GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: FUNCIONES, PROCEDIMIENTOS Y TRASPASO DE PARÁMETROS EN C++

---

1. Introducción

Cuando un programador competitivo implementa una solución a un problema o ejercicio de concurso se utiliza sin proponerselo en la mayoría uno o varios paradigmas de programación donde los que más predominan son el *Programación estructurada* y la *Programación funcional*, de los elementos que componen este último paradigma y su aplicación en la programación competitiva especificamente con el lenguaje C++ serán abordados en la presente guía

2. Conocimientos previos

2.1. Paradigma de programación

Se denominan paradigmas de programación a las formas de clasificar los lenguajes de programación en función de sus características. Los idiomas se pueden clasificar en múltiples paradigmas.

Algunos paradigmas se ocupan principalmente de las implicancias para el modelo de ejecución del lenguaje, como permitir efectos secundarios o si la secuencia de operaciones está definida por el modelo de ejecución. Otros paradigmas se refieren principalmente a la forma en que se organiza el código, como agrupar un código en unidades junto con el estado que modifica el código. Sin embargo, otros se preocupan principalmente por el estilo de la sintaxis y la gramática.

2.2. Programación estructurada

La programación estructurada es un paradigma de programación orientado a mejorar la claridad, calidad y tiempo de desarrollo de un programa de computadora recurriendo únicamente a subrutinas y a tres estructuras de control básicas: secuencia, selección (if y switch) e iteración (bucles for y while); asimismo, se considera innecesario y contraproducente el uso de la transferencia incondicional (GOTO); esta instrucción suele acabar generando el llamado código espagueti, mucho más difícil de seguir y de mantener, además de originar numerosos errores de programación.

2.3. Programación funcional

En informática, la programación funcional es un paradigma de programación declarativa basado en el uso de verdaderas funciones matemáticas. En este estilo de programación las funciones son ciudadanas de primera clase, porque sus expresiones pueden ser asignadas a variables como se haría con cualquier otro valor; además de que pueden crearse funciones de orden superior.

3. Desarrollo

La programación funcional se caracteriza por dividir la mayor cantidad posible de tareas en funciones o procedimientos, de esta forma estas tareas pueden ser usadas por otras funciones con diferentes objetivos. En el mundo de la programación, muchos acostumbramos hablar indistintamente de tres términos sin embargo poseen deferencias fundamentales.

Funciones: Las funciones son un conjunto de procedimiento encapsulados en un bloque, usualmente reciben parámetros, cuyos valores utilizan para efectuar operaciones y adicionalmente retornan un valor. Esta definición proviene de la definición de función matemática la cual posee un dominio y un rango, es decir un conjunto de valores que puede tomar y un conjunto de valores que puede retornar luego de cualquier operación.

Métodos: Los métodos y las funciones son funcionalmente idénticos, pero su diferencia radica en el contexto en el que existen. Un método también puede recibir valores, efectuar operaciones con estos y retornar valores, sin embargo en método está asociado a un objeto, básicamente un método es una función que pertenece a un objeto o clase, mientras que una función existe por sí sola, sin necesidad de un objeto para ser usada.

Procedimientos: Los procedimientos son básicamente lo un conjunto de instrucciones que se ejecutan sin retornar ningún valor, hay quienes dicen que un procedimiento no recibe valores o argumentos, sin embargo en la definición no hay nada que se lo impida. En el contexto de C++ un procedimiento es básicamente una función *void* que no nos obliga a utilizar una sentencia *return*.

3.1. Funciones en C++

La sintaxis para declarar una función es muy simple, veamos:

```
<tipo de dato> <nombreFuncion> (<parametros>){
   <bloque instrucciones>
}
```

Donde:

- **<tipo de dato>:** Se especifica el tipo de dato que retornará o devolverá la función. El tipo de dato debe corresponderse con los nativos del lenguaje o uno previemente definido por el propio programador previamente.
- **<nombreFuncion>:** Identificador de la función el cual debe cumplir con las misma restricciones y reglas para los identificadores de las variables. Como buena practica dicho indentificador debe indicar de forma corta de ser posible cual es el objetivo o que realiza la función.
- **<parametros>:** Grupo de variables definidas cada una por su tipo de dato e identificador, separadas por coma en caso de ser mas de una. Los parámetros son valores necesarios e indispensables para que la función pueda realizar sus operaciones e instrucciones. La necesidad de parámetros por parte de una función es definida por el programador que la implementa por tanto una función bien pudiera no tener parámetros ese caso se pone simplemente los paréntesis vacío (). Más adelante abordaremos un poco mas sobre los parámetros.
- **<bloque instrucciones>:** Conjunto de instrucciones o sentencias ya sean simples o compuestas que permiten ejecutar o llevar a cabo el objetivo con que fue creada la función. Dentro de dichas instrucciones estará la instrucción **return** la cual es la encargada de retornar el

resultado final de la función cuyo valor debe coincidir con el tipo de dato que retorna la función.

3.1.1. Consejos acerca de return

Debes tener en cuenta dos cosas importantes con la sentencia **return**:

- Cualquier instrucción que se encuentre después de la ejecución de return NO será ejecutada. Es común encontrar funciones con múltiples sentencias return al interior de condicionales, pero una vez que el código ejecuta una sentencia return lo que haya de allí hacia abajo no se ejecutará.
- El tipo del valor que se retorna en una función debe coincidir con el del tipo declarado a la función, es decir si se declara *int*, el valor retornado debe ser un número entero.

3.1.2. Invocando funciones C++

Ya hemos visto cómo se crean las funciones en C++, ahora veamos cómo hacemos uso de ellas o la invocamos.

```
//variante 1
<tipo de dato> <resultado> = <nombreFuncion> (<parametros>);

//variante 2
<tipo de dato> <resultado>;
<resultado> = <nombreFuncion> (<parametros>);

//variante 3
<nombreFuncion> (<parametros>);
```

Donde:

- **<tipo de dato>:** Se especifica el tipo de dato de la variable que recibirá o se le asignará el valor que retornará la función. Dicho tipo de dato debe ser igual al tipo de dato que retorna la función.
- **<resultado>:** Identificador de la variable que recibirá, almacenará o se le asignará el valor devuelto por la función.
- **<nombreFuncion>:** Nombre de la función que se desea invocar.
- **<parametros>:** Coleción de valores que necesita la función para su ejecucción. Dicha colección puede tener ninguno , uno o varios valores los cuales se separán por una coma y se pueden especificar el valor de forma literal o nombrar a la variable que tiene el valor que deseamos pasar a la función.

Como puedes notar es bastante sencillo invocar o llamar funciones en C++ (de hecho en cualquier lenguaje actual), sólo necesitas el nombre de la función y enviarle el valor de los parámetros.

Hay que hacer algunas salvedades respecto a esto. No obstante se debe tener en cuenta los siguientes detalles a la hora de invocar:

- El nombre de la función debe coincidir exactamente al momento de invocarla.
- El orden de los parámetros y el tipo debe coincidir. Hay que ser cuidadosos al momento de enviar los parámetros, debemos hacerlo en el mismo orden en el que fueron declarados y deben ser del mismo tipo (número, texto u otros).
- Cada parámetro enviado también va separado por comas.
- Si una función no recibe parámetros, simplemente no ponemos nada al interior de los paréntesis, pero SIEMPRE debemos poner los paréntesis.
- Invocar una función sigue siendo una sentencia habitual de C++, así que ésta debe finalizar con ';'como siempre.
- El valor retornado por una función puede ser asignado a una variable del mismo tipo.
- Una función puede llamar a otra dentro de sí misma o incluso puede ser enviada como parámetro a otra.

3.2. Procedimientos en C++

Los procedimientos son similares a las funciones, aunque más resumidos. Debido a que los procedimientos no retornan valores, no hacen uso de la sentencia return para devolver valores y no tienen tipo específico, solo **void**. Los procedimientos también pueden usar la sentencia return, pero no con un valor. En los procedimientos el return sólo se utiliza para finalizar allí la ejecución del procedimiento. Por tanto su sintaxis de declaración sería la siguiente:

```
void <nombreProcedimiento> (<parametros>){
   <bloque instrucciones>
}
```

Donde:

- **<nombreProcedimiento>:** Identificador del procedimiento el cual debe cumplir con las misma restricciones y reglas para los identificadores de las variables. Como buena practica dicho indentificador debe indicar de forma corta de ser posible cual es el objetivo o que realiza el procedimiento.
- **<parametros>:** Grupo de variables definidas cada una por su tipo de dato e identificador, separadas por coma en caso de ser mas de una. Los parámetros son valores necesarios e indispensables para que el procedimiento pueda realizar sus operaciones e instrucciones. La necesidad de parámetros por parte de un procedimiento es definida por el programador que la implementa por tanto un procedimiento bien pudiera no tener parámetros ese caso se pone simplemente los paréntesis vacío (). Más adelante abordaremos un poco mas sobre los parámetros.

- **<bloque instrucciones>:** Conjunto de instrucciones o sentencias ya sean simples o compuestas que permiten ejecutar o llevar a cabo el objetivo con que fue creada el procedimiento.

3.2.1. Invocando procedimientos C++

Ya hemos visto cómo se crean los procedimientos en C++, ahora veamos cómo hacemos uso de ellos o lo invocamos.

```
<nombreProcedimiento> (<parametros>);
```

Donde:

- **<nombreProcedimiento>:** Nombre del procedimiento que se desea invocar.
- **<parametros>:** Coleción de valores que necesita el procedimiento para su ejecucción. Dicha colección puede tener ninguno , uno o varios valores los cuales se separán por una coma y se pueden especificar el valor de forma literal o nombrar a la variable que tiene el valor que deseamos pasar al procedimiento.

3.3. Parámetros en C++

Los parámetros son variables locales a los que se les asigna un valor antes de comenzar la ejecución del cuerpo de una función o procedimientos. Su ámbito de validez, por tanto, es el propio cuerpo de la función o procedimientos.

Hay algunos detalles respecto a los parámetros de una función o procedimientos, veamos:

1. Una función o procedimiento pueden tener una cantidad cualquier de parámetros, es decir pueden tener cero, uno, tres, diez, cien o más parámetros. Aunque habitualmente no suelen tener más de 4 o 5.
2. Si una función tiene más de un parámetro cada uno de ellos debe ir separado por una coma.
3. Los parámetros de una función también tienen un tipo y un nombre que los identifica. El tipo del argumento puede ser cualquiera y no tiene relación con el tipo de la función.

En C++ existen dos alternativas para la transmisión de los parámetros a las funciones:

3.3.1. Paso por valor

Los parámetros formales son variables locales a la función, es decir, solo accesibles en el ámbito de ésta. Reciben como valores iniciales los valores de los parámetros actuales. Posteriores modificaciones en la función de los parámetros formales, al ser locales, no afectan al valor de los parámetros actuales.

Pasar parámetros por valor significa que a la función se le pasa una copia del valor que contiene el parámetro actual. Los valores de los parámetros de la llamada se copian en los parámetros de la cabecera de la función. La función trabaja con una copia de los valores por lo que cualquier

modificación en estos valores no afecta al valor de las variables utilizadas en la llamada. Aunque los parámetros actuales (los que aparecen en la llamada a la función) y los parámetros formales (los que aparecen en la cabecera de la función) tengan el mismo nombre son variables distintas que ocupan posiciones distintas de memoria. Por defecto, todos los argumentos salvo los arreglos se pasan por valor.

3.3.2. Paso por referencia

Los parámetros formales no son variables locales a la función, sino alias de los propios parámetros actuales. ¡No se crea ninguna nueva variable! Por tanto, cualquier modificación de los parámetros formales afectará a los actuales. El paso de parámetros por referencia permite que la función pueda modificar el valor del parámetro recibido. Vamos a explicar dos formas de pasar los parámetros por referencia:

1. **Paso de parámetros por referencia basado en punteros al estilo C:** Cuando se pasan parámetros por referencia, se le envía a la función la dirección de memoria del parámetro actual y no su valor. La función realmente está trabajando con el dato original y cualquier modificación del valor que se realice dentro de la función se estará realizando con el parámetro actual. Para recibir la dirección del parámetro actual, el parámetro formal debe ser un puntero.
2. **Paso de parámetros por referencia usando referencias al estilo C++:** Una referencia es un nombre alternativo (un alias, un sinónimo) para un objeto. Una referencia no es una copia de la variable referenciada, sino que es la misma variable con un nombre diferente. Utilizando referencias, las funciones trabajan con la misma variable utilizada en la llamada. Si se modifican los valores en la función, realmente se están modificando los valores de la variable original.

4. Implementación

4.1. Funciones y procedimientos

```
int funcionEntera(){ //Funcion sin parametros
   int suma = 5+5;
   return suma;
   //Aca termina la ejecucion de la funcion
   return 5+5;//Este return nunca se ejecutara
   //Intenta intercambiar la linea 3 con la 5
   int x = 10; //Esta linea nunca se ejecutara
}

char funcionChar(int n){//Funcion con un parametro
   if(n == 0){ //Usamos el parametro en la funcion
      return 'a'; //Si n es cero retorna a
                  //Notar que de aqui para abajo no se ejecuta nada mas
   }
   return 'x';//Este return solo se ejecuta cuando n NO es cero
```

```
}

bool funcionBool(int n, string mensaje){//Funcion con dos parametros
   if (n == 0){//Usamos el parametro en la funcion
      cout << mensaje;//Mostramos el mensaje
      return 1; //Si n es cero retorna 1
      return true; //Equivalente
   }
   return 0;//Este return solo se ejecuta cuando n NO es cero
   return false;//Equivalente
}

void procedimiento(int n, string nombre){
   if(n == 0){
      cout << "hola" << nombre;
      return;
  }
  cout << "adios" << nombre;
}

int main(){
   funcionEntera(); //Llamando o invocando a una funcion sin argumentos
   bool respuesta = funcionBool(1, "hola"); //Asignando el valor
                                            //retornado una variable
   procedimiento(0, "Juan");//Invocando el procedimiento
   //Usando una funcion como parametro
   procedimiento(funcionBool(1, "hola"), "Juan");
   return 0;
}
```

En el código anterior podemos ver cómo todas las funciones han sido invocadas al interior de la función main (la función principal), esto nos demuestra que podemos hacer uso de funciones al interior de otras. También vemos cómo se asigna el valor retornado por la función a la variable *'respuesta'* y finalmente, antes del return, vemos cómo hemos usado el valor retornado por *funcionBool* como parámetro del procedimiento.

4.2. Parámetros

4.2.1. Paso por valor

```
#include <iostream>
using namespace std;

int invertir(int num){
   int inverso = 0, cifra;
   while (num != 0){
      cifra = num % 10;
      inverso = inverso * 10 + cifra;
```

```
        num = num / 10;
    }
    return inverso;
}

int main(){
    int num; int resultado;
    cout << "Introduce un numero entero: ";
    cin >> num;
    resultado = invertir(num);
    cout << "Numero introducido: " << num << endl;
    cout << "Numero con las cifras invertidas: " << resultado << endl;
}
```

En la llamada a la función el valor de la variable *num* se copia en la variable *num* de la cabecera de la función. Aunque tengan el mismo nombre, se trata de dos variables distintas. Dentro de la función se modifica el valor de *num*, pero esto no afecta al valor de *num* en *main*. Por eso, al mostrar en *main* el valor de *num* después de la llamada aparece el valor original que se ha leído por teclado.

4.2.2. Paso de parámetros por referencia basado en punteros al estilo C

```
 #include <iostream>
using namespace std;
void intercambio(int *x, int *y){
    int z;
    z = *x;
    *x = *y;
    *y = z;
}

int main(){
    int a, b;
    cout << "Introduce primer numero: ";
    cin >> a;
    cout << "Introduce segundo numero: ";
    cin >> b;
    cout << endl;
    cout << "valor de a: " << a << " valor de b: " << b << endl;
    intercambio(&a, &b);
    cout << endl << "Despues del intercambio: " << endl << endl;
    cout << "valor de a: " << a << " valor de b: " << b << endl;
}
```

En la llamada, a la función se le envía la dirección de los parámetros. El operador que obtiene la dirección de una variable es &.

```
intercambio(&a, &b);
```

En la cabecera de la función, los parámetros formales que reciben las direcciones deben ser punteros. Esto se indica mediante el operador *.

```
void intercambio(int *x, int *y)
```

Los punteros x e y reciben las direcciones de memoria de las variables a y b. Al modificar el contenido de las direcciones x e y, indirectamente estamos modificando los valores a y b. Por tanto, pasar parámetros por referencia a una función o procedimiento es hacer que la función o procedimiento acceda indirectamente a las variables pasadas.

4.2.3. Paso de parámetros por referencia usando referencias al estilo C++

```
#include <iostream>
using namespace std;
void intercambio(int &x, int &y){
   int z;
   z = x;
   x = y;
   y = z;
}

int main( ){
   int a, b;
   cout << "Introduce primer numero: ";
   cin >> a;
   cout << "Introduce segundo numero: ";
   cin >> b;
   cout << endl;
   cout << "valor de a: " << a << " valor de b: " << b << endl;
   intercambio(a, b);
   cout << endl << "Despues del intercambio: " << endl << endl;
   cout << "valor de a: " << a << " valor de b: " << b << endl;
}
```

En la declaración de la función o procedimiento y en la definición se coloca el operador referencia a & en aquellos parámetros formales que son referencias de los parámetros actuales:

```
void intercambio(int &x, int &y)  //definicion de la funcion
```

Cuando se llama a la función:

```
intercambio(a, b);
```

se crean dos referencias (x e y) a las variables a y b de la llamada. Lo que se haga dentro de la función con x e y se está haciendo realmente con a y b. La llamada a una función usando referencias es idéntica a la llamada por valor.

5. Aplicaciones

Las funciones y los procedimientos son herramientas indispensables para el programador, tanto las funciones y procedimientos creadas por él mismo como las que le son proporcionadas por otras librerías, cualquiera que sea el caso, las funciones y los procdemientos permiten automatizar tareas repetitivas, encapsular el código que utilizamos, e incluso mejorar la seguridad, confiabilidad y estabilidad de nuestros programas. Dominar el uso de funciones y procdemientos es de gran importancia, permiten modularizar nuestro código, separarlo según las tareas que requerimos.

La utilización de funciones o procedimientos nos permitirá desarrollar soluciones mucho más legible y fácil de testear, nos concentramos en qué estamos haciendo y no en cómo se esta haciendo.

6. Complejidad

La complejidad de los funciones y procedimientos van a depender del grupo de instrucciones que la conforman e incluso de las invocaciones que en ella se haga a otras funciones o procedimientos.

7. Ejercicios

A continuación les proponemos un grupo de ejercicios bien sencillos que pueden ser solucionados directamente en la función main pero le proponemos que ahora le den solución usando ya sea funciones o procedimientos con o sin parámetros que sean invocados desde la función *main*:

- DMOJ - Hello World
- DMOJ - El Ogro Ork
- DMOJ - No divisibles
- DMOJ -Cortes al tablero de ajedrez
- DMOJ - Coordenadas del Cuadrado
- DMOJ - Par o impar
- DMOJ - Alex y la IOI
- DMOJ - Las notas de Ork
- DMOJ - A Plus B
- MOG - N - Arithmetic Mean

- MOG - A - An easy task I
- MOG - A - An easy task II
- MOG - A - A+B
- MOG - B - A-B