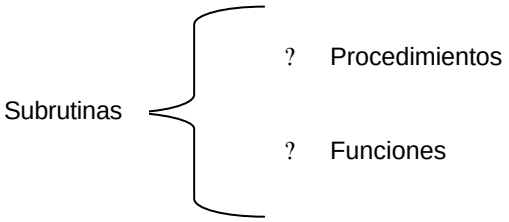
USO DE SUBRUTINAS, TRANSMISIÓN DE PARÁMETROS Y COMPILACIÓN CONDICIONAL EN C++

Bruno López Takeyas
Instituto Tecnológico de Nuevo Laredo
Reforma Sur 2007, C.P. 88250, Nuevo Laredo, Tamps. México
http://www.itnuevolaredo.edu.mx/takeyas
E-mail: takeyas@itnuevolaredo.edu.mx

Básicamente una Subrutina es un segmento de código que se escribe sólo una vez pero puede invocarse o ejecutarse muchas veces. Existen dos tipos: Procedimientos y Funciones.



1. Procedimientos

Son un tipo de subrutina que ejecuta un conjunto de acciones sin devolver valor alguno como resultado de dichas operaciones. Estos se identifican por su declaración *void()*.

P. ejem.

```
void Rutina(void);
void TAREA(void);
```

2. Funciones

A diferencia de los procedimientos, las funciones después de ejecutar un conjunto de acciones devuelven sólo un valor del tipo usado en la declaración de ésta por medio de *return()*.

P. ejem.

```
int SUMA(void); // Devuelve un valor de tipo entero
float CALCULA(void); // Devuelve un valor de tipo real
```

3. Limitación de return()

La cláusula *return()* sólo devuelve un valor. Si se desea que la función devuelva más de un valor debe usarse otro mecanismo.

4. Variables locales y globales

Las variables que se declaran dentro de una subrutina se llaman locales mientras que las variables globales se conocen a través del programa entero y se pueden usar en cualquier segmento de código manteniendo su valor. Se pueden declarar variables globales declarándolas fuera de cualquier función. Cualquier función puede acceder a ellas sin tener en cuenta en qué función esté dicha expresión.

5. Usando argumentos para pasar datos a subrutinas

El mecanismo para enviar información a las subrutinas es llamado argumento (algunos autores lo conocen como parámetro) y son los datos que se colocan entre paréntesis al invocarlas.
P. ejem.

```
int PROCESO(int x, float y)
```

Argumentos

Se pueden enviar varios argumentos a una subrutina, sin embargo es necesario precisar que deben estar declaradas las variables receptoras en el orden indicado considerando el tipo de dato apropiado.

6. Recibiendo un valor de una función

Una vez que se invoca una función es necesario utilizar la variable capaz de recibir el valor calculado por ésta, la cual debe ser del mismo tipo de la función. P. Ejem.

```
a=PROCESO(3, 5.25);
```

en el caso anterior, la variable "a" recibe el valor calculado por la función "PROCESO", quien acepta los argumentos 3 y 5.25 respectivamente.

```
/*
 Programa para el paso de argumentos por
referencia

Instructor: M.C. Bruno Lopez Takeyas
*/

#include <conio.h>
#include <iostream.h>

void RUTINA(int *y);  // Declaracion del
procedimiento RUTINA que acepta
                      // un argumento
(apuntador a un entero)

void main(void)
{
 int x=3;

 clrscr();
 cout << "\n\r Antes de la rutina x=" << x;

 RUTINA(&x);  // Envio de la direccion de "x"
como argumento

 cout << "\n\n\n\r Despues de la rutina x=" <<
x;

 getch();
 return;
}

void RUTINA(int *y)
{
  cout << "\n\n\n\r Valor recibido por y=" <<
*y;
  *y+=5;
  cout << "\n\n\n\r Valor modificado de y=" <<
*y;
  return;
}
```

7. Paso de argumentos por referencia

Existen dos formas de pasar argumentos a una subrutina: por valor y por referencia. Hasta este punto sólo se ha analizado el envío de valores, pero también se puede enviar la dirección de memoria de una variable a una subrutina.

Todos los arreglos se pasan por referencia a una subrutina

En el ejemplo de la Fig. 1 se muestra una variable ***x*** de tipo entero, que se pasa por referencia (se manda su dirección de memoria) a un procedimiento llamado RUTINA, quien recibe dicha dirección con una variable apuntador a un valor entero (***y***). La variable ***y*** recibe la dirección donde se aloja el valor de ***x*** y esto provoca que cuando se modifica lo que apunta ***y*** (valor de ***x***), indirectamente se modifica el valor de ***x***. Esto se refleja en memoria como lo indica la Fig. 2.

Memoria RAM

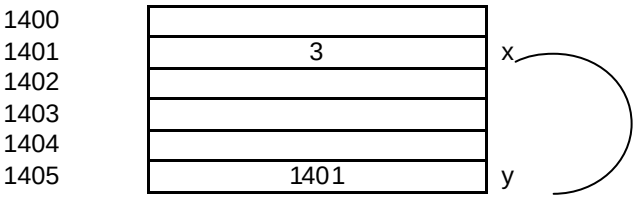


Fig. 2.- Apuntadores como receptores de direcciones

El programa de la Fig. 3 muestra una aplicación típica de envío de argumentos por referencia: el ordenamiento de un arreglo.

```
/*
 Programa para ordenar un arreglo
(pasandolo por referencia)

Instructor: M.C. Bruno Lopez Takeyas
*/

#include <conio.h>
#include <stdio.h>
#include <iostream.h>

#define TOP 10  // Cantidad de elementos
en el arreglo

void SORTEADOR(int B[TOP]);  //
Declaracion del que acepta el arreglo B
                      // para ordenarlo
en forma ascendente

void IMPRIME(int B[TOP]);  //
Procedimiento que imprime los elementos
                     // de un arreglo


void main(void)
{
 int A[TOP]={3,5,6,7,8,0,2,9,1,4};  //
Declaracion e inicializacion del
                             // arreglo
original

 clrscr();

 cout << "\n\rArreglo antes de sortearlo
...";
 IMPRIME(A);  // Imprime los elementos
del arreglo A antes de sortearlo

 SORTEADOR(A); // Procedimiento para
ordenar el arreglo A en forma ascendente

 cout << "\n\n\n\rArreglo despues de
sortearlo ...";
 IMPRIME(A);  // Imprime los elementos
del arreglo A despues de sortearlo
 getch();
 return;
}

void IMPRIME(int B[TOP])
{
int i;  // Variable local

  printf("\n\r");
  for(i=0;i<TOP;i++)
    printf("%2d ",B[i]);
  return;
}

void SORTEADOR(int B[TOP])
{
int i,j,aux;  // Variables locales

  for(i=0;i<TOP-1;i++)
    for(j=i+1;j<TOP;j++)
      if(B[i]>B[j])
      {
      aux=B[i];
      B[i]=B[j];      // Intercambio de
elementos
      B[j]=aux;
      }
  return;
}
```

Fig. 3.- Sorteador de un arreglo

8. Compilación condicional

Las directivas del preprocesador ***#if***, ***#ifdef***, ***#ifndef***, ***#else***, ***#elif***, ***#endif***, compilarán selectivamente varias porciones de un programa. La idea general es que si la expresión después de ***#if***, ***#ifdef*** o ***#ifndef*** es cierta, entonces el código que está entre una de las precedentes y un ***#endif*** se compilará; de lo contrario se saltará. La directiva ***#endif*** marca el final de un bloque ***#if***. El ***#else*** se puede usar con cualquiera de los anteriores de manera similar a las sentencias ***else*** e ***if***. El programa de la Fig. 4 ilustra el uso de estas directivas.

```
/*
 Programa para mostrar la forma de
compilar condicionalmente un programa

Instructor: M.C. Bruno Lopez Takeyas
*/

#include <conio.h>
#include <stdio.h>
#include <iostream.h>

#define CAPTURAR_ELEMENTOS  // Bandera
para activar/desactivar la compilacion
                     // de un segmento
de codigo
```

```
                      // Si se omite
esta definicion, se cancela la
                      // captura de los
elementos desde el teclado

#define TOP 10  // Cantidad de elementos
en el arreglo

void SORTEADOR(int B[TOP]);  //
Declaracion del que acepta el arreglo B
                      // para ordenarlo
en forma ascendente

void IMPRIME(int B[TOP]);  //
Procedimiento que imprime los elementos
                     // de un arreglo

void main(void)
{
#ifdef CAPTURAR_ELEMENTOS
  int A[TOP];
  int i;
#else if
  int A[TOP]={3,5,6,7,8,0,2,9,1,4};  //
Declaracion e inicializacion del
                            // arreglo
original
#endif CAPTURAR_ELEMENTOS

 clrscr();

#ifdef CAPTURAR_ELEMENTOS
  for(i=0;i<TOP;i++)
  {
    printf("\n\rA[%d] ? ",i);
    cin >> A[i];
  }
#endif CAPTURAR_ELEMENTOS

 cout << "\n\rArreglo antes de sortearlo
...";
 IMPRIME(A);  // Imprime los elementos
del arreglo A antes de sortearlo

 SORTEADOR(A); // Procedimiento para
ordenar el arreglo A en forma ascendente

 cout << "\n\n\n\rArreglo despues de
sortearlo ...";
 IMPRIME(A);  // Imprime los elementos
del arreglo A despues de sortearlo
 getch();
 return;
}

void IMPRIME(int B[TOP])
{
int i;  // Variable local


  printf("\n\r");
  for(i=0;i<TOP;i++)
    printf("%2d ",B[i]);
  return;
}

void SORTEADOR(int B[TOP])
{
int i,j,aux;  // Variables locales

  for(i=0;i<TOP-1;i++)
    for(j=i+1;j<TOP;j++)
      if(B[i]>B[j])
      {
      aux=B[i];
      B[i]=B[j];      // Intercambio de
elementos
      B[j]=aux;
      }
  return;
}
```

Fig. 4.- Compilación condicional

9. Encabezados creados por el programador (archivos *.h)

Los archivos de encabezados (también conocidos como archivos “include”) son de texto tal como los que codifica el programador usando el editor de programas de Turbo C++. Regularmente se encuentran almacenados en el subdirectorio \INCLUDE. Es posible colocar estatutos en el listado de programas que no son código de programación sino mensajes para el compilador. Estos mensajes llamados “directivas del compilador”, informan al compilador de definiciones de frases. Ciertas directivas del compilador se agrupan en los archivos de encabezados y pueden ser incluídas en el código fuente de los programas antes de compilarse. Sin embargo, el programador puede diseñar segmentos de código (regularmente con subrutinas) que utiliza repetidamente en sus programas, es entonces cuando surge la necesidad de crear archivos de encabezados que sólo incluye en sus programas cada vez que los necesita, basta con codificar las rutinas y grabarlas en un archivo con extensión **h**. La

Fig. 5 muestra un ejemplo de la forma de incluir estos encabezados creados por el usuario.

```
#include “c:\\tarea\\CAPTURA.h”
```

Fig. 5.- Encabezados creados por el programador.

10. Bibliografía

? Barkakati Nabajyoti. **“The Waite Group´s. Turbo C Bible”**. Howard W. Sams & Company. Estados Unidos. 1990.

? García Badell, J. Javier. **"Turbo C. Programación en manejo de archivos"**. Macrobit.

? Deitel y Deitel. **“C++ Cómo programar”**. Segunda edición. Pearson-Prentice Hall. Estados Unidos. 1999.

? Lafore, Robert. **“The Waite Group´s. Turbo C. Programming for the PC”**. Revised Edition. Howard W. Sams & Company. Estados Unidos. 1990.

? Schildt, Herbert. **“Turbo C. Programación avanzada”**. Segunda edición, McGraw Hill. Estados Unidos. 1990.

? Staugaard, Andrew. **“Técnicas estructuradas y orientadas a objetos. Una introducción utilizando C++”**. Segunda edición. Prentice Hall. Estados Unidos. 1998.