Introducción a los Lenguajes de Programación en Ciencias
C y C++

Henry R. Moncada

Universidad Nacional del Callao
Facultad de Ciencias Naturales y Matemáticas

5 de septiembre de 2025

Contenido

1. Objetivos
2. Un poco de historia
3. El Programa
4. El Entorno de Desarrollo en C y C++
5. Comentarios
6. Paso de Parámetros por Valor y por Referencia
7. Struct
8. Programación Orientada a Objetos
9. Standard Template Library
10. Asignación de Memoria Dinámica
11. Introducción a la Asignación de Memoria
12. Conclusión
13. Asignación de Memoria Dinámica
14. Conclusión

Objetivos

- Objetivos del tema:
  - Comprender la utilidad de las tres estructuras de control (secuencial, alternativa y repetitiva) en el paradigma de la programación estructurada.
  - Aprender a utilizar las sentencias asociadas a las estructuras de control y los algoritmos fundamentales basados en ellas.
  - Identificar distintos tipos de problemas y desarrollar soluciones algorítmicas para resolverlos.
  - Aplicar estos conocimientos en el lenguaje C de manera estructurada.

Un poco de historia

C es un **lenguaje de propósito general**, es decir, se pueden desarrollar aplicaciones de diversas áreas. Dentro de sus **principales características** podemos mencionar que:

- Es un **lenguaje estructurado**, tiene una abundante cantidad de operadores y tipos de datos.
- Es un **lenguaje de nivel medio**, pero se puede codificar a alto nivel, produce código objeto altamente optimizado, posee punteros y capacidad de aritmética de direcciones.
- C fue creado en los Laboratorios Bell de AT&T para correr originalmente en el sistema operativo Unix

Ventajas y desventajas de trabajar en C

Entre sus **múltiples ventajas**, podemos mencionar que:

- **C es un lenguaje altamente portable**, es decir, es independiente de la arquitectura de la máquina y, con pocas o ninguna modificación, un programa puede ejecutarse en una amplia variedad de computadoras.
- Es relativamente **flexible en la conversión de datos**.
- **Su eficiencia y claridad** han relegado el ensamblador casi al olvido en **Unix**. Porque, seamos sinceros, escribir en ensamblador es un ejercicio de paciencia que pocos quieren practicar.
- Posee un **compilador compacto pero poderoso**, gracias a su amplio conjunto de bibliotecas.

Sin embargo, también presenta **algunas desventajas**, entre ellas:

- La excesiva libertad en la escritura del código fuente puede llevar a cometer errores de programación que, al ser sintácticamente correctos, no se detectan en tiempo de compilación.
- Carece de instrucciones nativas para entrada y salida, así como para el manejo de cadenas de caracteres (strings), delegando estas funciones a las bibliotecas, lo que puede generar problemas de portabilidad.

Lenguaje de programación

- Un lenguaje de programación es un **idioma artificial** diseñado para **expresar computaciones** que pueden ser llevadas a cabo por máquinas como las computadoras.
- Pueden usarse para crear programas que controlen el comportamiento físico y lógico de una máquina, esto permite expresar algoritmos con precisión e interacción humano-maquina.
- Está formado de un conjunto de símbolos y reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos y expresiones.

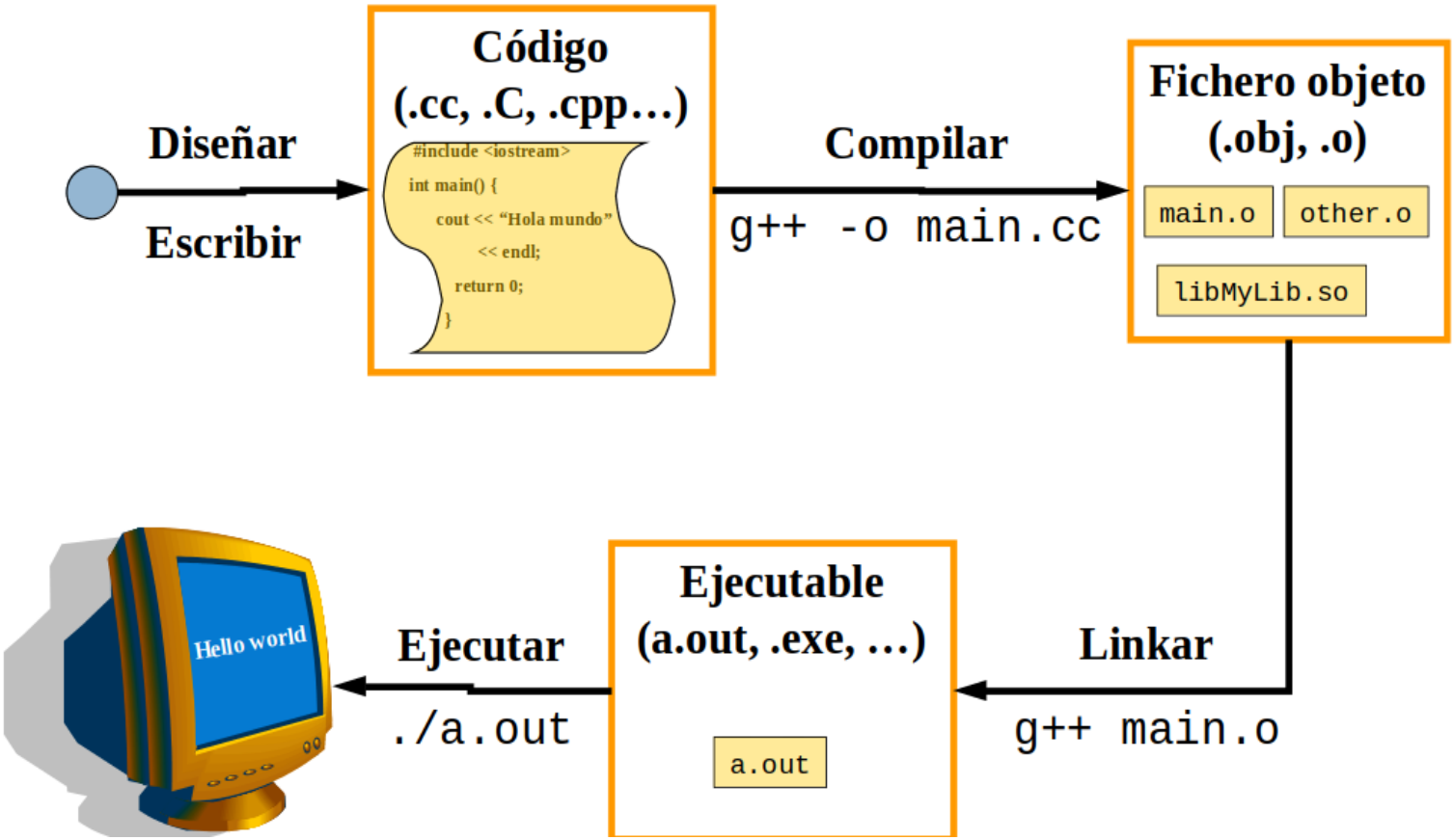
Tipos de Ficheros en C

Para crear un programa en C, primero se escribe el código fuente (el programa), luego se compila y, finalmente, se enlaza con las bibliotecas (o como diríamos en el argot técnico: **se linkea**, porque, claro, el español no tenía suficientes palabras).

- **Fuente (.c y .cpp):** Ficheros de texto en formato ASCII que contienen las instrucciones del programa.
- **Objeto (.o o .obj):** Archivos intermedios generados durante la compilación, entendibles por el ensamblador pero transparentes para el programador.
- **Librería (.a o .lib):** Conjunto de ficheros objeto agrupados en un solo archivo, generalmente utilizados para funciones estándar.
- **Cabecera (.h):** Contienen definiciones y declaraciones compartidas entre múltiples archivos fuente, además de esas funciones estándar que todo el mundo usa pero que nadie recuerda de memoria.
- **Ejecutables (.exe):** Aquí es donde todo el código máquina de los archivos objeto se une en un solo archivo, listo para ejecutarse o para lanzar errores misteriosos en tiempo de ejecución.

El Proceso de la Compilación

Un compilador es un programa que lee un programa escrito en un lenguaje de programación, el programa fuente, y lo traduce a un programa equivalente en otro lenguaje, el programa objeto.

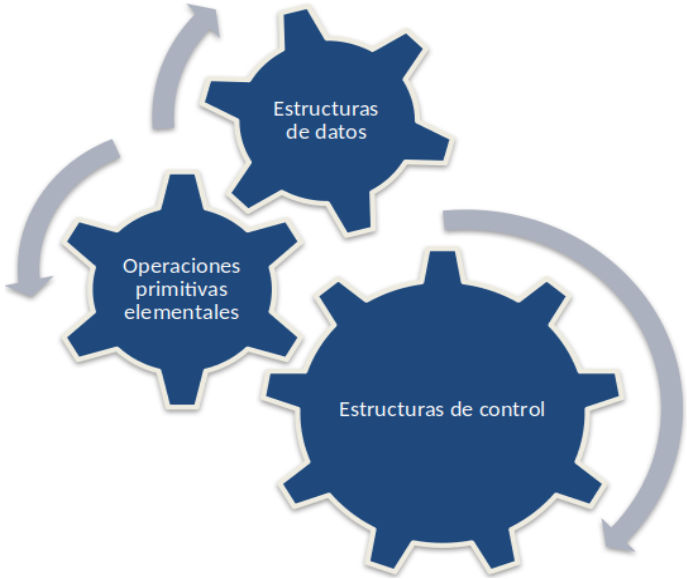


Tipos de Errores

- **Errores sintácticos**
  - Los lenguajes de programación tienen una sintaxis determinada para que puedan ser interpretados por el compilador.
  - El compilador detecta estos defectos de forma y muestra este tipo de errores.
  - Ej.: Dejarse un punto y coma al acabar una instrucción.
- **Errores en el enlace**
  - Se suele tratar de errores a la hora de nombrar las funciones, en los tipos o número de parámetros o del lugar donde se encuentran al llamar a una función...
- **Errores en ejecución**
  - Estos errores se dan porque en la ejecución de los programas ciertos valores pueden ser ilegales para ciertas operaciones.
  - Ejs.: División por cero, la raíz cuadrada de un valor negativo...
- **Errores semánticos**
  - Son los más difíciles de detectar y el entorno no puede ayudarnos, puesto que se tratan de discrepancias entre lo que hace el programa y lo que se pretende que haga.

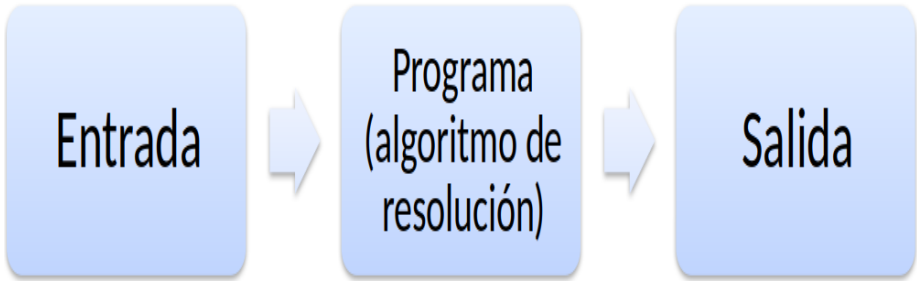
El Programa

- **Un programa** es un conjunto de instrucciones -**órdenes dadas al computador**- que guían al computador para realizar alguna actividad o resolver un problema.



Partes constitutivas de un programa

- El programador debe establecer el conjunto de especificaciones que debe contener el programa: entrada, salida y algoritmos de resolución, que incluirán las técnicas para obtener las salidas a partir de las entradas.



Programas Ejemplo 1

Primer Programa en C : El código fuente del programa **example_1_hola_mundo.c**

- ¡Ojo! Hay que usar la extensión ***.c** (minúscula)

```c
#include <stdio.h>

int main() {
    // Imprime el mensaje "Hola, mundo" en la consola
    printf("Hola, mundo\n");
    return 0;
}
```

- **#include <stdio.h>** Esta línea incluye la biblioteca estándar de entrada/salida **(stdio.h)**, que es necesaria para usar la función printf.
- **int main()** La función main es el punto de entrada de cualquier programa en **C**. El tipo de retorno int indica que esta función devolverá un valor entero al sistema operativo cuando termine (generalmente **0** para indicar éxito).
- **printf("Hola, mundo\n");** La función **printf** se utiliza para imprimir texto en la consola.
  - \n es un carácter de escape que agrega una nueva línea después del mensaje.
- **return 0;** Indica que el programa ha terminado correctamente. Un valor de retorno de **0** generalmente significa éxito.
- **Compila el programa** usando un compilador de C como gcc

```
$ gcc example_1_hola_mundo.c -o hola_mundo
```

- **Ejecuta el programa**

```
$ ./hola_mundo
```

Programas Ejemplo 2

Segundo Programa en C : El programa imprime el tamaño en bytes de varios tipos de datos básicos en C, código fuente del programa **example_2_print_bytes_size.c**

- El operador **sizeof** en C es una herramienta muy útil que permite determinar el tamaño en bytes de un tipo de dato o de una variable en tiempo de compilación. Es especialmente útil para comprender cómo se almacenan los datos en memoria y para escribir código más portátil.

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

int main() {
    printf("\nTamano en bytes de los tipos basicos:\n");
    printf(" char: %zu\n", sizeof(char));
    printf(" short int: %zu\n", sizeof(short int));
    printf(" int: %zu\n", sizeof(int));
    printf(" long int: %zu\n", sizeof(long int));
    printf(" float: %zu\n", sizeof(float));
    printf(" double: %zu\n", sizeof(double));
    printf(" long double: %zu\n", sizeof(long double));
    return 0;
}
```

Operador sizeof

- **Usar %zu :** %zu es el especificador de formato correcto para imprimir valores de tipo **size_t**, que es el tipo devuelto por **sizeof**.
- **Usar %d :** puede causar problemas en algunos sistemas, especialmente si **size_t** no tiene el mismo tamaño que un **int**.

Comentarios

En programación C, los comentarios de línea única se crean con `//` y continúan hasta el final de la línea, mientras que los comentarios de múltiples líneas se delimitan por `/*` y `*/`.

- Un comentario es una cadena de caracteres que no es tomada en cuenta por el compilador, esta va dentro de `/* y */`,
- **Comentarios de una sola línea:** Se inician con // y terminan al final de la línea. Ejemplo.

```
1 // Este es un comentario de una linea.
```

Comentarios una linea

- **Comentarios de múltiples líneas:** Se inician con `/*` y terminan con `*/`, Permiten comentarios que abarcan varias líneas, Ejemplo.

```
1 /*
2     Este es un comentario
3     de varias lineas.
4 */
```

Comentarios varias lineas

Paso de Parámetros por Valor y por Referencia

Paso de Parámetros por Valor

- Cuando pasas un valor por valor, la función recibe una copia del valor de la variable original **(Se pasa una copia del valor a la función)**.
- Cualquier modificación que hagas a la variable dentro de la función no afectará a la variable original en el código que llama a la función **(Los cambios dentro de la función NO afectan la variable original)**.

Ejemplo:

```c
#include <stdio.h>
void cambiarValor(int x) {
    x = 10; // Modifica la copia, no la original
}
int main() {
    int a = 5;
    printf("X = %d\n", x); // Imprime 5
    cambiarValor(a);
    printf("a = %d\n", a); // Imprime 5, a sigue siendo 5
    return 0;
}
```

Ejemplo de Paso por Valor

Paso de Parámetros por Referencia

- Cuando pasas un valor por referencia (usando punteros), la función recibe la dirección de memoria de la variable original **(Se pasa la dirección de memoria de la variable a la función)**.
- Cualquier modificación a la variable dentro de la función afectará a la variable original en el código que llama a la función **(Los cambios dentro de la función SÍ afectan la variable original)**.

Ejemplo:

```c
#include <stdio.h>
void cambiarValor(int *x) {
    *x = 10; // Modifica el valor en la direccion de memoria
}
int main() {
    int a = 5;
    printf("a = %d\n", a); // Imprime 5
    cambiarValor(&a);      // Pasamos la direccion de x
    printf("a = %d\n", a); // Imprime 10
    return 0;
}
```

Ejemplo de Paso por Referencia

STRUCT

Estructura en C

¿Qué es un `struct`?

Un **`struct`** en C es una estructura que permite agrupar variables de diferentes tipos bajo un mismo nombre.

Características:

- En C, las estructuras (structs) permiten agrupar diferentes tipos de datos en una sola entidad.
- Se pueden pasar a funciones por valor o por referencia, con diferentes implicaciones en rendimiento y modificación de datos.
- Es útil para representar objetos o entidades complejas (por ejemplo, estudiantes, productos, etc.).
- Los miembros de un **`struct`** pueden ser de diferentes tipos de datos.

Sintaxis básica:

```c
struct NombreStruct {
    tipoDato1 miembro1;
    tipoDato2 miembro2;
    // ...
};
```

Sintaxis básica

Ejemplo:

```c
#include <stdio.h>
struct Persona {
    char nombre[50];
    int edad;
};
```

Definicion de una estructura

Paso de Structs por Valor y por Referencia

Paso de Structs por Valor

- Se copia toda la estructura en la función.
- No se modifican los valores originales fuera de la función.

Ejemplo:

```c
#include <stdio.h>
struct Persona {
    char nombre[50];
    int edad;
};
void mostrarPersona(struct Persona p) {
    p.edad += 1;
    printf("2. Nombre: %s, Edad: %d\n",
      p.nombre, p.edad);
}
int main() {
    struct Persona p1 = {"Juan", 25};
    printf("1. Nombre: %s, Edad: %d\n",
      p1.nombre, p1.edad);
    mostrarPersona(p1);
    printf("3. Nombre: %s, Edad: %d\n",
      p1.nombre, p1.edad);
    return 0;
}
```

Paso por Valor

Paso de Structs por Referencia

- Se pasa la dirección de memoria de la estructura.
- Se pueden modificar los valores originales dentro de la función.

Ejemplo:

```c
#include <stdio.h>
struct Persona {
    char nombre[50];
    int edad;
};
void modificarEdad(struct Persona *p) {
    p->edad += 1;
    printf("2. Nombre: %s, Edad: %d\n",
      p->nombre, p->edad);
}

int main() {
    struct Persona p1 = {"Juan", 25};
    printf("1. Nombre: %s, Edad: %d\n",
      p1.nombre, p1.edad);
    modificarEdad(&p1);
    printf("3. Nombre: %s, Edad: %d\n",
      p1.nombre, p1.edad);
    return 0;
}
```

Paso por Referencia

Comparación de Métodos

Paso por Valor	**Paso por Referencia**
Se copia la estructura completa	Se pasa la dirección de memoria
No modifica la variable original	**Si Modifica** la variable original
Puede ser ineficiente si la estructura es grande	Más eficiente en estructuras grandes

Ejemplo 1: Representando un punto en 2D

Supongamos que queremos representar un punto en un plano cartesiano con coordenadas x y y.

```c
#include <stdio.h>

struct Punto {
    int x;
    int y;
};

int main() {
    struct Punto p1;
    p1.x = 3;
    p1.y = 5;

    printf("Coordenadas: (%d, %d)\n", p1.x, p1.y);
    return 0;
}
```

Estructura para un punto en 2D

Salida:

```
Coordenadas: (3, 5)
```

Ejemplo 2: Calculando la distancia entre dos puntos

Podemos usar una estructura para representar puntos y una función para calcular la distancia entre ellos.

```c
#include <stdio.h>
#include <math.h>

struct Punto {
    float x;
    float y;
};

float distancia(struct Punto p1, struct Punto p2) {
    return sqrt(pow(p2.x - p1.x, 2) + pow(p2.y - p1.y, 2));
}

int main() {
    struct Punto p1 = {1.0, 2.0};
    struct Punto p2 = {4.0, 6.0};

    printf("Distancia: %.2f\n", distancia(p1, p2));
    return 0;
}
```

Distancia entre dos puntos

Salida:

```
Distancia: 5.00
```

Ejemplo 3: Representando un estudiante

Podemos usar estructuras anidadas para representar datos más complejos, como un estudiante con su información personal.

```c
#include <stdio.h>
#include <string.h>

struct Fecha {
    int dia;
    int mes;
    int ano;
};

struct Estudiante {
    char nombre[50];
    int edad;
    struct Fecha fechaNacimiento;
};

int main() {
    struct Estudiante e1;
    strcpy(e1.nombre, "Juan Perez");
    e1.edad = 20;
    e1.fechaNacimiento.dia = 15;
    e1.fechaNacimiento.mes = 8;
    e1.fechaNacimiento.ano = 2003;

    printf("Nombre: %s\n", e1.nombre);
    printf("Edad: %d\n", e1.edad);
    printf("Fecha de nacimiento: %d/%d/%d\n",
           e1.fechaNacimiento.dia,
           e1.fechaNacimiento.mes,
           e1.fechaNacimiento.ano);
    return 0;
}
```

Estructura anidada para un estudiante

Salida:

```
Nombre: Juan Perez
Edad: 20
Fecha de nacimiento: 15/8/2003
```

Struct Resumen:

- Las estructuras (`struct`) permiten agrupar datos relacionados.
- Son útiles para modelar entidades complejas.
- Se pueden combinar con funciones y anidar para crear programas más organizados.

Aplicaciones:

- Representación de objetos en videojuegos.
- Gestión de bases de datos simples.
- Modelado de sistemas reales (por ejemplo, bancos, inventarios).

Review

- El C++ es un superconjunto del C
  - Soporta tanto metodologías de programación estructurada como OOP
  - Tiene la capacidad de usar librerías C y FORTRAN
- Características ajenas a la OOP y al C:
  - Chequeo de tipos mejorado (más estricto)
  - Constantes simbólicas (const) (chequeo de tipos constantes)
  - Sustitución de funciones inline (eficiencia de ejecución)
  - Argumentos por defecto (ahorro de código)
  - Sobrecarga de funciones y operadores (los tipos derivados tienen sintaxis iguales a los nativos)
  - Manejo de memoria dinámica
  - El tipo referencia (alias)

Programación Orientada a Objetos
(POO , OOP)

¿Qué es la Programación Orientada a Objetos?

- **La Programación Orientada a Objetos (POO , OOP)** es un paradigma de programación que usa objetos (los cuales son instancias de clases) y sus interacciones para diseñar aplicaciones y programas de computadora. Está basado en varias técnicas, incluyendo herencia, modularidad, polimorfismo y encapsulamiento

Conceptos fundamentales:

- **Clase:** Define la estructura y comportamiento de un objeto.
- **Objeto:** Es una instancia de una clase.
- **Herencia:** Permite que una clase derive de otra, reutilizando y extendiendo su funcionalidad.
- **Polimorfismo:** Permite que un objeto se comporte de diferentes maneras según su tipo.

Ejemplo 1: Creando una clase simple

Definamos una clase llamada **Persona** que tenga atributos como nombre y edad, y métodos para mostrar información.

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

class Persona {
public:
    string nombre;
    int edad;

    void mostrarInfo() {
        cout << "Nombre: " << nombre << ", Edad: "
     << edad << endl;
    }
};

int main() {
    Persona p1;
    p1.nombre = "Juan";
    p1.edad = 25;
    p1.mostrarInfo();
    return 0;
}
```

Clase básica en C++

Salida:

```
Nombre: Juan, Edad: 25
```

Ejemplo 2: Usando herencia para extender clases

La herencia permite crear una nueva clase (**Empleado**) que deriva de una clase existente (**Persona**).

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

class Persona {
public:
    string nombre;
    int edad;

    void mostrarInfo() {
        cout << "Nombre: " << nombre << ", Edad: "
     << edad << endl;
    }
};

class Empleado : public Persona {
public:
    string puesto;

    void mostrarPuesto() {
        cout << "Puesto: " << puesto << endl;
    }
};
```

Herencia en C++

```cpp
int main() {
    Empleado e1;
    e1.nombre = "Ana";
    e1.edad = 30;
    e1.puesto = "Gerente";

    e1.mostrarInfo();
    e1.mostrarPuesto();
    return 0;
}
```

Polimorfismo en C++

Salida:

```
Nombre: Ana, Edad: 30
Puesto: Gerente
```

Ejemplo 3: Implementando polimorfismo con funciones virtuales

El polimorfismo permite que una función se comporte de manera diferente según el tipo de objeto que la invoque.

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

class Animal {
public:
    virtual void hacerSonido() {
        cout << "Este animal hace un sonido." << endl;
    }
};

class Perro : public Animal {
public:
    void hacerSonido() override {
        cout << "Guau guau!" << endl;
    }
};

class Gato : public Animal {
public:
    void hacerSonido() override {
        cout << "Miau miau!" << endl;
    }
};
```

Polimorfismo en C++

```cpp
int main() {
    Animal* animales[2];
    animales[0] = new Perro();
    animales[1] = new Gato();

    for (int i = 0; i < 2; i++) {
        animales[i]->hacerSonido();
    }

    delete animales[0];
    delete animales[1];
    return 0;
}
```

Polimorfismo en C++

Salida:

```
Guau guau!
Miau miau!
```

Resumen POO:

Resumen:

- POO es un paradigma poderoso que organiza el código en torno a objetos.
- Las clases definen la estructura y comportamiento de los objetos.
- La herencia permite reutilizar y extender funcionalidades.
- El polimorfismo permite que los objetos se comporten de diferentes maneras.

Aplicaciones:

- Desarrollo de software modular y escalable.
- Modelado de sistemas complejos (videojuegos, simulaciones, etc.).
- Mejora la legibilidad y mantenibilidad del código.

Standard Template Library
(STL)

¿Qué es la Standard Template Library (STL)?

La STL es una biblioteca estándar de C++ que proporciona componentes reutilizables para manejar datos de manera eficiente.

Componentes principales:

- **Contenedores:** Almacenan colecciones de datos (por ejemplo, vectores, listas, mapas).
- **Iteradores:** Permiten recorrer los elementos de un contenedor.
- **Algoritmos:** Proporcionan funciones para manipular datos (por ejemplo, ordenar, buscar).

Ventajas:

- Código más limpio y reutilizable.
- Optimización del rendimiento.
- Reducción de errores comunes.

Ejemplo 1: Trabajando con `vector`

El contenedor **`vector`** es un arreglo dinámico que puede crecer o reducirse según sea necesario.

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

int main() {
    vector<int> numeros = {10, 20, 30, 40, 50};

    // Agregar un elemento
    numeros.push_back(60);

    // Recorrer el vector
    cout << "Elementos del vector:" << endl;
    for (int num : numeros) {
        cout << num << " ";
    }
    cout << endl;

    // Acceder a un elemento
    cout << "Primer elemento: " << numeros.front()
     << endl;
    cout << "Ultimo elemento: " << numeros.back()
     << endl;

    return 0;
}
```

Uso de vector

Salida:

```
Elementos del vector:
10 20 30 40 50 60
Primer elemento: 10
Ultimo elemento: 60
```

Ejemplo 2: Trabajando con `map`

El contenedor `map` almacena pares clave-valor de manera ordenada.

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

int main() {
    map<string, int> edades;
    edades["Juan"] = 25;
    edades["Ana"] = 30;
    edades["Carlos"] = 22;

    // Recorrer el mapa
    cout << "Edades:" << endl;
    for (const auto& par : edades) {
        cout << par.first << ": " << par.second << endl;
    }

    // Acceder a un valor
    cout << "Edad de Ana: " << edades["Ana"] << endl;

    return 0;
}
```

Uso de map

Salida:

```
Edades:
Ana: 30
Carlos: 22
Juan: 25
Edad de Ana: 30
```

Ejemplo 3: Usando algoritmos STL

Los algoritmos de la STL permiten realizar operaciones comunes como ordenar, buscar o eliminar elementos.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector<int> numeros = {5, 2, 9, 1, 5, 6};

    // Ordenar el vector
    sort(numeros.begin(), numeros.end());

    // Mostrar el vector ordenado
    cout << "Vector ordenado:" << endl;
    for (int num : numeros) {
        cout << num << " ";
    }
    cout << endl;

    // Buscar un elemento
    if (binary_search(numeros.begin(), numeros.end
     (), 5)) {
        cout << "El numero 5 esta en el vector." <<
      endl;
    } else {
        cout << "El numero 5 no esta en el vector."
      << endl;
    }

    return 0;
}
```

Uso de algoritmos STL

Salida:

```
Vector ordenado:
1 2 5 5 6 9
El numero 5 esta en el vector.
```

Resumen STL:

Resumen:

- La STL proporciona herramientas poderosas para manejar datos de manera eficiente.
- Los contenedores (`vector`, `map`, etc.) simplifican el almacenamiento y acceso a datos.
- Los algoritmos (`sort`, `binary_search`, etc.) facilitan operaciones comunes.

Aplicaciones:

- Desarrollo de software modular y escalable.
- Manipulación eficiente de grandes volúmenes de datos.
- Mejora la legibilidad y mantenibilidad del código.

ASIGNACIÓN DE MEMORIA DINÁMICA
C++ MEMORY MANAGEMENT

¿Qué es la asignación de memoria dinámica?

La asignación de memoria dinámica permite reservar y liberar memoria durante la ejecución del programa. Esto es útil cuando no se conoce el tamaño de los datos en tiempo de compilación.

Funciones principales en C:

- **malloc:** Reserva un bloque de memoria sin inicializar.

```
ptr = (castType*) malloc(size);
```

- **calloc:** Reserva y limpia (inicializa a cero) un bloque de memoria.

```
ptr = (castType*)calloc(n, size);
```

- **realloc:** Cambia el tamaño de un bloque de memoria previamente asignado.

```
ptr = realloc(ptr, x);
```

- **free:** Libera la memoria reservada.

```
free(ptr);
```

Operadores principales en C++:

- **new:** Reserva memoria dinámicamente.

```
data_type* pointer_variable = new data_type{value};
```

- **delete:** Libera memoria reservada con `new`.

```
delete pointer_variable;
```

Ejemplo 1: Reservando memoria con `malloc`

El siguiente ejemplo muestra cómo usar `malloc` para reservar memoria dinámicamente en C.

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

int main() {
    int n;
    printf("Ingrese el numero de elementos: ");
    scanf("%d", &n);

    // Reservar memoria para 'n' enteros
    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Error al asignar memoria.\n");
        return 1;
    }

    // Inicializar y mostrar los elementos
    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Liberar memoria
    free(arr);
    return 0;
}
```

Uso de malloc

Salida (para n = 5):

```
Ingrese el número de elementos: 5
1 2 3 4 5
```

Example 2: Allocating Memory with `calloc`

El siguiente ejemplo muestra cómo usar `calloc` para reservar memoria inicializada a cero en C.

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

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    // Allocate memory for 'n' integers and
     initialize to zero
    int *arr = (int *)calloc(n, sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    // Display elements (initialized to zero)
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Free memory
    free(arr);
    return 0;
}
```

Using calloc

Output (for n = 5):

```
Enter the number of elements: 5
0 0 0 0 0
```

Ejemplo 3: Redimensionando memoria con `realloc`

La función **`realloc`** permite cambiar el tamaño de un bloque de memoria previamente asignado.

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

int main() {
    int *arr = (int *)malloc(5 * sizeof(int));
    if (arr == NULL) {
        printf("Error al asignar memoria.\n");
        return 1;
    }

    // Inicializar el arreglo original
    for (int i = 0; i < 5; i++) {
        arr[i] = i + 1;
    }

    // Redimensionar el arreglo a 10 elementos
    arr = (int *)realloc(arr, 10 * sizeof(int));
    if (arr == NULL) {
        printf("Error al redimensionar memoria.\n");
        return 1;
    }

    // Inicializar los nuevos elementos
    for (int i = 5; i < 10; i++) {
        arr[i] = i + 1;
    }
```

Uso de realloc

```c
    // Mostrar todos los elementos
    for (int i = 0; i < 10; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Liberar memoria
    free(arr);
    return 0;
}
```

Uso de realloc

Salida:

```
1 2 3 4 5 6 7 8 9 10
```

Ejemplo 2: Reservando memoria con `new` en C++

En C++, los operadores **`new`** y **`delete`** son más seguros y fáciles de usar que las funciones de C.

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

int main() {
    int n;
    cout << "Ingrese el numero de elementos: ";
    cin >> n;

    // Reservar memoria para 'n' enteros
    int *arr = new int[n];

    // Inicializar y mostrar los elementos
    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
        cout << arr[i] << " ";
    }
    cout << endl;

    // Liberar memoria
    delete[] arr;
    return 0;
}
```

Uso de new y delete

Salida (para n = 5):

```
Ingrese el número de elementos: 5
1 2 3 4 5
```

Resumen memoria dinámica

Resumen:

- La asignación de memoria dinámica permite manejar datos cuyo tamaño no se conoce en tiempo de compilación.
- En C, se utilizan funciones como `malloc`, `calloc`, `realloc` y `free`.
- En C++, los operadores `new` y `delete` son más seguros y fáciles de usar.

Aplicaciones:

- Manejo de grandes volúmenes de datos.
- Implementación de estructuras de datos dinámicas (por ejemplo, listas, árboles).
- Optimización del uso de memoria en programas complejos.

Diferencias entre C y C++

El lenguaje **C** es un lenguaje de programación procedural desarrollado en la década de 1970 por Dennis Ritchie. Posteriormente, **C++** fue creado como una extensión de C por Bjarne Stroustrup en la década de 1980, añadiendo características orientadas a objetos y mejorando su funcionalidad. A continuación, se describen las principales diferencias entre ambos lenguajes.

1. Paradigma de Programación
   - **C**: Es un lenguaje *procedural*, lo que significa que se enfoca en funciones y procedimientos para resolver problemas.
   - **C++**: Es un lenguaje *multiparadigma*, que soporta tanto la programación procedural como la *programación orientada a objetos (POO)* y la programación genérica.
2. Soporte para Clases y Objetos
   - **C**: No tiene soporte para clases ni objetos. Todo el código se organiza en funciones y estructuras de datos básicas.
   - **C++**: Introduce el concepto de *clases* y *objetos*, permitiendo encapsular datos y funciones en una sola entidad.
3. Gestión de Memoria
   - **C**: La gestión de memoria es manual mediante funciones como `malloc` y `free`.
   - **C++**: Además de las funciones de C, introduce operadores como `new` y `delete` para la asignación y liberación dinámica de memoria.
4. Funciones y Sobrecarga
   - **C**: No permite la sobrecarga de funciones. Cada función debe tener un nombre único.
   - **C++**: Permite la *sobrecarga de funciones*, lo que significa que varias funciones pueden tener el mismo nombre pero diferentes parámetros.
5. Bibliotecas Estándar
   - **C**: Tiene una biblioteca estándar limitada (`stdio.h`, `stdlib.h`, etc.), enfocada principalmente en operaciones básicas.
   - **C++**: Incluye la *Standard Template Library (STL)*, que proporciona contenedores (como `vector`, `list`, `map`), algoritmos y utilidades avanzadas.

PUNTEROS (POINTERS)

¿Qué son los punteros?

Un **puntero** es una variable que almacena la dirección de memoria de otra variable.
Características principales:

- Permiten acceder y modificar directamente los valores almacenados en memoria.
- Son fundamentales para trabajar con estructuras dinámicas como listas, árboles y grafos.
- Facilitan la implementación de funciones que modifican datos sin necesidad de retornar valores.

Sintaxis básica:

```
1 int *puntero; // Declara un puntero a un entero
2 puntero = &variable; // Asigna la direccion de 'variable' al puntero
```

Ejemplo 1: Trabajando con punteros básicos

El siguiente ejemplo muestra cómo declarar, inicializar y usar un puntero para acceder al valor de una variable.

```c
#include <stdio.h>

int main() {
    int numero = 42;
    int *puntero = &numero; // Puntero apunta a 'numero'

    printf("Valor de numero: %d\n", numero);
    printf("Direccion de numero: %p\n", (void *)&numero);
    printf("Valor a traves del puntero: %d\n", *puntero);

    // Modificar el valor a traves del puntero
    *puntero = 100;
    printf("Nuevo valor de numero: %d\n", numero);

    return 0;
}
```

Uso básico de punteros

Salida:

```
Valor de numero: 42
Direccion de numero: 0x7ffee4b3c9ac
Valor a traves del puntero: 42
Nuevo valor de numero: 100
```

Ejemplo 2: Relación entre punteros y arreglos

En C y C++, los arreglos y los punteros están estrechamente relacionados. Un puntero puede usarse para recorrer un arreglo.

```c
#include <stdio.h>

int main() {
    int arreglo[5] = {10, 20, 30, 40, 50};
    int *puntero = arreglo; // Puntero apunta al primer elemento del arreglo

    // Recorrer el arreglo usando el puntero
    for (int i = 0; i < 5; i++) {
        printf("Elemento %d: %d\n", i, *(puntero + i));
    }

    return 0;
}
```

Punteros y arreglos

Salida:

```
Elemento 0: 10
Elemento 1: 20
Elemento 2: 30
Elemento 3: 40
Elemento 4: 50
```

Ejemplo 3: Usando punteros para llamar funciones

Un puntero a función permite almacenar la dirección de una función y llamarla dinámicamente. Esto es útil para implementar patrones como callbacks.

```c
#include <stdio.h>

// Funciones simples
void saludar() {
    printf("Hola, mundo!\n");
}

void despedirse() {
    printf("Adios, mundo!\n");
}

int main() {
    // Declarar un puntero a funcion
    void (*punteroFuncion)();

    // Asignar la direccion de una funcion
    punteroFuncion = saludar;
    punteroFuncion(); // Llama a 'saludar'

    punteroFuncion = despedirse;
    punteroFuncion(); // Llama a 'despedirse'

    return 0;
}
```

Punteros a funciones

Salida:

```
Hola, mundo!
Adios, mundo!
```

Conclusión

Resumen:

- Los punteros son variables que almacenan direcciones de memoria.
- Permiten acceder y modificar datos directamente en memoria.
- Son útiles para trabajar con arreglos, estructuras dinámicas y funciones.

Aplicaciones:

- Implementación de estructuras de datos dinámicas (listas, árboles, grafos).
- Optimización del uso de memoria en programas complejos.
- Programación avanzada con callbacks y funciones dinámicas.