C++ para Principiantes

Alex Dannis Lipa Quispe

2025-05-29

Table of contents

1 INTRODUCCIÓN

Bienvenio/a a este libro de "C++ Para Principiantes", darte la cálida vienvenida a este libro que fue echa especialmente para personas que a si como yo, teinen cierta dificultad entender y escribir código de manera fluida. Este es un pequeño paso para mi persona ya que será mi primer libro de programación y espero que pueda hacer muchos más.

Este libro tomará conceptos básicos sobre la programación, pero tocara varios temas importantes, empezando desde el hola mundo terminando en loq ue tenga que acabar… El libro tendrá explicación detallada sobre cada parte del código de una manera clara y fáscil de entender por lo que te pido que solo tengas los ánimos para poder aprender de una manera más divertida.

C++ es un lenguaje de programación de propósito general creado por **Bjarne Stroustrup** en los años 80, como una extensión del lenguaje C. Su objetivo era combinar la eficiencia y el control de C con características más modernas, como la **programación orientada a objetos**.

Desde entonces, C++ se ha convertido en uno de los lenguajes más utilizados del mundo, especialmente en áreas donde se necesita rendimiento, como:

- Desarrollo de **videojuegos**
- **Sistemas operativos**
- Programas que necesitan acceso directo a hardware
- **Software financiero** de alto rendimiento
- Aplicaciones **científicas y de ingeniería**

Características principales de C++

- **Compilado**: el código debe convertirse a lenguaje máquina antes de ejecutarse.
- **Tipado estático**: debes declarar el tipo de cada variable.
- **Eficiente y rápido**: ideal para tareas exigentes.
- **Multiplataforma**: puedes usarlo en Windows, Linux y macOS.
- **Paradigma múltiple**: permite programación estructurada, orientada a objetos y genérica.

2 Variables y Tipos de Datos

2.1 Tipos de datos

Los tipos de datos son una categoría que se asigna a las variables para que puedan almacenar un tipo de valor específico.

2.2 Variables

Las variables pueden tener cualquier nombre, excepto las **palabras reservadas** del lenguaje de programación que estés usando. Como buena práctica, te recomiendo que las variables sean fáciles de leer, como por ejemplo: `numero`, `nombre`.
Esto ayudará a que tu código sea fácil de entender en tu entorno de trabajo.

2.3 Datos numéricos

- **float**: ocupa 32 bits. Puede contener desde `1.4 × 10^-45` hasta `3.4028235 × 10^38`.
- **double**: ocupa 64 bits. Puede contener desde `4.9 × 10^-324` hasta `1.7976931348623157 × 10^308`.
- **short**: ocupa 16 bits. Puede contener desde `-32,768` hasta `32,767`.
- **int**: ocupa 32 bits. Puede contener desde `-2,147,483,648` hasta `2,147,483,647`.
- **long**: ocupa 64 bits. Puede contener desde `-9,223,372,036,854,775,808` hasta `9,223,372,036,854,775,8`

2.4 Cadenas de caracteres

- **char**: ocupa 8 bits. Puede contener desde el carácter nulo (`'\0'`) hasta `127` (para `signed char`) o `255` (para `unsigned char`).
- **string**: ocupa 8 bits por cada carácter. Puede contener desde `""` (cadena vacía) hasta un tamaño que depende de la memoria disponible.

2.5 Ejemplo de declaración de variables

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

int main() {
    // Variables numéricas
    int edad = 25;
    float altura = 1.75;
    double peso = 65.7;

    // Cadenas de texto
    char inicial = 'A';
    string nombre = "Carlos";

    // Mostrar los valores
    cout << "Nombre: " << nombre << endl;
    cout << "Inicial: " << inicial << endl;
    cout << "Edad: " << edad << endl;
    cout << "Altura: " << altura << " m" << endl;
    cout << "Peso: " << peso << " kg" << endl;

    return 0;
}
```

3 Operadores

3.1 Matemáticos

Son operadores que conocemos desde la primaria, los cuales son:

- Suma: denotado por (+).
- Resta: denotado por (-).
- Multiplicación: denotado por (*).
- División: denotado por (/).
- Residuo: denotado por (%).
- Potencia: denotado por (^).

Estos operadores nos sirven para que nuestros programas puedan ejecutar las operaciones aritméticas que conocemos, por ejemplo:

OPERADORES ARITMÉTICOS	OPERACIÓN	RESULTADO
Suma	8 + 9	17
Resta	10 - 9	1
Multiplicación	7 * 2	14
División	21 / 3	7
Residuo	21 % 2	1
Potencia	4 ^ 2	16

3.2 Relacionales

Son los signos que se usan para comparar dos valores, cuyo resultado es un valor booleano: **verdadero** o **falso**. Probablemente ya los conoces:

Operador	Símbolo	Operación	Resultado
Mayor que	>	4 > 7	Falso
Menor que	<	9 < 1	Falso
Mayor igual que	>=	7 >= 2	Verdadero
Menor igual que	<=	8 <= 8	Verdadero
Igual	==	7 == 1	Falso

Operador	Símbolo	Operación	Resultado
Diferente	!=	3 != 9	Verdadero

3.3 Lógicos

En matemáticas se conocen como tablas proposicionales. Aquí tomaremos los operadores **Y**, **O** y **Negación**.

Donde **V** es verdadero y **F** es falso.

Operador "Y"	Salida	Operador "O"	Salida	Negación "!"	Salida
V y V	V	V o V	V	!(V y F)	V
V y F	F	V o F	V	!(V y V)	F
F y V	F	F o V	V	!(F o V)	F
F y F	F	F o F	F	!(F o F)	V

3.4 Asignación

La asignación se usa en programación para dar un valor a una variable. Por ejemplo, si declaramos una variable entera llamada **a**:

```
int a;
```

Para asignarle un valor:

```
a = 5;
```

Ahora, si queremos restarle 1 a la variable, escribimos:

```
a = a - 1;  // Resultado: 4
```

Como el nombre de la variable se repite, podemos usar el operador compuesto:

```
a -= 1;     // Equivalente a: a = a - 1
```

De la misma manera, para sumar 1:

```
a = a + 1;  // Resultado: 6
```

O usando el operador compuesto:

```
a += 1;     // Equivalente a: a = a + 1
```

3.5 Operadores de incremento y decremento

Existen operadores que aumentan o disminuyen el valor en 1 de forma más sencilla:

Operador	Descripción	Ejemplo	Resultado si a = 5
`++a`	Incremento antes de usar el valor.	`++a;`	6
`a++`	Incremento después de usar el valor.	`a++;`	6 (pero el valor usado primero es 5)
`--a`	Decremento antes de usar el valor.	`--a;`	4
`a--`	Decremento después de usar el valor.	`a--;`	4 (pero el valor usado primero es 5)

4 Entradas y Salidas

4.1 Inicio:

Para empezar a escribir un código en C++ necesitas tener un lugar compilador o un lugar donde escribir código, eso te lo dejo a tu elección.

Primeramente, cada programa se inicia con:

```
#include <iostream>
```

La palabra **#include <iostream>** incluye la librería para poder hacer la entrada y salida de valores, los cuales son:

```
cin >> //esto se utiliza para entrar o digitar valores a tu programa.
cout << //esto se utiliza para mostrar mensaje en tu programa.
```

La estructura principal es de esta manera:

```
#include <iostream>
int main(){  //Esta es la función principal que leerá el programa


    return 0; //Si el programa funciona bien retornara 0
}
```

4.2 Salidas:

Para mostrar tu primer `hola mundo` en c++ se hace de la siguiente manera:

```
#include <iostream>
int main(){
    std::cout << "Hola mundo"<< std::endl;
    return 0;
}
```

Este fragmento de código imprimirá tu primer “Hola mundo”. Recuerda que para enviar un mensaje es importante que esté entre comillas dobles “ “, y si solo va a tener un carácter debe estar entre comillas simples ’ ’.

Para simplificar el código, si te resulta estresante, puedes usar **using namespace std**. Esto, de manera simple, permite abreviar u omitir el **std::**. El código quedará de la siguiente manera:

```
#include <iostream>
int main(){
    std::cout << "Hola mundo"<<std::endl;
    return 0;
}/* cabe recalcar que para proyectos grandes el "using namespace std"
ara tu programa menos optimo */
```

Los mensajes que pongo en los códigos no se leen porque son comentarios. Si deseas poner un comentario lineal, usa // y para comentarios multilínea debes usar / … / escribiendo el mensaje dentro de los asteriscos.

4.3 Entradas:

Para guardar y mostrar valores, para compilar el código hacemos esto:

```
#include <iostream>
int main(){
    int valor;
    std::cout << "Digite un valor: ";
    std:: cin >> valor;
    std::cout << "El valor es: "<<valor<<std::endl;
    return 0;
}
```

Este código lo que hace es declarar una variable de nombre **valor** de tipo entero (int) con el mensaje “Digite un valor”. Luego, en el programa, el usuario digitara un valor numérico que se almacenará en **valor**.

Para imprimir el valor introducido, se usa un `cout` seguido del nombre de la variable a mostrar. Cabe recalcar que si no se asigna un valor a la variable, se imprimirá un dato basura o un valor sin sentido.

```
Digite un valor: 15
El valor es: 15
```

Para hacer operaciones se hace de la siguiente manera:

```cpp
#include <iostream>
int main(){
    int num1 = 8, num2 = 4, suma, multiplicacion;
    suma = num1 + num2;  //en este caso res será igual a 12.
    multiplicacion = num1 * num2;
    // el endl es como un salto de linea.
    atd::cout << suma <<" || "<< multiplicacion<<std::endl;
    // tambien lo podemos poner de manera directa.
    std::cout << num1 - num2<<"  || "<< num1 / num2<<std::endl;
    return 0;
}
```

Hicimos las operaciones básicas que todos conocemos (+, -, *, /). El fragmento de código se mostrará de la siguiente manera:

```
12 || 32
4 || 2
```

Tienes que ser creativo para poder realizar cualquier tipo de operaciones aritméticas. Por ejemplo, realizaremos un ejercicio simple que es resolver esta ecuación: La fórmula es: $[a \times (\frac{b}{c})]$

Empecemos.

```cpp
#include <iostream>
int main(){
    float a,b,c, resultado;
    std::cout<<"Digite 3 numeros separados por espacios o saltos de linea: ";
    std::cin>>a>>b>>c;
    //En este caso primero se ejecuta la divisón y luego la multiplicaión.
    resultado = a*(b/c);
    std::cout << "El resultado es: "<<resultado<<std::endl;
    return 0;
}
```

Lo que hicimos es declarar 4 variables de tipo flotante para no perder los decimales del resultado. Ten en cuenta que **resultado** almacena el resultado y lo imprimimos como se muestra a continuación:

```
Digite 3 numeros separados por espacios o saltos de linea: 3 4 5
El resultado es: 12
```

Para finalizar es recomendable que tengas cuidado con las operaciones que agregues a tu programa.

5 Estructuras de Control

5.1 if-else

Una breve explicación sobre este tema es que cuando la condición dentro de un **if** sea verdadera, se ejecuta una parte del código. En cambio, si la condición es falsa, se ejecuta el código dentro de **else**.

La estructura principal es la siguiente:

```
#include <iostream>
using namespace std;
int main(){
    if(condición){
        acción  // solo entrara en esta parte del código se la condición es verdadera
    }
    else{
        acción // Este código se ejecuta si la afirmación es falsa
    }
    return 0;
}
```

Un ejemplo básico para entender aún más la condicional **if-else**.

```
#include <iostream>
using namespace std;
int main(){
    if(3>4){
        cout << "3 es mayor que 4."<<endl;
    }
    else{
        cout << "4 es mayor que 3."<< endl;
    }
    return 0;
}
```

En este ejemplo vemos que la afirmación de $\mathbf{3} > \mathbf{4}$ es falsa, por lo que se ejecuta el código que está dentro de **else**, como se muestra a continuación.

4 es mayor que 3.

Un ejercicio muy práctico es comprobar si un número es par o impar. En este código utilizaremos el operador **mod** (%) y el código quedará de la siguiente manera:

```
#include <iostream>
using namespace std;
int main(){
    int numero;
    cout << "Digite un numero entero: ";
    cin >> numero;
    if((numero % 2) == 0){
        cout << "El numero es par."<<endl;
    }
    else{
        cout << "El numero es impar."<<endl;
    }
    return 0;
}
```

En este ejercicio solicitamos al usuario que ingrese un número entero, el cual almacenamos en una variable de tipo entero llamada **numero**.

La lógica del programa se basa en la operación módulo (%), que calcula el **residuo** de una división:

```
flowchart LR
  A["Solicitar número (numero = entrada)"] --> B[numero = 4]
  B --> C{"numero % 2 == 0?"}
  C -->|Verdadero| D[Mostrar "El número es par"]
  C -->|Falso| E[Mostrar "El número es impar"]
```

Para mayor entendimiento, veamos el ejemplo de **4 % 2**, cuyo resultado es **0**, por lo que la afirmación queda así: **0 == 0**, la cual es verdadera, e imprimirá el fragmento de código que está dentro del **if**.

En cambio, si la entrada fuera 5, el cálculo sería: **(5 % 2) == 0 → 1 == 0**, que es falso, por lo que se ejecutará el código dentro del **else**.

Digite un numero entero: 4
El numero es par.

Digite un numero entero: 5
El numero es impar.

5.2 switch-case:

De una manera fácil de entender, esta estructura verifica una serie de posibles valores. Es decir, el usuario ingresa un dato y ese dato se compara con diferentes valores.

La estructura es la siguiente:

```
#include <iostream>
using namespace std;
int main(){
    switch(valor){
        case dato1: // si el valor es igual a dato1 ejecuta la acción.
            acción
            break; /* una vez realizada la accion la palabra reservada "break" ara
                      salir de la sentencia switch*/
        case dato2:
            acción
            break;
        case dato3:
            acción
            break;
        default:    // si ninguno de los opciones se cumple se ejecuta la accion
            accion
            break;
    }
    return 0;
}
```

Como se ve, es fácil de entender. Un ejemplo práctico será realizar un programa que, dado un valor de entrada, indique si una persona tiene que presentar un examen, no tiene que presentarlo o que ingrese un valor válido.

El programa quedaría así:

```
#include <iostream>
using namespace std;
int main(){
    char opcion; // la opción solo podrá contener un string 'a', 'b', etc;
    cout << "Diste el examen? (s/n): ";
    cin >> opcion; // solo podras ingresa si-> s ó no->n
    switch(opcion){
        case 's':
            cout << "Espera tu resultado."<<endl;
```

```
            break;
        case 'n':
            cout << "Dirigete al salon A3."<<endl;
            break;
        default:
            cout << "Ingresaste un valor invalido."<<endl;
            break;
    }
    return 0;
}
```

Diste el examen? (s/n): s
Espera tu resultado.

Lo que hace el programa es verificar si el carácter ingresado es `'s'`. En caso de ser así, imprime **"Espera tu resultado."**.
Si el carácter es `'n'`, imprime **"Dirígete al salón A3."**.
Y si no es ninguno de los dos, se imprimirá **"Ingresaste un valor inválido."**.

Con tu creatividad, sé que podrás crear programas con más opciones, como por ejemplo, una calculadora que, dada una opción, haga una suma, una resta o cualquier operación que desees.

5.3 Ternario:

El operador ternario es parecido al **if-else**, pero de manera resumida. Tiene esta estructura:

```
#include <iostream>
using namespace std;
int main(){
    (condicion) ? accion(si es verdadera) : acción si es falsa;
    return 0;
}
```

Si la condición es verdadera, se ejecuta el código que está después de `?`.
Si la condición es falsa, se ejecuta el código que está después de `:`.

Ejemplo:

```
#include <iostream>
using namespace std;
int main(){
    int a=5, b=6;
```

```
    int resultado;
    resultado = (a>b) ? a:b;
    cout << "El numero mayor es: " << resultado<<endl;
    return 0;
}
```

Primeramente, declaramos 3 variables enteras como se muestra. Ojo, el operador ternario retorna un valor de acuerdo a sus parámetros, que en este caso son **a** y **b**.

Primeramente compara los valores de la siguiente manera: `(a > b)?` — esta operación es como `5 > 6`, esta condición es falsa, por lo que retorna el código que está del lado derecho de `:` que es la **b**, dando como resultado:

```
El numero mayor es: 6
```

Puedes probrar de diferente manera, como en este otro ejemplo:

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

int main() {
    int a = 10, b = 20;
    string resultado;
    // Operador ternario
    resultado = (a > b) ? "Mayor." : "Menor o igual.";
    cout << resultado << endl;
    return 0;
}
```

En este caso retornamos una cadena de caracteres y, como sabemos, para poder guardar cadenas usamos la palabra reservada **string** seguida del nombre de la variable.

Como vemos en la salida, dará **Menor o igual**, porque el valor de **a** es menor que el valor de la variable **b**.

```
Menor o igual.
```

Puedes hacer una serie de ejercicios para tener más familiarización con las estructuras de control, para poder así desarrollar tu lógica de programación.

6 Bucles o Ciclos

Los **bucles** son una forma eficiente de repetir una serie de instrucciones sin tener que escribirlas múltiples veces.

Un ejemplo básico sería imprimir cinco veces el mensaje **"Hola mundo"**.

En este ejercicio, aprenderemos a hacerlo utilizando los diferentes tipos de **bucles o ciclos** que ofrece C++.

6.1 for:

El bucle **for** tiene 3 parámetros, los cuales son:

```
for(asignació; condición; iteración){
    acción se la condición es verdadera;
}
```

La tarea que pusimos al inicio la vamos a resolver, quedará de la siguiente manera:

```
#include <iostream>
using namespace std;
int main(){
  // La asignación si se puede hacer dentro del bucle
    for(int i=0; i<5; i++){
        cout<< "Hola mundo."<<endl;
    }
    return 0;
}
```

Lo que hace el código es recorrer desde 0 hasta 4. ¿Por qué desde cero? Porque asignamos el valor de 0 al inicio → **int i = 0;**
¿Por qué hasta 4? Porque una vez que **i** llegue a 5, la condición **i < 5** será falsa y saldrá del bucle.
¿Qué hace el **i++**? Lo que hace es incrementar **i** en 1 en cada iteración, es decir, **i** irá sumando **+1** hasta que la condición sea falsa.
El código tendrá la siguiente salida:

```
Hola mundo.
Hola mundo.
Hola mundo.
Hola mundo.
Hola mundo.
```

6.2 while:

Este bucle, al igual que el **for**, primero compara su condición; si es verdadera, se ejecuta el código, y si no, se salta. Sus partes son:

```
While(condición){
    acción a realizar si la condición es verdadera;
}
```

En esta parte tienes que tener cuidado porque si tienes un mínimo margen de error, el bucle se ejecutará infinitamente. Vamos a realizar el ejercicio anterior de igual manera que la anterior:

```
#include <iostream>
using namespace std;
int main(){
    int i=0;
    while(i<5){
        cout << "Hola mundo."<<endl;
        i++; // esta parte es crucial ya que sin ella se crearía un bucle infinito
    }
    return 0;
}
```

¿Qué es lo que está pasando? Primeramente declaramos una variable de tipo entero llamada **i** que tiene como valor **0**. La condición que se toma en cuenta es esta -> **(i < 5)**, ya sabemos que es verdadera y pasará a ejecutarse el código. Esta parte del código es fundamental -> **i++**. ¿Por qué? Porque sin ella, el valor de **i** seguirá siendo **0** y así hasta el infinito y más allá.

```
Hola mundo.
Hola mundo.
Hola mundo.
Hola mundo.
Hola mundo.
```

6.3 do-while:

El bucle **do-while** lo que hace primeramente es ejecutar la acción y luego verificar la condición. Sus partes son las siguientes:

```
do{
    acción
}While(condición);
```

Como ves, primeramente ejecuta una acción y luego compara la condición. Bueno, vamos a realizar la tarea:

```
#include <iostream>
using namespace std;
int main(){
    int i = 0;
    do{
        cout << "Hola mundo."<<endl;
        i++; // sin esto el buclue se hace infinitamente
    }while(i<5);
    return 0;
}
```

Primeramente se ejecuta el código y recién se evalúa la condición. Esto es muy útil si quieres asegurarte al menos una ejecución en tu código. La salida es:

```
Hola mundo.
Hola mundo.
Hola mundo.
Hola mundo.
Hola mundo.
```

Como viste, en la mayoría de los casos los bucles funcionan con inicializaciones y condiciones que controlan su ejecución, pero no siempre es así. A veces, un bucle puede detenerse o saltar iteraciones según ciertas condiciones específicas, por ejemplo, basándose en un carácter o valor determinado.

Si te quedó la duda, te invito a ver estos videos que te ayudarán a entender mejor los tres tipos de bucles (`for`, `while`, `do-while`). En los videos también se explican dos palabras reservadas muy importantes que se usan dentro de los bucles: **break** y **continue**.

- **break**: Esta palabra reservada se utiliza para detener por completo la ejecución de un bucle, sin importar si la condición principal sigue siendo verdadera. Cuando el programa encuentra un `break` dentro de un bucle, sale inmediatamente del mismo y continúa con la siguiente instrucción después del bucle.

- **continue**: Esta palabra reservada se usa para saltar la iteración actual del bucle y continuar con la siguiente. Es decir, cuando se ejecuta un `continue`, el código que queda dentro del bucle para esa iteración se omite y se pasa a evaluar la condición para la siguiente iteración.

Ambas instrucciones son muy útiles para controlar el flujo dentro de los bucles de forma precisa, haciendo que el código sea más flexible y eficiente.

Te recomiendo prestar atención a cómo se usan en los ejemplos para que comprendas bien su funcionamiento.

7 Arreglos o Vectores

Un **arreglo** (también llamado **vector** en programación básica) es una **estructura de datos** que permite almacenar múltiples valores **del mismo tipo** en una sola variable.

Cada valor dentro del arreglo se guarda en una **posición o índice**, comenzando desde el **índice 0**.

Los arreglos son útiles cuando se necesita **manejar colecciones de datos**, como listas de números, nombres o resultados, sin tener que declarar muchas variables por separado.

7.1 Vectores:

Como se mencionó anteriormente, los vectores almacenan una cantidad de valores de un solo tipo. A continuación, se muestran sus partes y un par de ejemplos:

```
// se tiene el tipo de dato entero -> int arreglo[5] es necesario que
tenga los corchetes y la cantidad de valores que vamos a utilizar.

    int arreglo[5] = {1,2,3,4,5};

// en este ejemplo lo asignamos de manera directa en el código.
```

Para poder mostrar el contenido de un arreglo puedes hacerlo con diferentes tipos de bucles, pero generalmente se utiliza el bucle **for**. Vamos a mostrar el código completo para mostrar el contenido del arreglo:

```
#include <iostream>
using namespace std;
int main(){
    int arreglo[5] = {1,2,3,4,5};
    for(int i=0; i<5; i++){
        cout << arreglo[i]<<" ";
    }
    return 0;
}
```

Recordemos lo siguiente: los arreglos tienen índices que inician desde 0, que en este caso sería:

```
tipo nombre[cantidad]      valores
int     arreglo[5]     = {1, 2, 3, 4, 5};

Los índices son ->        0, 1, 2, 3, 4
```

Entonces, lo que recorre el bucle son los índices del vector, que en el ejemplo tiene el nombre **arreglo**.
Ahora mostraremos la salida estándar:

```
1 2 3 4 5
```

Hagamos un ejercicio de entrada y salida de arreglos de tipo entero (`int`):

```
#include <iostream>
using namespace std;
int main(){
    int arreglo[5];
    cout<<"Digite 5 numeros: ";
    for(int i=0; i<5; i++){
        cin>>arreglo[i];
    }
    cout<<"Mostrando el arreglo."<<endl;
    for(int i=0; i<5; i++){
        cout<<arreglo[i]<<" ";
    }
    return 0;
}
```

Lo que hicimos primeramente es asignar un arreglo, como en este caso -> **int arreglo[5]**, que almacenará 5 valores enteros de tipo `int`.
Esta será la compilación exacta del código:

```
Digite 5 numeros: 1 2 3 4 5
Mostrando el arreglo.
1 2 3 4 5
```

Como se puede observar, es una manera fácil de guardar datos y mostrarlos en un arreglo.
Te invito a que practiques y desarrolles tu lógica de programación de manera constante.

7.2 Matrices:

La matriz es una estructura de datos en un plano bidimensional que tiene filas y columnas.
Como en los arreglos, las matrices empiezan por los índices [0][0].

Para formar una matriz necesitamos bucles anidados. ¿Qué son? Son bucles dentro de otros, como en este ejemplo:

```
    for(int i=0; i<3; i++){
        for(int j=0; j<3; j++){
            cout<<[i][j]<< " ";
        }
        cout << endl;
    }
```

Esta es la manera en la que se imprime una matriz de tipo 3 por 3.
Como se puede observar, tenemos dos bucles **for**, uno dentro del otro.
Para guardar e imprimir los datos, se hace de esta manera:

```
#include <iostream>
using namespace std;
int main(){
    int matriz[3][3];
    cout << "Digite un valor para la pariz: "<<endl;
    for(int i=0; i<3; i++){
        for(int j=0; j<3; j++){
            cout << "["<<i<<"]["<<j<<"]: ";
            cin >> matriz[i][j];
        }
    }
    cout << "Mostrando la matriz."<<endl;
    for(int i=0; i<3; i++){
        for(int j=0; j<3; j++){
            cout << matriz[i][j] <<" ";
        }
        cout << endl;
    }
    return 0;
}
```

Bien, en este pequeño ejercicio declaramos una matriz de tipo 3 por 3, guardamos los datos y los mostramos en pantalla.
El ejemplo nos quedará de la siguiente manera:

```
Digite un valor para la pariz:
[0][0]: 1
[0][1]: 2
[0][2]: 3
[1][0]: 4
[1][1]: 5
[1][2]: 6
[2][0]: 7
[2][1]: 8
[2][2]: 9
Mostrando la matriz.
1 2 3
4 5 6
7 8 9
```

Esta es la manera en la que debes guardar los datos en la matriz y mostrarlos.

8 Cadenas de Texto

8.1 char[n]:

Una forma común de almacenar cadenas de texto en C++ es usando un arreglo (vector) de tipo `char`. Es un método bastante eficiente, aunque limitado, ideal para programas que no requieren manejar textos muy largos en las entradas. Se declara y usa de la siguiente manera:

```
#include <iostream>
using namespace std;
int main(){
    char nombre[100]; //cuardamos memoría para 100 carácteres
    cout << "Digite su nombre: ";
    cin.getline(nombre,100,'\n');
    cout << "Su nombre guardado es: "<<nombre<<endl;
    return 0;
}
```

En este caso, para guardar cadenas de texto utilizamos `cin.getline()`, que recibe tres parámetros:

- En el primero, indicamos **dónde queremos guardar** la cadena (en este ejemplo, en la variable nombre).
- En el segundo, especificamos **cuántos espacios reservamos** para almacenar la cadena (la cantidad que definimos al inicio).

- Y en el tercero, indicamos cuándo debe **terminar la lectura**, que normalmente es un salto de línea `` `\n` `` (es decir, cuando presionamos la tecla **Enter**, se guarda la cadena hasta ese momento).
  Ejemplo de uso y compilación:

```
Digite su nombre: Maria Gonzales Perez
Su nombre guardado es: Maria Gonzales Perez
```

Algunas carácteristicas especiales que puedes usar son las siguientes. Para que algunas de estas funciones correctamente es necesario incluir la librería **#include<cstdlib>** (donde `var` es una variables):

Funciones clásicas de manejo de `char[]` (cstring)

Estas funciones trabajan con arreglos de caracteres (`char[]`), no con objetos `string`. Se requiere incluir la cabecera `#include <cstring>` y en algunos casos `#include <cstdlib>`.

Método	¿Para qué sirve?	Ejemplo	Resultado
`strlen(var)`	Obtener la longitud de una cadena	`int num = strlen("alex");`	`num = 4`
`strcpy(dest, src)`	Copiar el contenido de una cadena en otra	`strcpy(var1, "Alex");`	`var1 = "Alex"`
`strcmp(a, b)`	Comparar dos cadenas	`strcmp("hola", "hola") == 0`	`true` si son iguales
`strcat(a, b)`	Concatenar dos cadenas	`strcat(var1, "Alex");`	Si `var1 = "Hola "`, entonces `"Hola Alex"`
`strrev(var)`	Invertir una cadena	`char r[10] = strrev(var1);`	Si `var1 = "dia"`, entonces `"aid"`
`strupr(var)`	Convertir a mayúsculas	`strupr(var1);`	Si `var1 = "paz"`, será `"PAZ"`
`strlwr(var)`	Convertir a minúsculas	`strlwr(var1);`	Si `var1 = "PAZ"`, será `"paz"`
`strncmp(a, b, n)`	Comparar los primeros `n` caracteres de dos cadenas	`strncmp("alex", "a", 1) == 0`	`true` porque `'a' == 'a'`
`atoi(var)`	Convertir texto a entero	`int num = atoi("2");`	`num = 2`
`atof(var)`	Convertir texto a número con decimales	`float num = atof("2.3");`	`num = 2.3`

Nota: `strrev`, `strupr` y `strlwr` no son funciones estándar en C++, pero funcionan en algunos compiladores como **Dev-C++**.

Hagamos un ejercico simple que será, verificar si una palabra tiene la `a` minuscula en su nombre:

```
#include <iostream>
#include <cstring> // lo utilizamos para usar strlen()
using namespace std;
int main(){
    char nombre[20];
    bool bandera = false;
    int cantidad = 0;
    cout << "Digite su nombre: ";
    cin.getline(nombre, 20, '\n');
    cantidad = strlen(nombre);
    for(int i=0; i<cantidad; i++){
```

```
        if(nombre[i] == 'a'){
            bandera = true;
            break;
        }
    }
    if(bandera){
        cout << "Su nombre contiene almenos una a."<<endl;
    }else{
        cout << "Su nombre no contiene la vocal a."<<endl;
    }
    return 0;
}
```

Este programa verifica si al menos tu nombre contiene una **a**. Usamos 3 variables de diferentes tipos que son: **char[20]** para almacenar un nombre de máximo 20 caracteres, **bool** para verificar si hay al menos una **a** en el nombre y un **int** para saber exactamente cuántos caracteres tiene el nombre introducido.

La compilación del programa será de esta manera:

```
Digite su nombre: alex
Su nombre contiene almenos una a.
```

8.2 string

Otra manera de guardar cadenas de texto es usar los **string**, que es una forma más fácil de almacenar palabras y se entiende mejor.

Aquí te muestro un ejemplo:

```
#include <iostream>
#include <string>  //Agregamos esta librería para poder usar las strings
using namespace std;
int main(){
    string nombre;
    cout << "Digite su nombre: ";
    getline(cin, nombre); // de esta manera es como se guarda los strings
    cout << "Su nombre guardado es: "<<nombre<<endl;
    return 0;
}
```

Para guardar los **string** se usa la función reservada **getline()**, que en este caso utiliza dos parámetros.

Para saber qué poner en el primer parámetro, nos preguntamos: ¿Qué queremos hacer?
En este caso, queremos **guardar** y por ello ponemos `cin`.

En el segundo parámetro, va: ¿Dónde lo quieres guardar?
Por esa razón, colocamos el nombre de la variable de tipo **string**.

Algunas características básicas de los **string** son estas:

Métodos comunes de `string` en C++

MÉTODO	¿Para qué sirve?	Ejemplo	Resultado
`.empty()`	Verifica si la cadena está vacía	`string s = ""; s.empty();`	`true`
`.length()`	Devuelve el número de caracteres de la cadena	`string s = "Hola"; s.length();`	`4`
`.size()`	Igual que `.length()`	`s.size();`	`4`
`.substr(pos, n)`	Devuelve una subcadena desde `pos` con `n` caracteres	`s.substr(1, 2);`	`"ol"`
`.find()`	Devuelve la posición de una subcadena o carácter	`s.find("la");`	`2`
`.append(str)`	Añade `str` al final de la cadena	`s.append(" mundo");`	`"Hola mundo"`
`.insert(pos, s)`	Inserta una subcadena `s` en la posición `pos`	`s.insert(2, "xx");`	`"Hoxxla"`
`.erase(pos, n)`	Elimina `n` caracteres desde la posición `pos`	`s.erase(1, 2);`	`"Ha"` (de "Hola")
`.replace(pos,n,s)`	Reemplaza `n` caracteres desde `pos` por `s`	`s.replace(0, 4, "Hi");`	`"Hi"`
`toupper(c)`	Convierte un carácter a mayúscula	`toupper('a');`	`'A'`
`tolower(c)`	Convierte un carácter a minúscula	`tolower('A');`	`'a'`
`isalnum(c)`	Verifica si un carácter es alfanumérico	`isalnum('3');`	`true`
`isalpha(c)`	Verifica si un carácter es una letra	`isalpha('A');`	`true`
`isdigit(c)`	Verifica si un carácter es un dígito	`isdigit('8');`	`true`

Nota: funciones como `toupper()`, `isdigit()`, etc., requieren incluir la librería `#include <cctype>`.

Como modo de práctica, hagamos un programa que verifique si la primera y última palabra inician con un mismo carácter.

Si es así, que imprima **"hola mundo"**, y si no, muestre **"buenas noches"**.

```
#include <iostream>
#include <cctype> // para usar isalpha()
#include <cstring>
using namespace std;
int main(){
    string palabra;
    int tam = 0;
    cout << "Digite una palabra: ";
    getline(cin, palabra);
    tam = palabra.length();
    if(isalpha(palabra[0]) && isalpha(palabra[tam-1])){
        cout << "hola mundo."<<endl;
    }else{
        cout << "buenas noches."<<endl;
    }
    return 0;
}
```

En este programa lo que hicimos es verificar si una palabra empieza y termina con un mismo carácter.

Como en los vectores, las `string` tienen índices que empiezan en 0. Por esa razón, en la estructura de control **if** usamos `palabra[0]` y `palabra[tam-1]`.

Si la palabra es **casa**, lo que la condicional verificaría sería: `c` y `a`.

¿Son iguales? No, por eso el resultado sería **false** y el programa imprimiría "buenas noches".

En cambio, si la palabra fuera, por ejemplo, **alba**, la comparación sería entre `a` y `a`, dando **true** y el programa imprimiría "hola mundo".

Aquí te muestro un ejemplo de compilación:

```
Digite una palabra: casa
hola mundo.
---> caso contrario <---
Digite una palabra: casa7
buenas noches.
```

9 Estructuras

Las **estructuras** o `struct` en C++ son una forma de definir un **tipo de dato personalizado** que puede contener varios elementos de diferentes tipos. Una estructura es similar a una clase, pero se utiliza principalmente para **almacenar datos** y no para definir comportamientos.

9.1 Definición de estructura

Para definir una estructura en C++, se utiliza la palabra reservada struct, seguida del **nombre de la estructura** y los **elementos que la componen**.

```
struct Persona{
    string nombre;
    int edad;
    string direccion;
};
```

Uso de una estructura Una vez definida la estructura, puedes crear variables de ese tipo y acceder a sus miembros utilizando el operador punto (`.`).

Esto te permite agrupar datos relacionados bajo una sola variable, lo cual es muy útil para organizar la información de forma clara y ordenada.

```
  Persona persona; //creamos una variables de la estructura "Persona"
  persona.nombre = "juan";
  persona.edad = 15;
  persona.direccion = "calle 13";
```

Para acceder a las variables de la estructura, primeramente se coloca la variable que definimos anteriormente, seguida de un punto (`.`) y el nombre del campo de la estructura al que deseamos acceder, tal y como se muestra en el ejemplo anterior.

9.2 Ventajas de las estructuras

Las estructuras son muy útiles cuando necesitas almacenar varios datos relacionados en una sola unidad. Algunas ventajas importantes de utilizarlas son:

- Permiten organizar y estructurar los datos de manera lógica.
- Facilitan el acceso y la manipulación de los datos.
- Pueden ser utilizadas como parámetros de funciones o como valores de retorno.

```
#include <iostream>
#include <cstring>
using namespace std;
struct Persona{
    string nombre;
    int edad;
    string direccion;
}persona; // la variable lo podemos colocar aqui.
int main(){
    cout << "Digite su nombre: ";
    getline(cin, persona.nombre);
    cout << "Digite su edad: ";
    cin >> persona.edad;
    fflush(stdin); // lo usamos para vaciar el buffer.
    cout << "Digite su direccion: ";
    getline(cin, persona.direccion);
    cout<<"\nMostrando los datos de la persona."<<endl;
    cout << "El nombre es "<<persona.nombre<<endl;
    cout << "La edad es "<<persona.edad<<endl;
    cout << "La direccion es "<<persona.direccion<<endl;
    return 0;
}
```

En este caso, guardamos y mostramos los datos que el usuario nos proporciona. El resultado final del programa nos quedaría de la siguiente manera:

```
Digite su nombre: alex
Digite su edad: 15
Digite su direccion: calle 13

Mostrando los datos de la persona.
El nombre es alex
La edad es 15
La direccion es calle 13
```

9.3 Estructuras anidadas

Las estructuras anidadas en C++ se refieren a la capacidad de definir una estructura dentro de otra. Esto permite crear estructuras más complejas y jerárquicas, donde una estructura puede contener otra como uno de sus miembros.

Ejemplo de estructura anidada:

```
struct Direccion{
    string calle;
    string ciudad;
    string pais;
};
struct Persona{
    string nombre;
    int edad;
    Direccion direccion;
}persona; //declaramos la variable.
```

En este ejemplo, la estructura `Persona` contiene otra estructura llamada `Direccion` como uno de sus miembros. Puedes acceder a los campos de la estructura anidada usando el operador punto `(.)`.

Ventajas de las estructuras anidadas

Las estructuras anidadas son muy útiles cuando necesitas representar datos complejos y jerárquicos. Algunas ventajas importantes son:

- Permiten crear estructuras de datos más complejas y realistas.
- Facilitan la organización y el acceso a la información.
- Mejoran la legibilidad y mantenibilidad del código.

Aplicaciones de las estructuras anidadas

Las estructuras anidadas se utilizan en múltiples escenarios, tales como:

- Representar datos personales, como direcciones y contactos.
- Modelar estructuras complejas como árboles o grafos.
- Crear estructuras para juegos o simulaciones.

Ahora, vamos a completar el código y pediremos al usuario que llene los datos correspondientes:

```
#include <iostream>
#include <cstring>
using namespace std;
struct Direccion{
    string calle;
```

```
    string ciudad;
    string pais;
};
struct Persona{
    string nombre;
    Direccion direccion; //con estos parámetros designamos a la
                         //estructura anidada
    int edad;
}persona;
int main(){
    cout << "Digite su nombre: ";
    getline(cin, persona.nombre);
    cout << "Digite su calle: ";
    getline(cin, persona.direccion.calle);
    cout << "Digite su ciudad: ";
    getline(cin, persona.direccion.ciudad);
    cout << "Digite su pais: ";
    getline(cin, persona.direccion.pais);
    cout << "Digite su edad: ";
    cin >> persona.edad;
    cout<<"\nMostrando los datos de la persona\n";
    cout << "Nombre "<<persona.nombre<<endl;
    cout << "Calle " <<persona.direccion.calle<<endl;
    cout << "Ciudad "<<persona.direccion.ciudad<<endl;
    cout << "Pais " <<persona.direccion.pais<<endl;
    cout << "Edad " <<persona.edad<<endl;
}
```

Lo que hicimos es completar el ejemplo que mostramos al inicio. La salida estándar del programa quedaría de la siguiente manera:

```
Digite su nombre: alex
Digite su calle: calle 13
Digite su ciudad: puno
Digite su pais: peru
Digite su edad: 15

Mostrando los datos de la persona
Nombre alex
Calle calle 13
Ciudad puno
Pais peru
Edad 15
```

10 Funciones

En pocas palabras, las funciones consisten en dividir tareas para que el programa sea más legible. Como buena práctica de programación, es fundamental aprenderlas. Para ello, veremos una estructura básica:

```
En el tipo de la función, si se requiere retornar un tipo de valor en espesífico
puedes
usar (int, float, bool) que son las más comunes.

Existen funciones que no retornan nada y el más comun es el "void".

La manera en la que creamos la funcion puede y no puede tener valores es decir:
int sumar(int n, int m) //En este caso la función recebe dos parámetros de tipo
                        //entero.
void pedir() // Y en esta otra la función no recibe parámetros.
```

Si te das cuenta, en este libro verás que mayormente utilizamos funciones de tipo `void` para pedir datos en nuestros programas, mientras que usaremos otros tipos de funciones cuando necesitemos realizar una tarea específica que devuelva un valor.

Recuerda que en los parámetros puedes usar el nombre que tú quieras, pero debes tener en cuenta que es muy importante especificar correctamente el tipo de valor que tomará.

Una buena práctica de programación es prototipar las funciones. Esto significa declarar cómo serán antes del `main`, y luego, después de la función principal, escribimos el código que realiza esa tarea específica.

Ahora, vamos a crear un programa sencillo que retorne la suma de dos números enteros.

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

int suma(int, int); // este será nuestro prototipo de nueestra función

int main(){
    int num1, num2;
    int resultado; // como la funcion retorna un valor, ese valor lo guardamos en
```

```
                    // esta variables.
    cout << "Digite el primer numero: ";
    cin >> num1;
    cout << "Digite el segundo numero: ";
    cin >> num2;

    resultado = suma(num1, num2); // llamamos a la funcion poniendo el nombre de la
                                  // funcion seguida de los dos parámetros.
    cout << "El resultado es: "<<resultado<<endl; // finalmente lo imrpimimos
    return 0;
}
//realizamos el cuerpo de la funcion.
int suma(int a, int b){ // a y b son variables.
    int resultado;
    resultado = a + b;
    return resultado;   // la palabra "return" es como decir vota el resultado.
}
```

El programa es básico, ya que solo realiza una suma simple, pero para programas más grandes es **fundamental hacerlo con funciones**, y en todo caso, es necesario que lo aprendas como una buena práctica.

10.1 Paso de Parámetros

Para pasar arreglos, matrices o cadenas de texto, se requiere de otro tipo de parámetros que son:

```
int arreglo(int a[], int tamanio);
//El a[] es para pasar el arreglo y para saber la cantidad ponemos el "tamanio".

int matriz(int a[][100], int filas, int columnas);
//En este caso en necesario poner el tamaño máximo de columnas, seguida del total
// de filas y columnas

int cadenas(string nombre);
// para mandar de parámetro cadenas de texto.

int caracteres(char n);
//para poder mandar vocales y carácteres únicos.
```

Como ves, estos son los parámetros que debes aprender. Pero para hacerlo más interesante, vamos a crear un programa que realice una serie de opciones. Lo haremos `solo con funciones` para que así

puedas `reforzar tu conocimiento` y familiarizarte más con su uso.

Las opciones que habrá en el `MENU` serán las siguientes:

```
-------------->MENU<-------------
1. OPERACIONES ARITMÉTICAS.
2. OPERACIONES CON ARREGLOS.
3. OPERACIONES CON MATRICES.
4. JUEGO DE "ADIVINA EL NUMERO".
5. SALIR.
```

Cabe recalcar que cada tema será explicado de una manera casi profesional, porque yo también tengo mucho que aprender. Sin embargo, el motivo por el cual hago esto es para que no tengas complicaciones futuras y logres entender todo de la mejor manera posible. Haciendo esto, aprenderás conceptos fundamentales que te servirán mucho más adelante.

10.2 Programa N.º 1

Para poder realizar este programa, es fundamental retroceder y repasar los temas anteriores con atención. Esto te ayudará a que luego te sea mucho más fácil entender este pequeño programa.

11 Punteros

11.1 ¿Qué es un puntero?

Los punteros son una de esas cosas en C++ que al principio suenan difíciles… pero cuando los entiendes bien, ¡te das cuenta de que son súper poderosos!

Un puntero es una variable especial que guarda la dirección de memoria de otra variable. O sea, en lugar de guardar directamente un número o una letra, guarda el lugar donde esa información está almacenada en la memoria de tu computadora.

Piensa que tienes una casa (una variable normal) y un GPS (el puntero) que te dice exactamente en qué calle y número está esa casa. El puntero no es la casa, pero sabe cómo llegar a ella.

11.2 ¿Cómo se declara un puntero?

Para declarar un puntero, se usa el símbolo *:

```
tipo *nombre_puntero;
```

Por ejemplo, para un puntero a un entero:

```
int *puntero;
```

Eso significa: "puntero es una variable que guarda la dirección de un **int**".

11.3 Ejemplo básico de punteros

Veamos un ejemplo sencillo para que lo veas en acción:

```
#include <iostream>
using namespace std;
int main(){
    int numero = 10;
    int *puntero = &numero; // & = operador de dirección
    cout << "Valor de la variable: " << numero << endl;
```

```
    cout << "Dirección de memoria de la variable: " << &numero << endl;
    cout << "Valor guardado en el puntero: " << puntero << endl;
    cout << "Valor al que apunta el puntero: " << *puntero << endl;
    return 0;
}
```

¿Qué está pasando aquí?

`int numero = 10;` Creamos una variable normal. `int *puntero = &numero;` Le decimos al puntero que guarde la dirección de `numero`. `*puntero` Esto significa: "muéstrame el valor que está en esa dirección".

Resultado esperado

```
Valor de la variable: 10
Dirección de memoria de la variable: 0x61ff08
Valor guardado en el puntero: 0x61ff08
Valor al que apunta el puntero: 10
```

‘

Nota: las direcciones como `0x61ff08` van a cambiar cada vez que ejecutes el programa. Eso es normal.

Cambiar el valor desde el puntero

Una de las cosas más geniales de los punteros es que puedes **modificar el valor original** de una variable usando el puntero. Mira:

```
#include <iostream>
using namespace std;
int main(){
    int numero = 5;
    int *ptr = &numero;
    *ptr = 99; // cambia el valor original de numero
    cout << "Nuevo valor de numero: " << numero << endl;
    return 0;
}
```

Esto imprime:

```
Nuevo valor de numero: 99
```

¡Sí! Aunque nunca escribimos `numero = 99;`, el valor cambió.
¿Por qué? Porque **`*ptr` apunta directamente a `numero`**, así que cualquier cambio afecta a la variable original.

11.4 Aplicación:

Intercambiar valores con punteros Una de las aplicaciones clásicas de punteros es intercambiar valores entre dos variables usando funciones.

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

void intercambiar(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(){
    int x = 5, y = 8;
    intercambiar(&x, &y);
    cout << "x = " << x << ", y = " << y << endl;
    return 0;
}
```

Salida:

```
x = 8, y = 5
```

¿Por qué funciona?

Porque estamos pasando la dirección de memoria de `x` y `y` a la función. Entonces cualquier cambio se hace directamente sobre los valores originales.

Consejos para trabajar con punteros:

- Siempre inicializa tus punteros (aunque sea con nullptr).
- Ten cuidado al usar * (puede causar errores si no sabes qué estás apuntando).
- Puedes usar punteros para crear arreglos dinámicos, estructuras, funciones recursivas y más.
- Aprender punteros es la puerta a temas más avanzados como memoria dinámica y estructuras enlazadas.

12 Pilas

12.1 Introducción

Imagina que estás apilando platos en tu cocina. Pones uno sobre otro y, cuando necesitas uno, tomas el que está arriba. Así funcionan las pilas en programación: el último elemento en entrar es el primero en salir. A esto se le llama estructura LIFO (Last In, First Out). En este capítulo aprenderás a trabajar con pilas utilizando listas enlazadas en C++. Verás cómo agregar y quitar elementos, además de comprender cómo fluye la lógica internamente.

12.2 Teoría

Una pila es una estructura de datos que permite insertar y eliminar elementos sólo desde un extremo, llamado "tope". Las operaciones principales son:

`push:` insertar un nuevo elemento (en nuestro código se llama agregarPila).

`pop:` eliminar el elemento del tope (en nuestro código se llama quitarPila).

En este ejemplo, usamos una lista enlazada simple, donde cada nodo contiene un carácter y un puntero al siguiente nodo.

12.3 Código y explicación

Ten en cuenta que el código estará por partes y tu reto será juntarlos y hacer que compile:

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

struct Nodo{
    char dato;
    Nodo *siguiente;
};
```

- Creamos una estructura `Nodo`, que guarda un dato de tipo `char` y un puntero al siguiente nodo.

- Así es como simulamos una pila dinámica.

```
void menu();
void agregarPila(Nodo *&, char);
void quitarPila(Nodo *&, char &);
```

- Declaramos funciones auxiliares para mostrar el menú, agregar elementos a la pila y retirarlos.

```
int main(){
    menu();
    system("pause");
    return 0;
}
```

La función `main` simplemente llama al **menú** y espera una tecla antes de terminar. **Menú principal**

```
void menu(){
    Nodo *pila = NULL;
    int opcion;
    char dato;
    do{
        cout << "\n\t\tMENU DE PILAS\n";
        cout << "1. Agregar elemento a la pila\n";
        cout << "2. Mostrar todos los elementos de la pila\n";
        cout << "3. salir"<<endl;
        cout << "Ingrese una opcion: ";
        cin >> opcion;
```

- Creamos un puntero llamado `pila`, inicialmente vacío (`NULL`).
- Mostramos un menú con tres opciones: agregar, mostrar y salir.

```
        switch(opcion){
            case 1: cout << "Digite un caracter: ";
                    cin >> dato;
                    agregarPila(pila, dato);
                    break;
```

- Si elige la opción 1, el usuario digita un carácter y se agrega a la pila.

```
            case 2: cout << "\nMostrando todos los elementos de la pila...";
                    while(pila != NULL){
                        quitarPila(pila, dato);
                        if(pila != NULL){
```

```
                cout << dato << " , ";
            }
            else{
                cout << dato << ".";
            }
        }
        cout << endl;
        system("pause");
        break;
```

- Si elige la opción 2, mostramos y eliminamos todos los elementos de la pila, uno por uno.
- Si aún hay elementos, se imprime con coma; el último termina con punto.

```
            case 3: break;
            default: cout << "Opcion no valida, intente de nuevo." << endl;
        }

        if(opcion != 3){
            system("cls"); // Limpiar la pantalla
        }
    }while(opcion != 3);
}
```

- Si elige salir, termina el ciclo. Si no, se limpia la pantalla y vuelve al menú. Agregar elemento (push)

```
void agregarPila(Nodo *&pila, char n){
    Nodo *nuevo_nodo = new Nodo();
    nuevo_nodo->dato = n;
    nuevo_nodo->siguiente = pila;
    pila = nuevo_nodo;
}
```

- Creamos un nuevo nodo.
- Lo enlazamos con el nodo anterior (el que era el tope).
- Luego, `pila` apunta al nuevo nodo. Ahora es el nuevo tope. Quitar elemento (pop)

```
void quitarPila(Nodo *&pila, char &n){
    Nodo *aux = pila;
    n = aux->dato;
    pila = aux->siguiente;
    delete aux;
```

```
}
```

- Guardamos el nodo actual en `aux`.
- Extraemos su valor a `n`.
- Movemos el tope (`pila`) al siguiente nodo.
- Liberamos la memoria. **Ejemplo en ejecución**

```
        MENU DE PILAS
1. Agregar elemento a la pila
2. Mostrar todos los elementos de la pila
3. salir
Ingrese una opcion: 1
Digite un caracter: A

Ingrese una opcion: 1
Digite un caracter: B

Ingrese una opcion: 2
Mostrando todos los elementos de la pila...
B , A.
```

- Como puedes ver, el último carácter en entrar fue el primero en salir. Justo como una pila de platos.

13 Colas

13.1 Introducción

Imagina que estás en la fila de una tienda para pagar. Llega una persona, se pone al final. El cajero atiende al primero en la fila. Esa es la lógica de una cola en programación: el primero en entrar es el primero en salir. A esto se le llama estructura FIFO (First In, First Out). En este capítulo, aprenderás a crear y manejar colas en C++ utilizando listas enlazadas.

13.2 Teoría

Una cola es una estructura lineal donde los datos se insertan por un extremo llamado final y se eliminan por otro llamado frente. Las operaciones más comunes son:

- `enqueue`: insertar un elemento al final (en nuestro código: `insertar`).
- `dequeue`: eliminar un elemento del frente (en nuestro código: `quitar`).

Usaremos listas enlazadas para construir la cola, sin usar bibliotecas externas ni STL.

13.3 Código y explicación

Ten en cuenta que el código estará por partes y tu reto será juntarlos y hacer que compile:

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

struct Nodo{
    char dato;
    Nodo *siguiente;
};
```

- Definimos la estructura básica de la cola: cada `Nodo` tiene un carácter (`dato`) y un puntero al siguiente nodo.

```
void menu();
void insertar(Nodo *&, Nodo *&, char);
bool cola_vacia(Nodo *);
void quitar(Nodo *&, Nodo *&, char &);
```

- Declaramos las funciones para el menú, insertar, verificar si la cola está vacía y quitar elementos.

```
int main(){
    menu();
    system("pause");
    return 0;
}
```

La función **main** solo llama al menú y espera una tecla para terminar. Menú principal

```
void menu(){
    int opcion;
    char dato;
    Nodo *frente = NULL;
    Nodo *fin = NULL;
```

- **frente** es el primer elemento de la cola.
- **fin** apunta al último elemento.

```
    do{
        cout<<"\tMenu de opciones\n";
        cout<<"1. Insertar un caracter en la cola\n";
        cout<<"2. Mostrar los elementos de la cola\n";
        cout<<"3. Salir"<<endl;
        cout<<"Opcion: ";
        cin >> opcion;
```

- Mostramos el menú al usuario con tres opciones.

```
        switch(opcion){
            case 1: cout<<"\nDigite un caracter: ";
                    cin >> dato;
                    insertar(frente, fin, dato);
                    break;
```

- Si elige la opción 1, se inserta un nuevo carácter al final de la cola.

```
            case 2: cout<<"\nMostrando los elementos de la cola: ";
                    while(frente != NULL){
                        quitar(frente, fin, dato);
                        if(frente != NULL){
                            cout << dato<< " , ";
                        }
                        else{
                            cout << dato<<"."; // Último elemento
                        }
                    }
                    cout << endl;
                    system("pause");
                    break;
            case 3: break;
        }
        system("cls");
    }while(opcion != 3);
}
```

- Si elige la opción 2, se recorren y eliminan todos los elementos de la cola.
- Se imprime el dato y se libera la memoria. **Insertar elemento (`enqueue`)**

```
void insertar(Nodo *&frente, Nodo *&fin, char n){
    Nodo *nuevo_nodo = new Nodo();
    nuevo_nodo->dato = n;
    nuevo_nodo->siguiente = NULL;
```

- Creamos un nuevo nodo con el dato ingresado y apuntamos su `siguiente` a `NULL` (porque será el último).

```
    if(cola_vacia(frente)){
        frente = nuevo_nodo;
    }
    else{
        fin->siguiente = nuevo_nodo;
    }
    fin = nuevo_nodo;
}
```

- Si la cola está vacía, tanto `frente` como `fin` apuntan al nuevo nodo.
- Si ya hay nodos, lo conectamos al final y actualizamos el puntero `fin`. **Verificar si la cola está vacía**

```
bool cola_vacia(Nodo *frente){
    return (frente == NULL)? true : false;
}
```

- Una función sencilla que devuelve `true` si no hay elementos en la cola. **Quitar elemento (dequeue)**

```
void quitar(Nodo *&frente, Nodo *&fin, char &n){
    n = frente->dato;
    Nodo *aux = frente;

    if(frente == fin){
        frente = NULL;
        fin = NULL;
    }
    else{
        frente = frente->siguiente;
    }
    delete aux;
}
```

- Se guarda el dato del frente, se mueve el puntero `frente` al siguiente nodo y se libera el anterior.
- Si solo había un elemento, ambos punteros quedan en `NULL`. **Ejemplo en ejecución**

```
    Menu de opciones
1. Insertar un caracter en la cola
2. Mostrar los elementos de la cola
3. Salir
Opcion: 1
Digite un caracter: A

Opcion: 1
Digite un caracter: B

Opcion: 2
Mostrando los elementos de la cola: A , B.
```

Como puedes ver, el primer carácter en entrar fue el primero en salir. Exactamente como en una fila de personas.

14 Listas enlazadas

14.1 Introducción

Una lista enlazada es como una cadena de personas tomadas de la mano: cada persona conoce a la siguiente. En programación, una lista enlazada es una estructura de datos en la que cada elemento (llamado nodo) contiene un dato y un puntero al siguiente nodo. A diferencia de los arreglos, no necesitamos saber el tamaño de antemano, y podemos insertar o eliminar elementos sin mover todos los demás.

14.2 Teoría

¿Qué es una lista enlazada?

Es una colección dinámica de nodos donde cada nodo apunta al siguiente. Existen distintos tipos:

- Lista enlazada simple: cada nodo apunta al siguiente.
- Lista doblemente enlazada: cada nodo apunta al anterior y al siguiente.
- Lista circular: el último nodo apunta al primero.

Aquí trabajaremos con una `lista enlazada simple`.

Operaciones básicas:

- Insertar elementos al inicio o al final.
- Mostrar todos los elementos.
- Eliminar elementos.

14.3 Código y explicación

A continuación, te presento un código básico de lista enlazada con inserción al inicio, recorrido y eliminación de nodos.

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

```
struct Nodo {
    int dato;
    Nodo* siguiente;
};
```

- Cada `Nodo` contiene un dato (`int`) y un puntero al siguiente nodo de la lista.

```
void insertarInicio(Nodo*&, int);
void mostrarLista(Nodo*);
void eliminarLista(Nodo*&);
```

- Declaramos las funciones que nos permiten manipular la lista.

```
int main() {
    Nodo* lista = NULL;
    int opcion, valor;

    do {
        cout << "\n--- MENÚ DE LISTA ENLAZADA ---\n";
        cout << "1. Insertar elemento al inicio\n";
        cout << "2. Mostrar lista\n";
        cout << "3. Eliminar lista\n";
        cout << "4. Salir\n";
        cout << "Opción: ";
        cin >> opcion;

        switch(opcion) {
            case 1:
                cout << "Ingrese un número: ";
                cin >> valor;
                insertarInicio(lista, valor);
                break;
            case 2:
                cout << "Contenido de la lista: ";
                mostrarLista(lista);
                break;
            case 3:
                eliminarLista(lista);
                cout << "Lista eliminada.\n";
                break;
            case 4:
```

```
                cout << "Saliendo...\n";
                break;
            default:
                cout << "Opción inválida.\n";
        }
        system("pause");
        system("cls");
    } while(opcion != 4);

    return 0;
}
```

- Menú clásico que permite insertar, mostrar y eliminar la lista enlazada. **Función: Insertar al inicio**

```
void insertarInicio(Nodo*& lista, int valor) {
    Nodo* nuevo_nodo = new Nodo();
    nuevo_nodo->dato = valor;
    nuevo_nodo->siguiente = lista;
    lista = nuevo_nodo;
}
```

- Se crea un nuevo nodo, se enlaza al inicio y el puntero de la lista se actualiza.
- Esta operación es rápida, siempre toma el mismo tiempo sin importar cuántos nodos haya. **Función: Mostrar la lista**

```
void mostrarLista(Nodo* lista) {
    while(lista != NULL) {
        cout << lista->dato;
        if (lista->siguiente != NULL)
            cout << " -> ";
        lista = lista->siguiente;
    }
    cout << "." << endl;
}
```

- Recorre todos los nodos desde el primero hasta el último.
- Imprime cada dato seguido por una flecha, excepto el último que termina en punto. **Función: Eliminar todos los nodos**

```
void eliminarLista(Nodo*& lista) {
    Nodo* aux;
```

```
    while(lista != NULL) {
        aux = lista;
        lista = lista->siguiente;
        delete aux;
    }
}
```

- Libera la memoria nodo por nodo, evitando pérdidas de memoria (muy importante en programas grandes).

Ejemplo en ejecución

```
--- MENÚ DE LISTA ENLAZADA ---
1. Insertar elemento al inicio
2. Mostrar lista
3. Eliminar lista
4. Salir
Opción: 1
Ingrese un número: 5

Opción: 1
Ingrese un número: 8

Opción: 2
Contenido de la lista: 8 -> 5.
```

- Como estamos insertando al inicio, el último elemento agregado aparece primero.

15 Listas Doblemente Enlazadas

15.1 Teoría

Las listas doblemente enlazadas son una estructura de datos dinámica en la que cada nodo almacena tres cosas:

- Un dato (el valor que queremos guardar).
- Un puntero al siguiente nodo.
- Un puntero al nodo anterior.

A diferencia de las listas enlazadas simples, donde solo se puede avanzar hacia adelante, con una lista doblemente enlazada podemos movernos en ambos sentidos: hacia adelante (next) y hacia atrás (prev). Esto las hace especialmente útiles para operaciones como insertar o eliminar nodos en cualquier parte de la lista de forma eficiente.

Imagina una fila de personas tomadas de las manos con los de adelante y atrás: así es una lista doblemente enlazada.

15.2 Código explicado

Vamos a implementar una lista doblemente enlazada que permite:

1. Insertar elementos al final.
2. Mostrar los elementos de izquierda a derecha.
3. Mostrar los elementos de derecha a izquierda.
4. Eliminar elementos por valor.

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

// Estructura del nodo
struct Nodo {
    int dato;
    Nodo* siguiente;
    Nodo* anterior;
```

```
};

// Prototipos
void insertarFinal(Nodo*&, Nodo*&, int);
void mostrarInicioAFin(Nodo*);
void mostrarFinAInicio(Nodo*);
void eliminarElemento(Nodo*&, Nodo*&, int);
void menu();

int main() {
    menu();
    return 0;
}
```

Menú de opciones

```
void menu() {
    Nodo* cabeza = NULL;
    Nodo* cola = NULL;
    int opcion, valor;

    do {
        cout << "\n\t\tMENU - LISTA DOBLEMENTE ENLAZADA\n";
        cout << "1. Insertar elemento al final\n";
        cout << "2. Mostrar de inicio a fin\n";
        cout << "3. Mostrar de fin a inicio\n";
        cout << "4. Eliminar un elemento por valor\n";
        cout << "5. Salir\n";
        cout << "Seleccione una opcion: ";
        cin >> opcion;

        switch(opcion) {
            case 1:
                cout << "Ingrese un numero: ";
                cin >> valor;
                insertarFinal(cabeza, cola, valor);
                break;
            case 2:
                mostrarInicioAFin(cabeza);
                break;
            case 3:
```

```
                mostrarFinAInicio(cola);
                break;
            case 4:
                cout << "Ingrese el valor a eliminar: ";
                cin >> valor;
                eliminarElemento(cabeza, cola, valor);
                break;
            case 5:
                cout << "Saliendo del programa...\n";
                break;
            default:
                cout << "Opcion no valida.\n";
        }
        system("pause");
        system("cls");
    } while(opcion != 5);
}
```

Insertar al final

```
void insertarFinal(Nodo*& cabeza, Nodo*& cola, int valor) {
    Nodo* nuevo = new Nodo();
    nuevo->dato = valor;
    nuevo->siguiente = NULL;
    nuevo->anterior = cola;

    if (cabeza == NULL) {
        cabeza = nuevo;
    } else {
        cola->siguiente = nuevo;
    }
    cola = nuevo;
}
```

Mostrar de inicio a fin

```
void mostrarInicioAFin(Nodo* cabeza) {
    cout << "Lista de inicio a fin: ";
    while (cabeza != NULL) {
        cout << cabeza->dato << " ";
        cabeza = cabeza->siguiente;
    }
```

```
    cout << endl;
}
```

Mostrar de fin a inicio

```
void mostrarFinAInicio(Nodo* cola) {
    cout << "Lista de fin a inicio: ";
    while (cola != NULL) {
        cout << cola->dato << " ";
        cola = cola->anterior;
    }
    cout << endl;
}
```

Eliminar un nodo por valor

```
void eliminarElemento(Nodo*& cabeza, Nodo*& cola, int valor) {
    Nodo* actual = cabeza;

    while (actual != NULL && actual->dato != valor) {
        actual = actual->siguiente;
    }

    if (actual == NULL) {
        cout << "Elemento no encontrado.\n";
        return;
    }

    if (actual == cabeza) {
        cabeza = actual->siguiente;
        if (cabeza != NULL) cabeza->anterior = NULL;
    } else {
        actual->anterior->siguiente = actual->siguiente;
    }

    if (actual == cola) {
        cola = actual->anterior;
        if (cola != NULL) cola->siguiente = NULL;
    } else {
        if (actual->siguiente != NULL)
            actual->siguiente->anterior = actual->anterior;
    }
```

```
    delete actual;
    cout << "Elemento eliminado con éxito.\n";
}
```

Ejemplo de uso

```
> Insertar: 10
> Insertar: 20
> Insertar: 30
> Mostrar de inicio a fin: 10 20 30
> Mostrar de fin a inicio: 30 20 10
> Eliminar: 20
> Mostrar de inicio a fin: 10 30
```

Como puedes ver, puedes navegar fácilmente hacia adelante y hacia atrás, e incluso eliminar elementos en cualquier posición.

15.3 Conclusión

Las listas doblemente enlazadas son herramientas muy poderosas cuando necesitas recorrer la lista en ambos sentidos, insertar o eliminar nodos en posiciones arbitrarias. Aunque ocupan un poco más de memoria (por el puntero adicional), su flexibilidad las hace muy útiles en muchos algoritmos y aplicaciones reales como editores de texto, sistemas de navegación, e incluso juegos.

16 Recursión

16.1 Teoría

La recursión es una técnica de programación en la que una función se llama a sí misma para resolver un problema. Esta técnica se basa en dividir un problema grande en problemas más pequeños, hasta llegar a un caso base que puede resolverse directamente.

- Una función recursiva debe tener:
- Un caso base (condición de parada): evita que la función se llame infinitamente.
- Una llamada recursiva que reduce el problema en cada paso.

Imagina unas muñecas rusas (matrioskas), cada una contiene una más pequeña dentro. Recursión funciona así: vas abriendo una a una hasta llegar a la más pequeña (caso base), y luego las vuelves a cerrar en orden inverso.

16.2 Código explicado

Vamos a resolver dos problemas clásicos con recursión:

1. Calcular el factorial de un número.
2. Calcular el n-ésimo número de la secuencia Fibonacci.

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

// Prototipos
int factorial(int);
int fibonacci(int);
void menu();

int main() {
    menu();
    return 0;
}
```

Menú de opciones

```
void menu() {
    int opcion, numero;

    do {
        cout << "\n\t\tMENU - RECURSION\n";
        cout << "1. Calcular factorial\n";
        cout << "2. Calcular n-ésimo Fibonacci\n";
        cout << "3. Salir\n";
        cout << "Seleccione una opcion: ";
        cin >> opcion;

        switch (opcion) {
            case 1:
                cout << "Ingrese un número: ";
                cin >> numero;
                cout << "Factorial de " << numero << " es: " << factorial(numero) << endl;
                break;
            case 2:
                cout << "Ingrese la posición n: ";
                cin >> numero;
                cout << "Fibonacci(" << numero << ") = " << fibonacci(numero) << endl;
                break;
            case 3:
                cout << "Saliendo del programa...\n";
                break;
            default:
                cout << "Opcion no valida.\n";
        }
        system("pause");
        system("cls");
    } while(opcion != 3);
}
```

Factorial de forma recursiva

```
int factorial(int n) {
    if (n == 0 || n == 1) // Caso base
        return 1;
    else
        return n * factorial(n - 1); // Llamada recursiva
```

```
}
```

Fibonacci de forma recursiva.

```
int fibonacci(int n) {
    if (n == 0) return 0; // Caso base 1
    if (n == 1) return 1; // Caso base 2
    return fibonacci(n - 1) + fibonacci(n - 2); // Llamadas recursivas
}
```

Ejemplo de uso.

```
> Ingrese 5 para factorial → Resultado: 120
> Ingrese 6 para fibonacci → Resultado: 8
```

16.3 Conclusión

La recursión es una forma elegante y poderosa de resolver problemas, especialmente cuando la solución se puede expresar en términos del mismo problema. Aunque no siempre es la más eficiente (como en Fibonacci sin memoización), es clave para entender algoritmos más avanzados como recorridos en árboles o resolución de laberintos.

17 Árboles Binarios

17.1 Teoría

Un árbol binario es una estructura de datos jerárquica en la que cada nodo tiene a lo sumo dos hijos: uno izquierdo y uno derecho. Es muy útil para representar relaciones jerárquicas, búsquedas rápidas y estructuras como árboles de expresión, árboles de decisión, etc.

Cada nodo contiene:

- Un dato.
- Un puntero al hijo izquierdo.
- Un puntero al hijo derecho.

Imagina un árbol genealógico: cada persona puede tener dos hijos, y así se van ramificando generaciones. ¡Así son los árboles binarios!

17.2 Código explicado

Vamos a crear un árbol binario que permita:

1. Insertar elementos.
2. Mostrar elementos en orden (inorden).
3. Mostrar en preorden y postorden.

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

// Estructura del nodo
struct Nodo {
    int dato;
    Nodo* izq;
    Nodo* der;
};
```

```
// Prototipos
Nodo* insertar(Nodo*, int);
void inorden(Nodo*);
void preorden(Nodo*);
void postorden(Nodo*);
void menu();

int main() {
    menu();
    return 0;
}
```

Menú de opciones

```
void menu() {
    Nodo* raiz = NULL;
    int opcion, valor;

    do {
        cout << "\n\t\tMENU - ARBOL BINARIO\n";
        cout << "1. Insertar elemento\n";
        cout << "2. Mostrar Inorden\n";
        cout << "3. Mostrar Preorden\n";
        cout << "4. Mostrar Postorden\n";
        cout << "5. Salir\n";
        cout << "Seleccione una opcion: ";
        cin >> opcion;

        switch (opcion) {
            case 1:
                cout << "Ingrese un número: ";
                cin >> valor;
                raiz = insertar(raiz, valor);
                break;
            case 2:
                cout << "Recorrido inorden: ";
                inorden(raiz);
                cout << endl;
                break;
            case 3:
                cout << "Recorrido preorden: ";
```

```
                preorden(raiz);
                cout << endl;
                break;
            case 4:
                cout << "Recorrido postorden: ";
                postorden(raiz);
                cout << endl;
                break;
            case 5:
                cout << "Saliendo...\n";
                break;
            default:
                cout << "Opcion no valida.\n";
        }
        system("pause");
        system("cls");
    } while(opcion != 5);
}
```

Insertar en el árbol

```
Nodo* insertar(Nodo* nodo, int valor) {
    if (nodo == NULL) {
        Nodo* nuevo = new Nodo();
        nuevo->dato = valor;
        nuevo->izq = nuevo->der = NULL;
        return nuevo;
    } else if (valor < nodo->dato) {
        nodo->izq = insertar(nodo->izq, valor);
    } else {
        nodo->der = insertar(nodo->der, valor);
    }
    return nodo;
}
```

Recorridos

```
void inorden(Nodo* nodo) {
    if (nodo != NULL) {
        inorden(nodo->izq);
        cout << nodo->dato << " ";
```

```
        inorden(nodo->der);
    }
}

void preorden(Nodo* nodo) {
    if (nodo != NULL) {
        cout << nodo->dato << " ";
        preorden(nodo->izq);
        preorden(nodo->der);
    }
}

void postorden(Nodo* nodo) {
    if (nodo != NULL) {
        postorden(nodo->izq);
        postorden(nodo->der);
        cout << nodo->dato << " ";
    }
}
```

Ejemplo de uso

```
> Insertar: 8, 3, 10, 1, 6
> Inorden: 1 3 6 8 10
> Preorden: 8 3 1 6 10
> Postorden: 1 6 3 10 8
```

17.3 Conclusión

Los árboles binarios son una base fundamental para muchas estructuras más avanzadas como árboles AVL, B, B+, heaps y más. Entender cómo insertar y recorrer árboles es esencial para resolver problemas que implican decisiones jerárquicas, búsquedas rápidas y almacenamiento eficiente.