UNIDAD V -FUNCIONES

Programación

Tabla de Contenidos

¿FUNCIONES, MÉTODOS O PROCEDIMIENTOS?

- Funciones
- Métodos
- Procedimientos

MÉTODOS Y PROCEDIMIENTOS

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

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.

Las **funciones** son un conjunto de instrucciones que realizan una tarea específica. En general toman ciertos valores de entrada, llamados parámetros y proporcionan un valor de salida o valor de retorno; aunque en **C++**, tanto unos como el otro son opcionales, y pueden no existir.

```
INT SUM(INT A, INT B)
{
   RETURN A + B;
}
```

```
int main()
{
    int i = sum(10, 32);
    int j = sum(i, 66);
    cout << "The value of j is" << j << endl;
// 108
}
```

La función se puede invocar, o *llamar*, desde cualquier número de lugares en el programa. Los valores que se pasan a la función son los *argumentos, cuyos* tipos deben ser compatibles con los tipos de parámetro en la definición de función.

ELEMENTOS DE UNA DECLARACIÓN DE FUNCIÓN

Una *declaración* de función mínima consta del tipo de valor devuelto, el nombre de función y la lista de parámetros (que pueden estar vacíos), junto con palabras clave opcionales que proporcionan instrucciones adicionales al compilador. El ejemplo siguiente es una declaración de función:

```
int sum(int a, int b);
```

Una definición de función consta de una declaración, más el *cuerpo*, que es todo el código entre las llaves:

```
int sum(int a, int b)
{
    return a + b;
}
```

LOS ELEMENTOS NECESARIOS DE UNA DECLARACIÓN DE FUNCIÓN SON LOS SIGUIENTES:

1. El tipo de valor devuelto, que especifica el tipo del valor que devuelve la función, o **void** si no se devuelve ningún valor. En C++11, **auto** es un tipo de valor devuelto válido que indica al compilador que deduzca el tipo de la instrucción return. En C++14, también se permite decltype(auto). Para obtener más información, consulte más adelante Deducción de tipos en tipos de valor devueltos.
2. El nombre de función, que debe comenzar con una letra o un carácter de subrayado y no puede contener espacios. En general, un carácter de subrayado inicial en los nombres de función de la biblioteca estándar indica funciones de miembro privado o funciones no miembro que no están pensadas para que las use el código.
3. La lista de parámetros, que es un conjunto delimitado por llaves y separado por comas de cero o más parámetros que especifican el tipo y, opcionalmente, un nombre local mediante el cual se puede acceder a los valores de dentro del cuerpo de la función

DEFINICIONES DE FUNCIÓN

- Una definición de *función* consta de la declaración y el cuerpo de la función, entre llaves, que contiene declaraciones variables, instrucciones y expresiones. En el ejemplo siguiente se muestra una definición de función completa:

```
tipo_ retorno    nombre_función (lista_de_parámetros)
          Cabecera de la función
{
declaraciones_de_variables;
declaraciones;
}
```

EJERCICIOS 1 Y 2

```
//Función C++ para sumar dos números
int suma(int a, int b)
{
        int c;
        c = a + b;
        return c;
}
```

```
int funcionEntera()//Función sin parámetros
{
   int suma = 5+5;
   return suma; //Acá termina la ejecución de la función
   return 5+5;  //Este return nunca se ejecutará
                //Intenta intercambiar la línea 3 con la 5 int x = 10;
                //Esta línea nunca se ejecutará
}
```

PARÁMETROS POR VALOR

La función cuadrado() es un clásico ejemplo que muestra el paso de parámetros por valor, en ese sentido la función cuadrado() recibe una copia del parámetro n. En la misma función se puede observar que se realiza un calculo (n ∗ n), sin embargo el parámetro original no sufrirá cambio alguno, esto seguirá siendo cierto aún cuando dentro de la función hubiera una instrucción parecida a n = n ∗ n; o n∗ = n;.

Sintaxis

```
<tipo> [clase::] <nombre> ( [Parámetros] )
{
    cuerpo;
}
```

Ejemplo

```
// regresar el cuadrado de un número
double cuadrado(double n)
{
    return n*n;
}
```

PARÁMETROS POR REFERENCIA

La función cuadrado2() es un clásico ejemplo que muestra el paso de parámetros por referencia, en ese sentido la función cuadrado2() recibe el parámetro n. En la misma función se puede observar que se realiza un calculo (n ∗ n), sin embargo el parámetro original sufrirá cambio, esto seguirá siendo cierto aun cuando dentro de la función hubiera una instrucción parecida a n = n ∗ n; o n∗ = n;

Ejemplo

```
// regresar el cuadrado de un número
double cuadrado2(double &n)
{
    n *= n;
    return n;
}
```

PARÁMETROS CONSTANTES

Los parámetros usados por una función pueden declararse como constantes (const) al momento de la declaración de la función. Un parámetro que ha sido declarado como constante significa que la función no podrá cambiar el valor del mismo (sin importar si dicho parámetro se recibe por valor o por referencia).

Ejemplo

```
int funcionX( const int n );
void printstr( const char *str );
```

PARÁMETROS CON VALOR POR DEFECTO

Los parámetros usados por una función pueden declararse con un valor por defecto. Un parámetro que ha sido declarado con valor por defecto es opcional a la hora de hacer la llamada a la función.

Ejemplo

```
void saludo( char* mensaje = "Hola sudafrica 2010" );

la misma puede ser invocada como:

saludo();  // sin parámetro
saludo("Sea usted bienvenido a C++"); // con parámetro
```

EJEMPLO 3

```
char funcionChar(int n)//Función con un parámetro {
        if(n == 0)//Usamos el parámetro en la función
       {
       return 'a'; //Si n es cero retorna a
                              //Notar que de aquí para abajo no se ejecuta nada
       más
        }
 return 'x';//Este return sólo se ejecuta cuando n NO es cero
 }
```

EJEMPLO 4

```
bool funcionBool(int n, string mensaje)//Función con dos parámetros
{
     if(n == 0)//Usamos el parámetro en la función
          {
       cout << mensaje;
       //Mostramos el mensaje
            return 1;
           //Si n es cero retorna 1
           return true;
           //Equivalente
         }
   return 0;
      //Este return sólo se ejecuta cuando n NO es cero return false;//Equivalente
}
```

PROTOTIPOS DE FUNCIONES

Antes de usar una función C debe tener *conocimiento* acerca del tipo de dato que regresará y el tipo de los parámetros que la función espera.

El estándar ANSI de C introdujó una nueva (mejor) forma de hacer lo anterior respecto a las versiones previas de C.

La importancia de usar prototipos de funciones es la siguiente:

- Se hace el código más estructurado y por lo tanto, más fácil de leer.
- Se permite al compilador de C revisar la *sintaxis* de las funciones llamadas.

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

float encontprom(int  , int );
main()
  {
    int a=7, b=10;
    float resultado;
    resultado= encontprom(a, b);
    printf("Promedio=%f\n",resultado);

    system("pause");
    return 0;
  }
float encontprom(int num1, int num2)
{
    float promedio;

    promedio = (num1 + num2)/2.0;
    return promedio;
}
```

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

float encontprom(int num1, int num2)
{
   float promedio;

   promedio = (num1 + num2);
   return promedio;
}
main()
 {
   int a=7, b=10;
   float resultado;

   resultado= encontprom(a, b);
   printf("Promedio=%d\n",resultado);

   system("pause");
   return 0;
 }
```

```
// Programa que lee un año y muestra si es o no bisiesto
#include <iostream>
using namespace std;
int bisiesto(int);   //declaración o prototipo de la función
int main()
{
   int anio;
   cout<<"Introduce a"<<(char)164<<"o: "; //164 ascii de ñ
   cin >> anio;
   if(bisiesto(anio))  //llamada a la función
      cout << "Bisiesto" << endl;
   else
      cout << "No es bisiesto" << endl;
```

```
int bisiesto(int a)   //definición de la función
{
    if(a%4==0 and a%100!=0 or a%400==0)
        return 1;
    else
        return 0;
```

Recursividad

Es una técnica que permite que una función se llame a sí misma

Un algoritmo es recursivo si al estar encapsulado en una función es llamado de la misma función

La recursión va ligado a repetición

Introducción

- La recursividad es una alternativa a la repetición o iteración.
  - En tiempo de computadora y ocupación de memoria, la solución recursiva es menos eficiente que la iterativa, existen situaciones en las que la recursividad es una solución simple y natural a un problema que en otro caso será difícil de resolver

Notación de función

Una función recursiva es aquella que se llama a sí mismo, bien directamente o bien indirectamente a través de otra función

Una función que tiene sentencias entre las que se encuentra al menos una que llama a la propia función se dice que es recursiva. Así, la organización recursiva de una función funcion1 sería la siguiente:

```
void funcion1(...)
{
    ... funcion1(); // llamada recursiva ...
}
```

Caso Base

una solución simple para un caso

Paso Recursivo

puede haber más de un caso base

Función

Partes de funciones recursivas

- **Caso base:** Es el caso más simple de una función recursiva, y simplemente devuelve un resultado (el caso base se puede considerar como una salida no recursiva).

- **Caso general**: Relaciona el resultado del algoritmo con resultados de casos más simples. Dado que cada caso de problema aparenta o se ve similar al problema original, la función llama una copia nueva de si misma, para que empiece a trabajar sobre el problema más pequeño y esto se conoce como una llamada recursiva y también se llama el paso de recursión.

Método

Obtener una definición exacta del problema a resolver. (Esto, por supuesto, es el primer paso en la resolución de cualquier problema de programación).

A continuación, determinar el tamaño del problema completo que hay que resolver. Este tamaño determinará los valores de los parámetros en la llamada inicial a la función.

Resolver el caso base en el que el problema puede expresarse no recursivamente. Por último, resolver el caso general correctamente en términos de un caso más pequeño del mismo problema, una

TIPOS DE RECURSIVIDAD

Directa

Indirecta

Tipos de recursión

- **Recursividad directa**: Aquella en cuya definición sólo aparece una llamada recursiva. Se puede transformar con facilidad en algoritmos iterativos.
    - Factorial

- **Recursividad indirecta**: Se da cuando hay más de una llamada a sí misma dentro del cuerpo de la función, resultando más difícil de hacer de forma iterativa.
    - Fibonacci

VENTAJAS Y DESVENTAJAS

Ventajas

- Solución natural
- La solución iterativa es más difícil de implementar
- Resuelve problemas complejos

Desventajas

- Se **puede** llegar a un clico infinito
- Es más complejo de programar.

Ejemplo con recursividad

- El factorial de un entero no negativo *n*, esta definido como:

$$n! = n * (n-1) * (n-2) * \ldots * 2 * 1$$

- Donde 1! es igual a 1 y 0! se define como 1.
- El factorial de un entero *k* puede calcularse de manera iterativa como sigue:

```
fact  = 1;
K = 3;
for (i = 1; i <= k; i++){
    fact = fact *(i+1);
}
```

i	fact *(i+1)	fact
1	1 *(0+1)	1
2	1 *(1+1)	2
3	2 *(2+1)	6

Ejemplo con recursividad

- Ahora de manera recursiva se puede definir el factorial como:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n * (n - 1)! & \text{si } n > 0 \end{cases}$$

Donde el caso base es: 1 Si $n = 0$.

El caso general es: $n * (n\text{-}1)!$ Si $n > 0$.

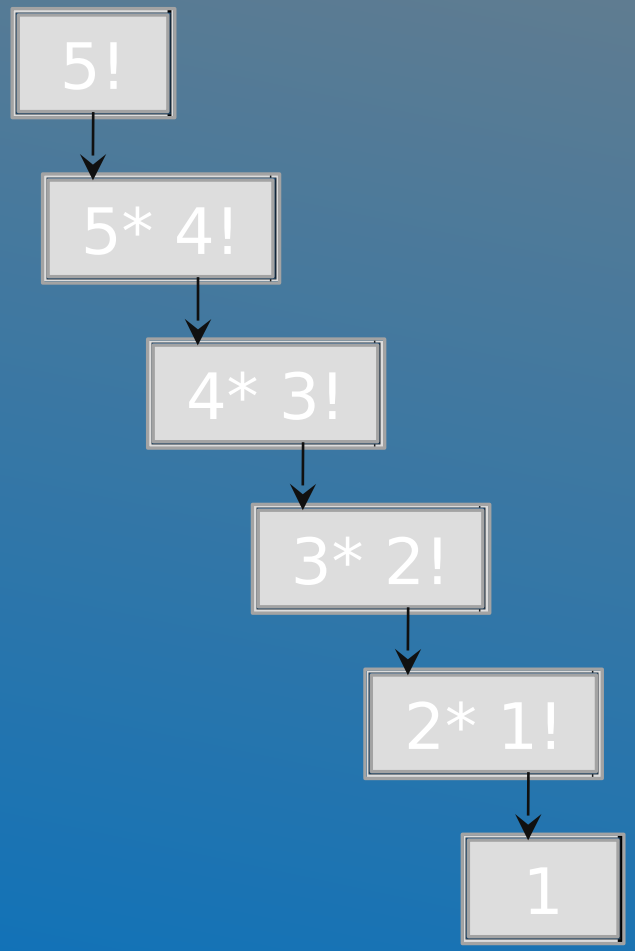
- Por ejemplo si se quiere calcular el factorial de 5, se tendría:

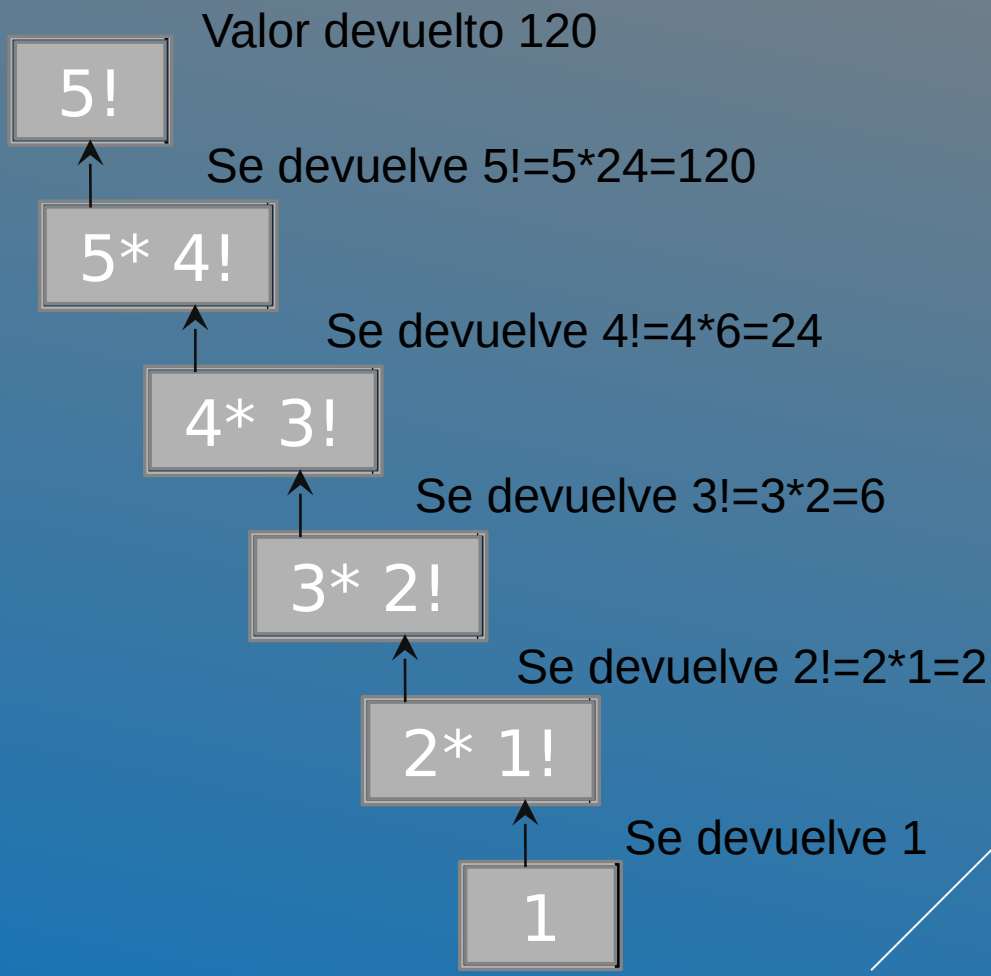
5! = 5*4*3*2*1

5! = 5*(4*3*2*1)

5! = 5*4!

Ejemplo con recursividad





Iterativa

```
#include <iostream>
using namespace std;
long factorial (int n);

int main()
{
  int n;
  do {
    cout << "Introduzca número n: ";
    cin >> n;
  }while (n < 0);
  cout  << " \t" << n << "!= " << factorial(n) << endl;
  return 0;
}
```

```
long factorial (int n)
{
  if (n <= 1)
    return 1;
  else
  {
    long resultado = n * factorial(n - 1);
    return resultado;
  }
}
```

RECURSIVA

Se observa en esta serie que comienza con 0 y 1, y tiene la propiedad de que cada elemento es la suma de los dos elementos anteriores, por ejemplo:

0 + 1 = 1

1 + 1 = 2

2 + 1 = 3

3 + 2 = 5

5 + 3 = 8

...

Entonces se puede establecer que:

fibonacci(0) = 0

fibonacci(1) = 1

...

fibonacci(n)= fibonacci(n - 1) + fibonacci(n - 2)

SERIE FIBONACCI

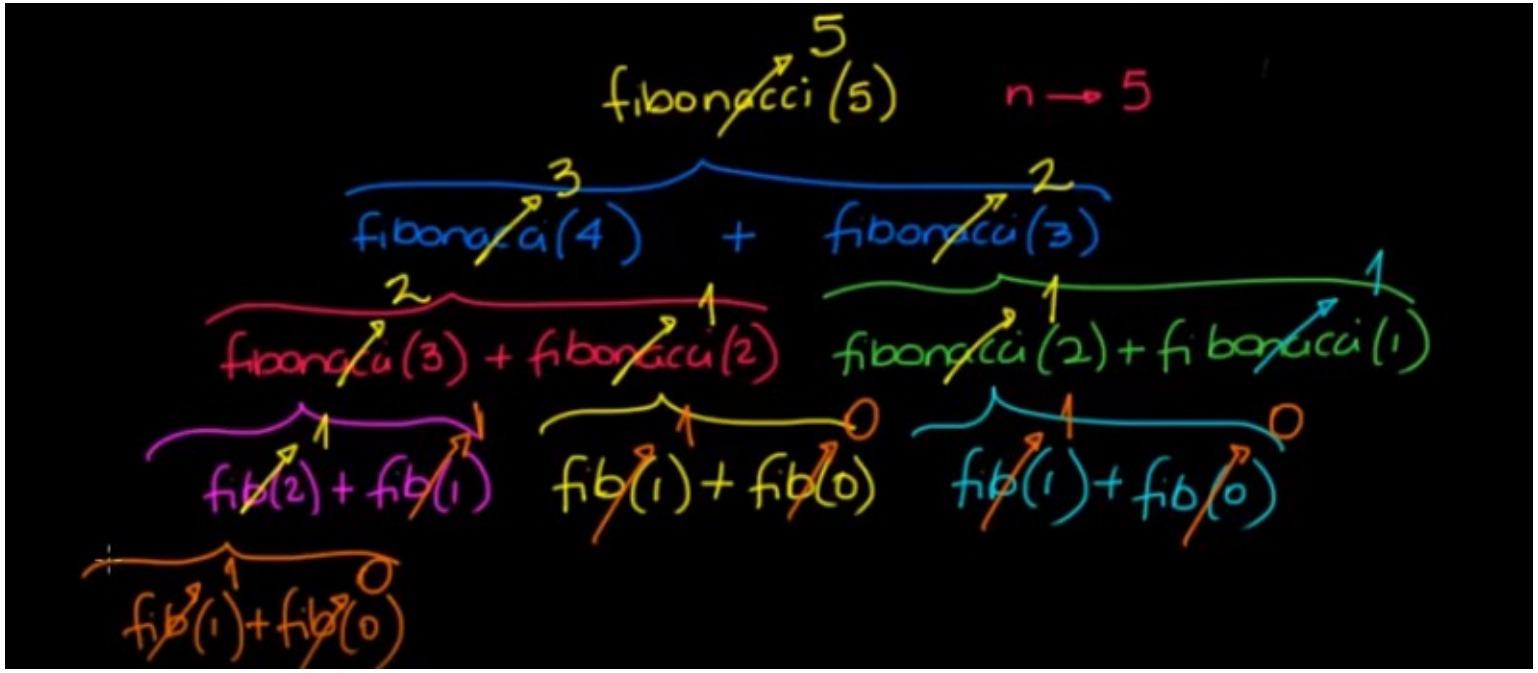
Y LA DEFINICIÓN RECURSIVA SERÁ:

fibonacci(n) = n → si n = 0 o n = 1

fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2) → si n > = 2

```
long fibonacci (int n)
{
  if (n == 0 || n == 1)
    return n;
  fibinf = 0;
  fibsup = 1;
  for (int i = 2; i <= n; i++)
  {
    int x;
    x = fibinf;
    fibinf = fibsup;
    fibsup = x + fibinf;
  }
  return fibsup;
}
```

```
long fibonacci (int n)
{
  if (n == 0 || n == 1)
    return n;
  else
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```



RECURSIÓN INFINITA

En realidad la recursión infinita significa que cada llamada recursiva produce otra llamada recursiva y ésta a su vez otra llamada recursiva y así para siempre. En la práctica dicha función se ejecutará hasta que la computadora agota la memoria disponible y se produce una terminación anormal del programa.

Recursión vs. Iteración

- Las principales cuestiones son la claridad y la eficiencia de la solución.
  - En general: Una solución no recursiva es más eficiente en términos de tiempo y espacio de computadora.
  - La solución recursiva puede requerir gastos considerables, y deben guardarse copias de variables locales y temporales.
  - Aunque el gasto de una llamada a una función recursiva no es peor, esta llamada original puede ocultar muchas capas de llamadas recursivas internas.
  - El sistema puede no tener suficiente espacio para ejecutar una solución recursiva de algunos problemas.

Recursión vs. Iteración

- Una solución recursiva particular puede tener una ineficiencia inherente. Tal ineficiencia no es debida a la elección de la implementación del algoritmo; más bien, es un defecto del algoritmo en si mismo.
- Un problema inherente es que algunos valores son calculados una y otra vez causando que la capacidad de la computadora se exceda antes de obtener una respuesta.
- La cuestión de la claridad en la solución es, no obstante, un factor importante.
- En algunos casos una solución recursiva es más simple y más natural de escribir.