar por teclado un número entero en la variable `x` es el siguiente:

```
#include <stdio.h>

int main()
{
  int x;
  scanf("%d", &x);
  return 0;
}
```

Por otro lado, a la hora de leer un carácter utilizando la función `scanf`, se debe tener la precaución de dejar un espacio en blanco en la cadena de caracteres antes de poner el especificador de formato de tipo carácter, el resultado sería el siguiente:

```
char var;
scanf(" %c", &var);
```

Eso se hace para especificar a la función `scanf` que se omita la lectura de espacios en blanco, evitando así que dicha función lea como carácter ingresado un *"enter"* que haya sido ingresado anteriormente.

2.2. Estructuras de control

A continuación, se describirá la codificación de las estructuras de control utilizadas en la asignatura. Una explicación en mayor profundidad de cada una puede encontrarse en [5, Cap 3].

2.2.1. Selección simple: si

La proposición `if-else` se utiliza para expresar selecciones simples. Formalmente la sintaxis es la siguiente:

```
if(condición)
{
  // acciones verdadero
}
else
{
  // acciones falso
}
```

donde la parte del `else` es optativa. La *condición* se evalúa, si es verdadera (esto es, si la *condición* tiene un valor distinto de cero), se ejecutan las acciones correspondientes, caso contrario se ejecutan las acciones correspondientes al valor falso.

A continuación veremos un ejemplo del uso de esta estructura.

```
#include <stdio.h>

int main()
{
  int nro;
  scanf("%d", &nro);
  if(nro>0)
```

```
8     {
9       printf("Nro positivo\n");
10    }
11    else
12    {
13      printf("Nro negativo o cero\n");
14    }
15    return 0;
16  }
```

En este ejemplo, vemos que tanto el cuerpo del `if` como del `else` consisten de sólo una linea. En estos casos, puede omitirse el uso de las llaves, y delimitar el alcance de una sentencia utilizando un punto y coma al final de dicha única acción, resultando:

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int nro;
6    scanf("%d", &nro);
7    if(nro>0) printf("Nro positivo\n");
8    else printf("Nro negativo o cero\n");
9    return 0;
10 }
```

2.2.2. Selección múltiple: según sea

La proposición `switch` es una decisión múltiple que prueba si una *expresión* coincide con una cantidad de valores constantes, y traslada el control adecuadamente. La sintaxis de esta estructura de control es la siguiente.

```
1  switch(expresión)
2  {
3    case expresión-constante:
4          //acciones
5          break;
6    case expresión-constante:
7          //acciones
8          break;
9    default:
10         //acciones
11 }
```

Cada `case` se etiqueta con un valor constante. Si un `case` coincide con el valor de la expresión, la ejecución comienza allí. Todas las expresiones `case` deben ser diferentes . El etiquetado como `default` se ejecuta si ninguno de los otros se satisface. El `default` es optativo, si no está y ninguno de los casos coincide, no se toma acción alguna. La sentencia `break` provoca una salida inmediata del `switch`. Dado que los `case` funcionan como etiquetas, después que se ejecuta el código para uno, la ejecución pasa al siguiente a menos que se explicite que esto no debe suceder a través de la sentencia `break`.

A continuación veremos un ejemplo del uso de esta estructura.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int nro;
6    scanf("%d", &nro);
```

```
7     switch(nro)
8     {
9       case 1: case 3: case 5: case 7: case 8: case 10: case 12:
10        printf("El mes ingresado tiene 31 dias\n");
11        break;
12      case 4: case 6: case 9: case 11:
13        printf("El mes ingresado tiene 30 dias\n");
14        break;
15      case 2:
16        printf("El mes ingresado tiene 28/29 dias\n");
17        break;
18      default:
19        printf("El nro ingresado no corresponde a un mes\n");
20    }
21    return 0;
22  }
```

En el ejemplo la variable de control del `switch` es el entero `nro`. Los valores que siguen a cada sentencia `case` son valores posibles de dicha variable. Por lo tanto si se utilizara una variable de tipo carácter, los valores que siguen a cada caso deben ser caracteres válidos, es decir que deben ir entre apóstrofos, como por ejemplo: `case 'A':`.

2.2.3. Estructuras de repetición

Repetir para

En en lenguaje C, la implementación del bucle de repetición *Para* es llevada a cabo por la siguiente instrucción:

```
1  for(inicialización; condición; incremento)
2  {
3    //acciones
4  }
```

donde en el campo *inicialización* se debe escribir una expresión que asigne a la/s variable/s de control del bucle un valor inicial claramente definido. Luego, la *condición* será evaluada al comienzo de cada iteración del bucle para saber si deben ejecutarse la acciones, es decir, se ejecutará siempre y cuando *condición* = *verdadero*. Evidentemente, en esta condición deberá/n estar implicada/s la/s variable/s de control. Finalmente, en el *incremento* se debe incluir una expresión que modifique de algún modo la/s variable/s de control del bucle para poder salir del mismo en algún momento. Veremos esto con un ejemplo simple.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int i;
6    for(i = 1; i<11; i++)
7    {
8      printf("%d\n", i);
9    }
10   return 0;
11 }
```

Como resultado de ejecutar este programa, se mostrarán por pantalla los números enteros del 1 al 10 en lineas consecutivas. Se ve del ejemplo que la variable de control es el entero `i` que se inicializa en 1, luego el bucle se repetirá siempre y cuando el valor de dicha variable sea menor a 11 (es decir hasta que tome el valor 10), finalmente se establece que el incremento sea de uno (recordando que `i++` es equivalente a escribir `i = i + 1`).

Repetir mientras

La sintaxis para implementar un *Repetir Mientras* es la siguiente:

```
while(condición)
{
  //acciones
}
```

Aquí la *condición* se evalúa al comienzo de cada ciclo del bucle, si esta es *verdadera*, se ejecutarán las acciones en el cuerpo del `while`. Es importante recordar que la/s variable/s involucrada/s en la condición debe/n ser modificada/s dentro del cuerpo de la estructura de repetición, para evitar entrar en un bucle infinito. Veremos a continuación, un ejemplo ilustrativo de su uso.

```
#include <stdio.h>

int main()
{
  int nro;
  scanf("%d", &nro);
  while (nro!=0)
  {
    scanf("%d", &nro);
  }
  return 0;
}
```

Aquí se ingresan valores para la variable entera `nro` mientras esta tenga un valor distinto de cero. Para lograrlo, dicha variable debe ser leída por primera vez fuera del bucle ya que la condición se evalúa al comienzo del mismo, y las variables involucradas en la *condición* deben estar correctamente declaradas y definidas.

Repetir hasta

Esta estructura de repetición no tiene una equivalencia en C tal y como la como la implementamos en *pseudo-código*, sin embargo si existe la siguiente estructura:

```
do
{
  //acciones
} while(condición);
```

En este caso, las acciones se ejecutan al menos una vez, y luego la *condición* se evalúa al final de cada ciclo del bucle, si esta es *verdadera*, se volverá al comiendo del `do`, ejecutándose nuevamente las acciones.

A diferencia del *repetir hasta*, donde se repite **hasta** que la *condición* sea *verdadera* (o lo que es lo mismo, **mientras** que sea *falsa*), en el bucle `do-while` se repite **mientras** dicha *condición* sea *verdadera* .

Repetiremos ahora el ejemplo del caso anterior, visto en con el ciclo `while`, pero utilizando ahora la estructura `do-while`.

```
#include <stdio.h>

int main()
{
  int nro;
  do
  {
    scanf("%d", &nro);
```

```
9     }while(nro!=0);
10    return 0;
11  }
```

El funcionamiento es idéntico al caso anterior, pero como aquí la condición se evalúa al final del ciclo, no hace falta leer la variable de control fuera del bucle, ya que este se ejecuta al menos una vez.

2.3. Estructuras de datos

Un dato estructurado es un conjunto de datos agrupados bajo un mismo nombre y lógicamente vinculados de manera tal que representen un comportamiento específico [6].

En el lenguaje C existen varios tipos de estructuras de datos disponibles, se analizarán a continuación los más comunes.

2.3.1. Arreglos

Los arreglos se definen en lenguaje C de manera semejante a como se hace en *pseudo-código*. Para ello se debe especificar el tipo de datos que contendrá el arreglo en cuestión, darle un nombre y a continuación entre corchetes su dimensión. Ejemplos válidos son:

```
1  int notas[4];
2  float alturas[5];
3  char nombre[10];
4  int matriz[3][2];
```

En la linea 1 se declara una arreglo unidimensional de 4 elementos de tipo entero. En la linea 3, vemos la declaración de un arreglo de 10 caracteres, o lo que es lo mismo un *string* de 10 caracteres (este tipo particular de cadenas será explicado en detalle más adelante). Finalmente, en la linea 4 podemos ver la declaración de un arreglo bidimensional de enteros de nombre `matriz`.

Hay que tener presente que, a diferencia de lo que sucede al trabajar con *pseudo-código* , en C los arreglos siempre comienzan en la posición 0, y la última posición de un arreglo de longitud N será la $N - 1$.

Para acceder a un elemento de un arreglo se utiliza su nombre y la posición a la que se quiere acceder entre corchetes, por ejemplo `notas[0]` es el primer elemento del arreglo `notas`.

Al igual que en *pseudo-código* para cargar un arreglo habrá que hacerlo elemento por elemento, posiblemente utilizando bucles `for`. A continuación se muestra un ejemplo de carga de dos arreglos.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    float alturas[5];
6    int matriz[3][2], i, j;
7
8    //carga alturas
9    for(i=0; i<5; i++)
10   {
11     scanf("%f", &alturas[i]);
12   }
13   //carga matriz
14   for(i=0; i<3; i++)
15   {
16     for(j=0; j<2; j++)
17     {
```

```
18          scanf("%d", &matriz[i][j]);
19        }
20      }
21      return 0;
22  }
```

Sin embargo, existe la posibilidad de inicializar arreglos en su declaración de una manera más cómoda:

```
1  float alturas[5] = {1.75, 1.80, 1.71, 1.87, 1.68};
2  int matriz[3][2] = {{3, 7}, {4, 7}, {9, -1}};
```

Es importante recordar que esta sintaxis no es válida para asignar valores a un arreglo en ninguna otra parte de un programa que no sea la propia declaración del arreglo en cuestión.

Un concepto importante respecto a los arreglos, que será de gran importancia al trabajar con subalgoritmos, es que el nombre de un arreglo es una *referencia* al mismo. Partiendo de los arreglos declarados en el ejemplo, vemos que `notas` es una *referencia* al arreglo de 4 enteros de ese nombre.

2.3.2. Cadenas de caracteres: strings

Las cadenas de caracteres son un tipo particular de arreglo, en el cual sus elementos son de tipo carácter (`char` en C). Para declarar una cadena de caracteres, por lo tanto habrá que hacerlo de la misma manera que se lo hace con otros tipos de arreglo. Por ejemplo, se muestra a continuación la declararación de una cadena que contiene el nombre de una persona, donde dicha cadena tenga a lo sumo 25 caracteres.

```
1  char nombre[25];
```

Con este tipo de arreglos, existe la posibilidad de inicializarlos en la propia declaración, asignándole una cadena de caracteres delimitada por comillas. Por ejemplo:

```
1  char nombre[25] = "Juan_Sosa";
```

Recordando lo visto en la sección 2.1, para mostrar el contenido de una variable de tipo cadena, se utiliza el *especificador de formato* `%s`. Así, la instrucción para mostrar el contenido de la variable `nombre` sería:

```
1  printf("%s", nombre);
```

Luego, para leer una cadena de caracteres, se deberá pasar a la función de entrada `scanf` una referencia a dicho arreglo. Y como fue oportunamente explicado, el nombre de un arreglo es una referencia al mismo. Por lo tanto, resultaría:

```
1  scanf("%s", nombre);
```

De donde se ve que, a diferencia de la lectura de variables simples, no se debe utilizar el *operador dirección* `&`.

Por otro lado, para tratar con este tipo de cadenas existen algunas funciones que nos permiten realizar operaciones sobre las mismas, que no podrían ser utilizadas en otro tipo de arreglos. Estas son, entre otras:

- asignación
- comparación

Estas funciones están disponibles en la biblioteca `string.h`, que deberá ser incluida al comienzo del programa en que quieran ser utilizadas.

Asignación de strings

Queda claro que no es posible asignar los valores de un arreglo a otro en una sola instrucción, ya que sólo es posible trabajar con elementos individuales de un arreglo. Para hacerlo, habría que recorrer todos los elementos de un arreglo e ir asignándolos a cada uno de los elementos de un segundo arreglo. Por ejemplo:

```
int main()
{
  int A[3] = {10, 11, 12}, B[3], i;

  //copiado de los elementos de A en los de B
  for(i=0; i<3; i++)
    B[i] = A[i];

  return 0;
}
```

Este programa asignará a cada uno de los elementos del arreglo `B`, los valores almacenados en `A`.

Sin embargo, es posible utilizar la función `strcpy` (por *string copy*), definida en la biblioteca antes mencionada, para copiar el contenido de una cadena de caracteres en otra. Vemos un ejemplo a continuación.

```
#include <string.h>

int main()
{
  char A[20] = "Juan_Sosa", B[20];

  //copiado del string A en B
  strcpy(B, A);
  return 0;
}
```

Comparación de strings

Es posible realizar la comparación entre dos cadenas de caracteres, para determinar por ejemplo si son iguales, o cuál de las dos es mayor (entendíendose que, por ejemplo "AB" es mayor que "AA"). Para lo cual se utiliza la función `strcmp` (por *string compare*). A esta función se le deben dar dos cadenas y devolverá uno de tres valores posibles:

$$\texttt{strcmp(string1, string2)} = \begin{cases} \textit{valor positivo} & \texttt{si string1 > string2} \\ \textit{cero} & \texttt{si string1 = string2} \\ \textit{valor negativo} & \texttt{si string1 < string2} \end{cases}$$

Vemos un ejemplo.

```
#include <string.h>

int main()
{
  char A[3] = "AB", B[3] = "AA";
  int rta;

  //compara  A y B
  rta = strcmp(A, B);
  return 0;
}
```

En este caso, al ser `A` mayor que `B`, el resultado almacenado en `rta` será un número positivo.

2.3.3. Registros

Un registro es una estructura de datos que puede tomar por valor a un conjunto de distintos o iguales tipos de datos. En C, este tipo de estructura se implementa como sigue.

```
struct nommbre_registro
{
  tipo_variable_1 nombre_variable_1;
  tipo_variable_2 nombre_variable_2;
  //...
  tipo_variable_n nombre_variable_n;
};
```

Por ejemplo, un registro utilizado para representar puntos en el plano debería contener dos coordenadas reales `x` e `y`, resultando de la siguiente manera:

```
struct punto_en_el_plano
{
  float x;
  float y;
};
```

Luego, para declarar una variable del tipo registro `punto_en_el_plano` dentro del `main`, se utiliza la siguiente sintaxis, semejante a la declaración de cualquier otra variable.

```
struct punto_en_el_plano pto;
```

Donde la nueva variable `pto` tendrá dos campos reales, estos son `x` e `y`, y para acceder a los mismos se utiliza el nombre de la variable, seguido de un punto y luego el nombre del campo en cuestión. Por ejemplo para asignarles valores a cada campo de la variable `pto` se procedería como sigue.

```
pto.x=1.2;
pto.y=4.3;
```

Más información referida al uso de registros en lenguaje C puede ser encontrada en [5, Cap 6].

2.3.4. Archivos

Para trabajar con archivos en C, es necesaria la declaración de una variable de tipo `FILE`. En particular para manipular un archivo se necesita una referencia a una variable del tipo `FILE`, para lo cual se utilizará el concepto antes visto de *puntero*.

Para abrir un archivo se utiliza la función `fopen`, que recibirá el nombre del archivo con su ruta y el modo en el cual será abierto (`r` para lectura `-por read-`, `w` para escritura `-por write`, `a` para agregado `-por append` o `a+` para lectura y posterior agregado).

Para leer y escribir de un archivo que haya sido previamente abierto, se utilizan las funciones `fscanf` y `fprintf`, que funcionan del mismo modo que las funciones entrada y salida previamente estudiadas, sólo que recibirán además como primer parámetro la referencia al archivo con el que deseamos trabajar. Finalmente, luego de haber realizado todas las operaciones que se deseen sobre un archivo, este debe ser cerrado utilizando la función `fclose` que recibe una referencia al archivo que se desea cerrar. Veamos todo esto a través de un simple ejemplo.

Supongamos que se tiene un archivo de nombre `alumnos.txt` como el siguiente:

```
Juan    Perez          20
Laura   Gonzales       19
Pablo   Rossi          21
```

Luego, el programa para leer y mostrar por pantalla la totalidad de este archivo, y finalmente agregar una nueva linea al mismo, podría ser como sigue.

```
1  #include <stdio.h>
2
3  struct alumno
4  {
5    char nombre[10]; // string de 10 caracteres
6    char apellido[10];
7    int edad;
8  };
9
10 int main()
11 {
12   struct alumno A;
13   FILE *fIN; //fIN es un referencia a un archivo
14
15   fIN=fopen("alumnos.txt","a+");
16   //lectura del archivo - 1ra linea
17   fscanf(fIN,"%s %s %d", A.nombre, A.apellido, &A.edad);
18   //mientras no se llegue al fin de archivo (EOF: end of file) de
       fIN
19   while(!(feof(fIN)))
20   {
21     //mostrado en pantalla
22     printf("%s %s %d\n", A.nombre, A.apellido, A.edad);
23     //lectura del archivo - linea siguiente
24     fscanf(fIN,"%s %s %d", A.nombre, A.apellido, &A.edad);
25   }
26   //agregado de una nueva linea
27   fprintf(fIN,"%s %s %d\n","Esteban", "Lopez", 19);
28   fclose(fIN);
29   return 0;
30 }
```

Analicemos a continuación las lineas 17 y 24. Recordando lo visto sobre el funcionamiento de la función de entrada `scanf`(en este caso `fscanf`) debe recibir referencias a las variables donde guardar los valores leídos. Esto es claro en el caso de la variable entera `edad` en donde para obtener una referencia se utiliza el operador *dirección* `&`. Sin embargo, en las cadenas de caracteres `nombre` y `edad` no se ha utilizado dicho operador. Esto se debe a que, como fue explicado en la sección relativa a arreglos, el nombre de un arreglo es ya una referencia al mismo.

Para un mayor detalle respecto a la manipulación de archivos en C, véase [5, Cap 7].

2.4. Subalgoritmos

Los subalgoritmos dividen tareas grandes de computación en varias más pequeñas, y permiten la posibilidad de construir sobre lo que otros ya han hecho, en lugar de comenzar desde cero. Los subalgoritmos apropiados ocultan los detalles de operación de las partes de programas que no necesitan saber acerca de ellos, así que dan claridad a la totalidad y facilitan la penosa tarea de hacer cambios.

Conceptualmente, se diferencian dos tipos de subalgoritmos según las tareas que desarrollen: los subalgoritmos función y los subalgoritmos subrutina. Sin embargo, en el lenguaje C sólo existen las primeras, pero veremos el modo en el que esto puede ser subsanado.

Una explicación en detalle respecto a la utilización de función puede encontrase en [5, Cap 4].

Funciones

La sintaxis para la declaración de una función es la siguiente:

```
1 tipo_de_retorno nombre_de_la_función(lista de parámetros)
2 {
3   // declaración de variables locales
4   // acciones de la función
5   return (variable_o_expresión_de_retorno);
6 }
```

donde:

- `tipo_de_retorno` es el tipo de dato que devolverá la función(`int`, `float`, `char`, etc).
- `nombre_de_la_función` es el nombre con el cual se declarará a la función, y con el cual se la llamará cuando se necesite.
- `lista de parámetros` es una lista, separada por comas, de variables que la función recibirá como parámetros. Cada elemento de esta lista deberá ser declarado del mismo modo que que se vió para la declaración de variables de un algoritmo principal.
- `return` es la sentencia final de la función, es el mecanismo para devolver el resultado de la función, y está debe devolver un valor que coincida con el tipo de retorno establecido en el encabezado de la función.

Veamos esto a través de un ejemplo sencillo. La siguiente función se encarga de calcular el área de un triángulo:

```
1 float area_triang(float base, float h)
2 {
3   float sup;
4
5   sup = base*h/2;
6   return sup;
7 }
```

Esta función de nombre `area_triang` recibe los parámetros `base` y `h`, ambos de tipo `float`, y devuelve el valor de la variable local `sup`, también de tipo `float`.

Subrutinas

Para poder implementar una subrutina en C se utiliza una función cuyo tipo de retorno es `void`, es decir, una función que no retorna ningún valor (y que por ende, no debe tener sentencia `return`). En el ejemplo anterior los parámetros de la función le fueron pasados *por valor*, es decir que la función sólo recibe una copia del contenido de los argumentos con los que se llame a dicha función. Para poder utilizar parámetros como salida o como entrada/salida, estos se deben pasar *por referencia*. Para hacer esto, se utiliza el concepto de punteros anteriormente visto. Es decir, la función en cuestión recibirá un puntero a la variable que se quiere utilizar *por referencia*.

Volviendo al ejemplo anterior, pero implementándolo a través de una subrutina, obtendríamos lo siguiente:

```
1 void sub_area_triang(float base, float h, float *sup)
2 {
3   *sup = base*h/2;
4 }
```

Donde el * que precede a la variable `sup` en el encabezado indica que se trata de un puntero, es decir de una referencia. Por otro lado, el * en la linea 3 es el operador de `indirección` utilizado para modificar el contenido de la referencia `sup`.

Veremos a continuación un algoritmo principal que haga uso de ambas funciones.

```
1  int main()
2  {
3    float bas=4, alt=6, sup1, sup2;
4
5    sup1 = area_triang(bas, alt);
6    printf("La superficie es: %g\n", sup1);
7    sub_area_triang(bas, alt, &sup2);
8    printf("La superficie es: %g\n", sup2);
9    return 0;
10 }
```

En la linea 5 vemos el llamado a la función `area_triang` que devuelve un valor de tipo `float` que, en este caso, es asignado a la variable `sup1`. Por otro lado, en la linea 7, se ve el llamado a la función `sub_area_triang` (implementación en C de una "subrutina") a la cual se le pasan los dos primeros argumentos por valor y el tercero por referencia, lo cual es evidente por el uso de operador de *dirección* `&`. En este caso, el resultado es guardado en la variable `sup2` que modifica su valor dentro de la función `sub_area_triang`. El resultado en pantalla de ambos `printf` será entonces idéntico.

Arreglos como parámetros

En este punto, es importante recordar que siempre que se pasan arreglos como argumento, estos se pasan por referencia. Y además, que el nombre de un arreglo es de por sí una referencia al mismo. Teniendo esto presente, veremos una función encargada de poner en cero un arreglo de 4 elementos, y su utilización en un programa principal.

```
1  void limpia(float v[4])
2  {
3    int i;
4    for(i=0; i<4; i++) v[i] = 0;
5  }
6
7  int main()
8  {
9    float vect[4] = {2,-1,3, 5};
10
11   limpia(vect);
12   return 0;
13 }
```

En el ejemplo, el vector de dimensión 4 es inicializado en la linea 9, y luego es pasado *por referencia* en la linea 11 a la función `limpia`. Por lo tanto, los cambios que se hagan sobre los elementos del vector en dicha función se verán reflejados en el algoritmo principal. Como consecuencia de esto, al ejecutarse este programa, el arreglo `vect` terminará con todos sus elementos en cero.

Un aspecto que no fue explicado, pero que también puede verse en este ejemplo, es que toda función que se quiera utilizar en un punto del programa debe figurar escrita antes de su utilización.

Referencias

[1] L. Joyanes Aguilar, *Fundamentos de programación: algoritmos y estructura de datos.* No. 004.42, McGraw-Hill, 1988.

[2] Wikipedia, "Lenguaje de programación — Wikipedia, la enciclopedia libre," 2016.

[3] Wikipedia, "Anexo:lenguajes de programación — Wikipedia, la enciclopedia libre," 2016.

[4] Wikipedia, "C (lenguaje de programación) — Wikipedia, la enciclopedia libre," 2016.

[5] B. Kernighan and D. Ritchie, *The C Programming Language*. Pearson Education, 1991.

[6] G. Gagliano, C. Alarcón, L. Angleone, and otros, *Elementos Esenciales para Programacion*. No. 004.42, Proyecto Latin, 2014.