

# Funciones

Programación en C++



# Problemas complejos → programas complejos

- Una forma de solucionar un problema complejo es dividirlo en partes (problemas más sencillos) y dividir estas partes en otras más simples, hasta llegar a problemas más pequeños fáciles de resolver.
- Normalmente las partes en que se divide un programa deben poder desarrollarse independientemente entre sí.
- Cuando vamos de mayor a menor, se le llamo diseño descendente.
- Un subprograma se conoce también como procedimientos(subrutinas) o funciones.
- Una función puede :
  - Aceptar datos
  - Realizar algunos cálculos
  - Devolver resultados

# Funciones - Definición

- 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.

# Funciones

- Las funciones → herramienta indispensable para el programador
- Funciones creadas
  - por él mismo
  - proporcionadas por otras librerías
- Las funciones permiten:
  - automatizar tareas repetitivas
  - encapsular el código que utilizamos
  - e incluso mejorar la seguridad, confiabilidad y estabilidad de nuestros programas.

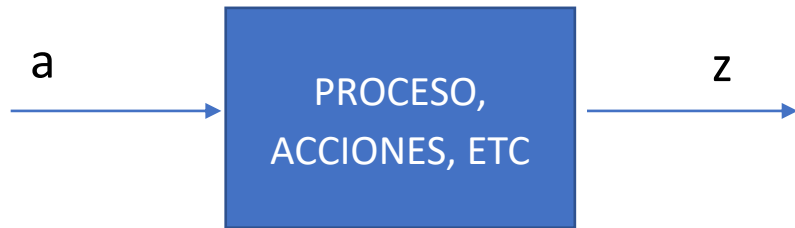
# Funciones → Modularizar

- Dominar el uso de funciones es de gran importancia, permiten modularizar nuestro código
- Separarlo según las tareas que requerimos, por ejemplo: Una función para abrir, otra para cerrar, otra para actualizar, etc.
- Básicamente una función en nuestro código debe contener la implementación de una utilidad de nuestra aplicación, es decir que por cada utilidad básica (abrir, cerrar, cargar, mover, etc.) sería adecuado tener al menos una función asociada a ésta.

# Programación Modular

- Un programa modular consta de un programa principal y uno o varios subprogramas. Los subprogramas son programas o módulos independientes que resuelven una tarea específica.
- La comunicación entre módulos se realiza a través de parámetros que contienen los datos que se desean introducir al módulo llamado.

# Declarando funciones en C++



```
Tipo nombreFuncion(tipo nombreArgumento)
{
    /*
        * bloque de instrucciones de la función
    */
    return valor_de_salida;
}
```

```
Tipo nombreFuncion(tipo nombreArgumento1, tipo nombreArg2, tipo nombreArg3)
{
    /*
        * bloque de instrucciones de la función
    */
    return valor_de_salida;
}
```

# Ejemplo 1

- Haga un programa principal que solicite un número entero y diga si un número es mayor o menor que cero, y a la vez si es par o impar.
  - Haga una función que retorne una variable que indique si un número entero es cero, mayor que cero o menor que cero.
  - Haga una función que retorne una variable que indique si un número es par o impar.



## Ejemplo 2.

Realice una función que se utilice para calcular el factorial de un número.

# Recursividad

- Recursividad es la propiedad que tiene un programa (lenguaje) de permitir que un programa se llame a sí mismo.
- Ejemplo:
- Recordemos la característica de factorial:

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

# Argumentos de entrada del main

- Los argumentos para main permiten realizar un práctico análisis de línea de comandos de los argumentos. El lenguaje define los tipos **argc** y **argv**. Los nombres argc y argv son tradicionales, pero se les puede asignar el nombre que se quiera.
- Las definiciones de los argumentos son las siguientes:
- **argc**
  - Un entero que contiene el número de argumentos que aparecen detrás de argv. El parámetro argc es siempre mayor o igual que 1.
- **argv**
  - Una matriz de cadenas terminadas en null que representan los argumentos de la línea de comandos especificados por el usuario del programa. Por convención, argv[0] es el comando con el que se invoca el programa. argv[1] es el primer argumento de la línea de comandos. El último argumento de la línea de comandos es argv[argc - 1], y argv[argc] siempre es NULL.

# Ejemplo del main con argumentos de entrada

```
#include <iostream>
using namespace std;
int main (int argc, char *argv[])
{
    for (int i = 0; i<argc; i++)
    {
        cout << argv[i] << "\n";
    }
    return 0;
}
```

# Punteros

- Los punteros en C++ (o apuntadores) son quizá uno de los temas que más confusión causan al momento de aprender a programar en C++.
- Sin embargo no es para tanto y que todo depende de dos elementos: el signo & (ampersand) y el \* (asterisco)
- Útiles para el uso de estructuras dinámicas


# Ejemplo

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

int main(int argc, char *argv[])
{
    int x;
    int *apuntador = &x; //Creamos un apuntador a la memoria de x
    cout << "Ingrese un numero entero: ";
    cin >> *apuntador; // Almacenamos en la memoria el dato

    delete [] apuntador; //Después de operar con punteros es necesario liberar la memoria.
    apuntador = NULL;

    cout << "Usted ingreso el numero: " << x << "\n";
    return 0;
}
```




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

int main(int argc, char *argv[])
{
    int x;
    int *apuntador = &x; //Creamos un apuntador a la memoria de x
    cout << "Ingrese un numero entero: ";
    cin >> *apuntador; // Almacenamos en la memoria el dato

    cout << "Usted ingreso el numero: " << x << "\n";
    cout << "Usted ingreso el numero: " << &apuntador << "\n";

    return 0;
}
```



# Detalles al crear y usar punteros en C++

- El tipo de dato del apuntador debe coincidir con el de la variable cuya posición en memoria apuntan. En el ejemplo vemos que tanto variable como apuntador son enteros.
- Siempre que queremos usar el apuntador debemos anteponer el asterisco (\*) para indicar que usaremos el valor en la posición de memoria apuntada.
- De no usar el asterisco el comportamiento sería impredecible. Estaremos haciendo uso de la dirección de memoria más no del valor almacenado en ésta.
- Después de usar un puntero, especialmente si trabajamos con arreglos o matrices, es MUY recomendable liberar la memoria utilizada con la función delete (tal como en el ejemplo)
- Un puntero o apuntador puede ser de cualquier tipo de dato, inclusive los podemos usar con tipos complejos.



# Los punteros y el ampersand

- El ampersand es un operador de C++ y es comúnmente utilizado para los punteros.
- Este operador nos permite obtener la dirección de memoria de una variable cualquiera y es justo esto (la dirección en memoria) lo que utilizan los punteros para referenciar valores.

# Apuntadores y el asterisco


- El asterisco es, por decirlo de alguna forma, el operador por excelencia de los punteros.
- Su utilidad radica en que, si el valor de dicho apuntador corresponde a una dirección de memoria, el asterisco nos permite resolverla y acceder al valor almacenado allí.
- Viéndolo desde otro enfoque, un apuntador es únicamente una dirección de memoria (un número) y el asterisco es el que hace la magia de obtener el valor referenciado por dicha dirección.

# Ámbito en C++

- Cuando se declara un **elemento** de programa como una clase, función o variable, su nombre solo se puede "ver" y usar en determinadas partes del programa.
- El contexto en el que se ve un nombre se denomina **ámbito**.
- Por ejemplo, si declara una variable x dentro de una función, x solo es visible dentro de ese cuerpo de la función. Tiene ámbito local.
- Es posible que tenga otras variables con el mismo nombre en el programa; siempre que estén en distintos ámbitos, no infringen la regla de definición única y no se genera ningún error.
- Para las variables no estáticas automáticas, el ámbito también determina cuándo se crean y destruyen en la memoria del programa.



# Ámbito en C++

- **Ámbito global**
    - Un nombre global es uno que se declara fuera de cualquier clase, función o espacio de nombres.
    - Sin embargo, en C++ incluso estos nombres existen con un espacio de nombres global implícito.
    - El ámbito de los nombres globales se extiende desde el punto de declaración hasta el final del archivo en el que se declaran. En el caso de los nombres globales, la visibilidad también se rige por las reglas de vinculación que determinan si el nombre es visible en otros archivos del programa.
  - **Ámbito del espacio de nombres (namespace)**
    - Un nombre que se declara dentro de un espacio de nombres, fuera de cualquier clase o definición de enumeración o bloque de función, es visible desde su punto de declaración hasta el final del espacio de nombres.
    - Un espacio de nombres se puede definir en varios bloques en distintos archivos.
- 




# Ámbito en C++

- **Ámbito local**


- Un nombre declarado dentro de una función o lambda, incluidos los nombres de parámetro, tienen ámbito local.
- A menudo se conocen como "locales".
- Solo son visibles desde su punto de declaración hasta el final de la función o cuerpo lambda.

- **Ámbito de clase**

- Los nombres de los miembros de clase tienen ámbito de clase, que se extiende a lo largo de la definición de clase independientemente del punto de declaración.
  - La accesibilidad de los miembros de clase se controla aún más mediante las palabras clave public, private y protected.
  - Solo se puede acceder a los miembros públicos o protegidos mediante los operadores de selección de miembros (., o ->) o operadores de puntero a miembro (.\* o ->\*).
- 



# Ámbito en C++

- **Ámbito de instrucción**
  - Los nombres declarados en una instrucción for, if, while o switch son visibles hasta el final del bloque de instrucciones.
  - **Ámbito de función**
  - Una etiqueta tiene ámbito de función, lo que significa que es visible a lo largo de un cuerpo de función incluso antes de su punto de declaración.
  - El ámbito de función permite escribir instrucciones como goto cleanup antes de declarar la etiqueta cleanup.
- 

# Ejercicio

- Cree un ejercicio que diga si una persona es mayor de edad con solo ingresar su edad. Utilizando punteros

