

# APUNTES DEL CURSO

## “PROGRAMACIÓN BÁSICA”



Autor: Garcia Villegas, Christian

## Contenido

PROGRAMA .....	13
Tipados & no tipados.....	14
Compilado, interpretado y intermedio .....	14
Paradigmas de programación .....	16
Editores y entornos de desarrollo .....	17
Tipos de errores .....	18
HERRAMIENTAS EN LÍNEA .....	19
PARA COMPARTIR CÓDIGO .....	19
Herramienta colaborativa.....	20
MI PRIMER PROGRAMA .....	20
OPERADORES EN C++ .....	21
TIPOS DE DATOS .....	25
LECTURA Y ESCRITURA .....	31
Entrada y salida estándar .....	31
scanf - printf .....	32
gets - puts.....	33
VARIABLES Y TIPOS DE DATOS .....	35
ESTRUCTURAS .....	51
Union.....	51
Struct.....	51
ARREGLO CON VECTORES.....	52
ORDENAMIENTO DE VECTORES.....	53
Iterativos .....	53
Recursivo .....	53
Constantes en C++.....	54
FUNCIONES CON CADENA DE CARACTERES .....	56
CONVERSIONES .....	57
CADENAS.....	58
FUNCIONES RECURSIVAS .....	59
PROGRAMACIÓN ORIENTADA OBJETOS .....	61
Abstracción.....	62
Encapsulamiento .....	63
Herencia .....	66
Polimorfismo .....	67

Funciones Virtuales .....	69
STACK(PILA) .....	70
CONSTRUCTORES & DESTRUCTORES .....	72
SISTEMAS DE PROTECCIÓN.....	74
FUNCIONES Y CLASES AMIGAS .....	76
Funciones Amigas.....	76
Funciones Externas Amigas .....	79
Funciones amigas en otras clases. ....	80
Clases amigas .....	81
MODIFICADORES PARA MIEMBROS .....	83
Funciones inline .....	83
Niveles De Acceso .....	85
const.....	86
COMPONENTES ESTÁTICOS .....	87
Static .....	87
PUNTEROS.....	89
OBJETOS DENTRO DE OBJETOS C++.....	90
JERARQUÍA DE CLASES .....	91
Herencia .....	91
Herencia abstracta .....	91
Compatibilidad de tipos .....	92
Objetos como parámetros.....	92
Parámetros de una función .....	92
Pasar un objeto por valor .....	93
Pasar un objeto de una subclase .....	93
HERENCIA MÚLTIPLE .....	94
ERRAR NO ES DE HUMANOS .....	95
Principales clases de errores .....	97
Stack.....	98
Plantillas .....	99
EXPLICIT EN C++ .....	103
SOLID.....	106
OPERATOR EN C++ .....	108
ESCRITURA DE ARCHIVOS.....	115
FUNCIONES MAS UTILIZADAS.....	121
sizeof().....	121

Getch()	121
System("cls")	121
Getline.....	122
PROYECTO DE CAMPO .....	123
Proyecto Estructurados .....	123
Proyectos orientados a objetos .....	126
Evaluaciones.....	127
Test.....	127
Quiz .....	128
BIBLIOGRAFÍA.....	129
Plataforma OpenEGD .....	129
Libros.....	129
Web .....	129
ANEXOS .....	130
Rubrica para proyecto final .....	130
Formato del proyecto 50% .....	130
Formato del informe final del proyecto .....	130
MATERIAL DE CERTIFICACIÓN .....	131

Tema:

Preguntas & respuesta

1. .

2. .

3. .

4. .

5. .

6. .

7. .

8. .

9. .

10. .

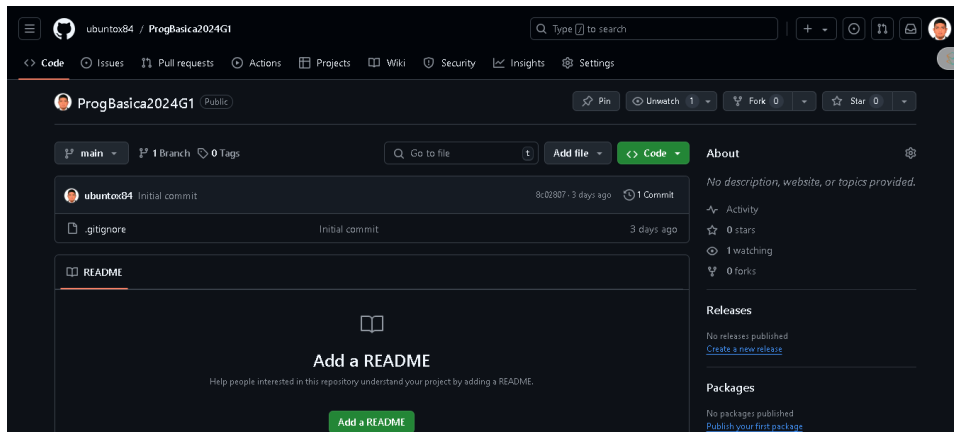
Desarrollo

Aportes

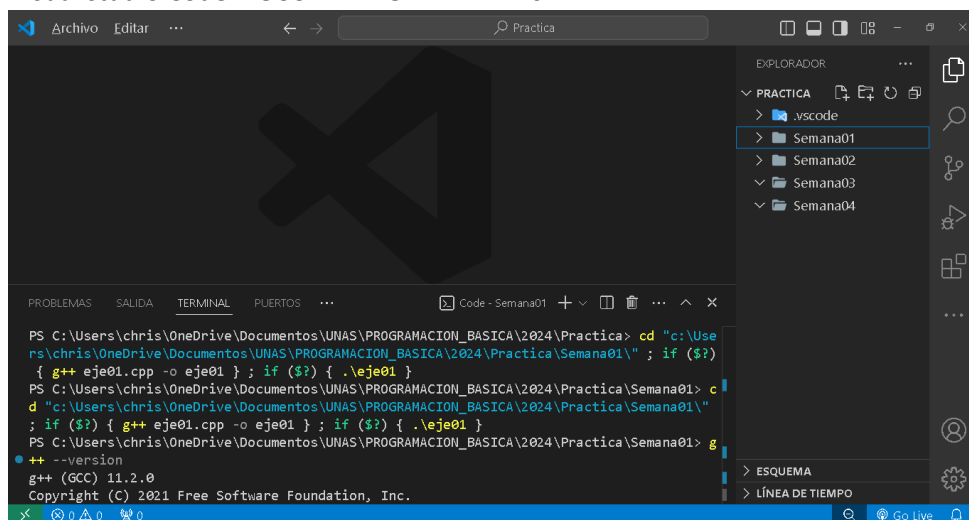
Material de consulta

## Herramienta

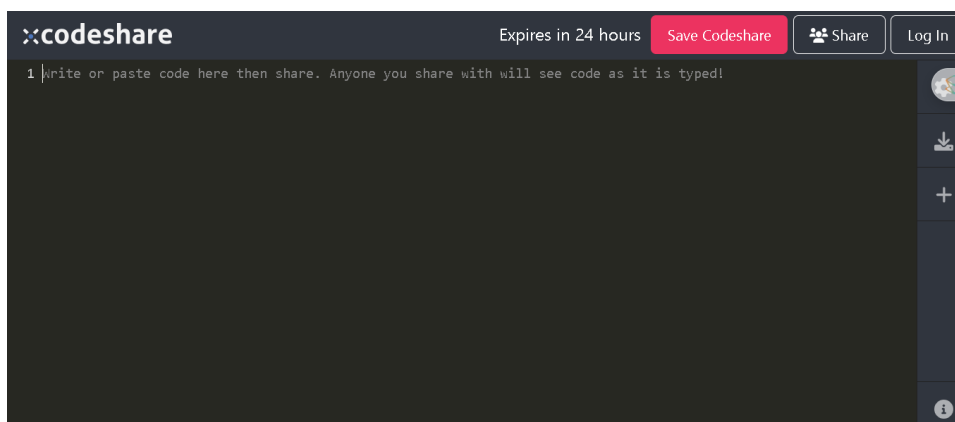
### ➔ Repositorio de código: GITHUB



### ➔ Visual studio Code – GCC – MINGW – TDM264.



### ➔ Sharecode.



### ➔ Microsoft Teams.

## ESTADO DEL ALUMNO

- Entender la lógica de programación (herramienta Psint y C++)

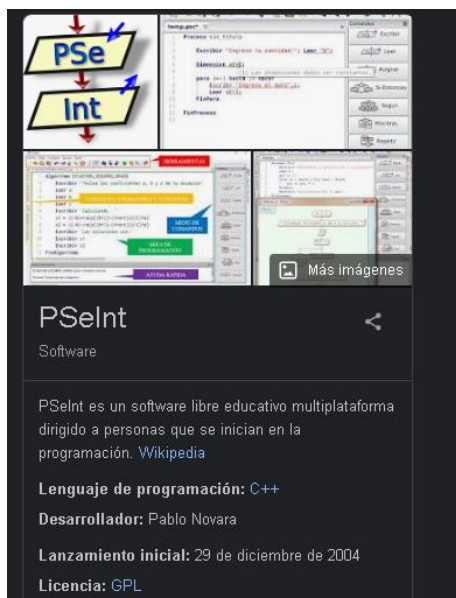
PSeInt es una herramienta educativa que permite a los estudiantes aprender los conceptos básicos de programación utilizando pseudocódigo. PSeInt (Pseudocode Interpreter) es ideal para principiantes porque proporciona un entorno donde los usuarios pueden escribir algoritmos utilizando un lenguaje similar al español, facilitando la transición a lenguajes de programación más avanzados.

### Algunas características clave de PSeInt:

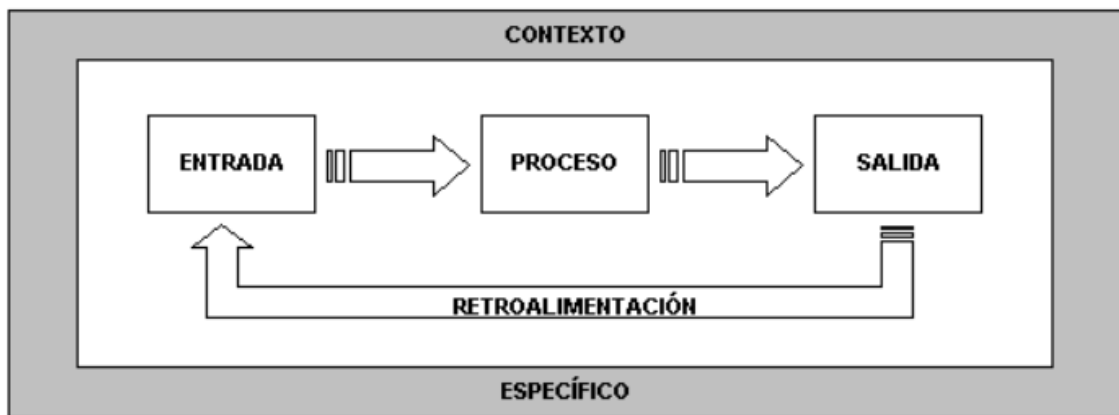
- **Pseudocódigo en español:** Permite escribir algoritmos en un lenguaje que es comprensible para hispanohablantes.
- **Generación de diagramas de flujo:** Automáticamente genera el diagrama de flujo de los algoritmos, lo que facilita la comprensión visual del proceso lógico.
- **Depuración:** Ofrece herramientas para depurar y verificar si los algoritmos funcionan correctamente.
- **Interfaz simple:** Está diseñado con una interfaz amigable para usuarios que están comenzando en la programación.

```
pseudocodigo Copiar código

Algoritmo SumaDosNumeros
    Escribir "Ingrese el primer número: "
    Leer num1
    Escribir "Ingrese el segundo número: "
    Leer num2
    suma = num1 + num2
    Escribir "La suma es: ", suma
FinAlgoritmo
```




## → Caja Negra



### Relación de la caja negra con un programa:

1. **Interfaz y abstracción:** Al trabajar con una caja negra, tienes una interfaz clara que te dice cómo interactuar con el programa o una función. Sabes qué entrada debes proporcionar y qué salida esperar. Los detalles de la lógica interna no son necesarios para el usuario. Por ejemplo, si tienes una función llamada `calcularPromedio`, no necesitas saber los detalles de cómo se realiza la suma o el conteo de elementos, solo necesitas saber qué tipo de datos ingresar (por ejemplo, una lista de números) y esperar que devuelva un valor (el promedio).
2. **Modularidad:** En la programación, los programas se dividen en módulos o funciones más pequeñas. Cada uno de estos módulos puede tratarse como una caja negra. Esto permite que los programadores dividan el trabajo en partes, donde un equipo puede trabajar en una función sin afectar cómo se desarrollan otras partes del programa. La única información que comparten es la interfaz (las entradas y salidas).
3. **Reutilización de código:** La caja negra permite la reutilización de código porque el enfoque está en las entradas y salidas, no en cómo se implementa internamente. Puedes utilizar una función o un módulo sin conocer su código, lo que te ahorra tiempo al no tener que escribirla desde cero o modificarla.
4. **Ejemplo en pseudocódigo:** Imagina que tienes una función que calcula el área de un círculo. No necesitas saber exactamente cómo se realiza el cálculo, simplemente la utilizas como una caja negra:

pseudocodigo


 Copiar código

```
Funcion areaCirculo(radio)
    area = 3.1416 * radio ^ 2
    Retornar area
FinFuncion
```

Cuando utilizas esta función en otro lugar del programa, no te importa cómo se calcula el área, simplemente llamas a la función con el valor del radio:



pseudocodigo

 Copiar código

```
Escribir "Ingrese el radio: "  
Leer radio  
Escribir "El área es: ", areaCirculo(radio)
```

Aquí, la función `areaCirculo` actúa como una **caja negra**. Sabes que debes darle un radio y que te devolverá el área, pero no necesitas preocuparte por los detalles internos.

#### Ventajas de la caja negra:

- **Reducción de la complejidad:** No necesitas saber cómo funciona todo internamente, lo que simplifica la tarea de usar o integrar funciones en un programa.
- **Facilita la colaboración:** Los equipos pueden trabajar en diferentes partes del sistema sin necesidad de conocer los detalles de los otros módulos.
- **Pruebas y depuración más fáciles:** Al probar un módulo como una caja negra, solo se necesita comprobar si la salida es correcta con respecto a las entradas dadas.

En resumen, la caja negra en programación se centra en la **interacción externa** con una función o módulo, ocultando los detalles internos para que puedas concentrarte en los resultados y el uso eficiente del programa.

## 1. Introducción a la programación básica

La programación básica implica aprender a escribir instrucciones que una computadora pueda entender y ejecutar para resolver problemas. Algunos conceptos fundamentales incluyen:

- **Algoritmos:** Un algoritmo es una secuencia de pasos lógicos que permiten resolver un problema. Aprender a escribir algoritmos es uno de los primeros pasos en programación.
- **Lenguajes de programación:** Un lenguaje de programación es una herramienta que permite traducir los algoritmos en instrucciones que la computadora puede entender. Ejemplos de lenguajes básicos incluyen Python, JavaScript, C++, etc.
- **Variables:** Son espacios de memoria que guardan información. Por ejemplo, puedes tener una variable edad que guarde el valor de la edad de una persona.
- **Estructuras de control:** Son instrucciones que alteran el flujo de un programa, como condicionales (if-else) y bucles (while, for), que permiten realizar decisiones y repeticiones dentro del programa.
- **Funciones:** Las funciones permiten encapsular código para que pueda ser reutilizado. Una función recibe entradas, realiza un proceso y devuelve un resultado.
- **Entrada y salida:** Interactuar con el usuario a través de entradas (datos proporcionados por el usuario) y salidas (resultados mostrados al usuario).

Un ejemplo de código básico en Python sería:

```
Un ejemplo de código básico en Python sería:
```

```
python Copiar código
```

```
# Este programa pide el nombre del usuario y lo saluda
nombre = input("¿Cuál es tu nombre? ")
print("Hola, " + nombre + "!")
```

## 2. Compartiendo experiencia

Como principiante en programación, es importante ser consciente de los siguientes puntos:

- **Comenzar con problemas pequeños:** Al inicio, es fácil sentirse abrumado. En lugar de intentar resolver grandes problemas desde el principio, empieza con programas simples y gradualmente aumenta la complejidad.
- **Errores son normales:** Los errores o "bugs" son parte natural del proceso de aprendizaje. No te frustres cuando el código no funcione como esperabas. La depuración (debugging) es una habilidad clave que todo programador debe desarrollar.
- **Practicar constantemente:** La programación es una habilidad práctica. Es necesario escribir mucho código para ganar fluidez y comprender cómo funcionan los conceptos básicos.

- **Buscar recursos y apoyo:** Existen muchos recursos en línea (cursos, tutoriales, foros) y comunidades que pueden ayudarte. Compartir tus dudas con otros programadores y aprender de sus experiencias acelera el proceso.
- **Aprender de ejemplos reales:** Mirar ejemplos de código te ayudará a comprender cómo se resuelven problemas comunes. Al modificar y experimentar con estos ejemplos, mejorarás tus habilidades.

### 3. Reconociendo la importancia

La programación es una herramienta poderosa que permite la creación de soluciones automáticas, desde simples aplicaciones hasta sistemas complejos. Algunas razones clave por las que la programación es importante:

- **Desarrollo de habilidades de resolución de problemas:** Programar te enseña a pensar de manera lógica y estructurada, descomponiendo problemas grandes en tareas más pequeñas y manejables.
- **Automatización:** Con la programación, puedes automatizar tareas repetitivas, lo que ahorra tiempo y esfuerzo en muchos contextos.
- **Aplicaciones prácticas:** Aprender a programar abre puertas a muchas oportunidades laborales y también te permite crear soluciones personalizadas para problemas en tu vida diaria, como aplicaciones, herramientas o scripts.
- **Innovación y creatividad:** Programar te permite ser creativo y crear proyectos desde cero. Puede ir desde el desarrollo de una aplicación móvil hasta la construcción de un sistema de inteligencia artificial o la creación de videojuegos.
- **Demanda laboral:** En el mercado laboral actual, saber programar es una habilidad altamente valorada. La demanda de desarrolladores y profesionales de tecnología está creciendo rápidamente.

## RECOMENDACIÓN

- Acceso a una computadora con herramientas para la clase.
- Programar todos los días (solucionar un problema o leer sobre tecnología)
- Tener el editor de código Visual Studio Code.
- Aprender 1 lenguajes, pero debemos ser especialistas en un lenguaje.
- Para reducir el tiempo en entender sobre un tema específico “infograma”.
- Para buscar información usted debe colocar el año actual para evitar problemas de versiones o cambios de sintaxis.

## Paradigmas en informática











- Forma de escribir código

### Algunas de ellas:

- Programación imperativa
- programación orientada objetos
- programación basada en reglas

## Lenguajes de programación

- es un conjunto de reglas y símbolos que permiten a una persona dar instrucciones a una computadora.

		<a href="#">About us</a> <a href="#">Knowledge</a> <a href="#">News</a> <a href="#">Coding Standards</a> <a href="#">TIOBE Index</a> <a href="#">Contact</a> <a href="#">Schedule a demo</a>				
Dec 2023	Dec 2022	Change	Programming Language		Ratings	Change
1	1			Python	13.86%	-2.80%
2	2			C	11.44%	-5.12%
3	3			C++	10.01%	-1.92%
4	4			Java	7.99%	-3.83%
5	5			C#	7.30%	+2.38%
6	7	▲		JavaScript	2.90%	-0.30%
7	10	▲		PHP	2.01%	+0.39%
8	6	▼		Visual Basic	1.82%	-2.12%
9	8	▼		SQL	1.61%	-0.61%

# PROGRAMA

Es el proceso de crear algoritmos y sistemas usando lenguajes de programación para comunicarnos con las computadoras y poder crear programas.

¿Por qué debemos aprender C++?

- En uno de los lenguajes más utilizados para aprender a programar
- Tiene programación imperativa y orientada objetos

## **Veloz**

Manejo de memoria sin maquina virtual ni recolector de basura.

## **Complejo**

Una gran cantidad de librerías y nuevos paradigmas, constantemente en crecimiento y nunca termina.

Usado en...

Usado para aplicaciones que requieren de una optimización de recursos y manejo de memoria como prioridad, al mismo tiempo de ser escalable, modular y distribuido en varias plataformas.

## Tipados & no tipados



## Compilado, interpretado y intermedio



Compilar es ...



Y el compilador de c++ es:





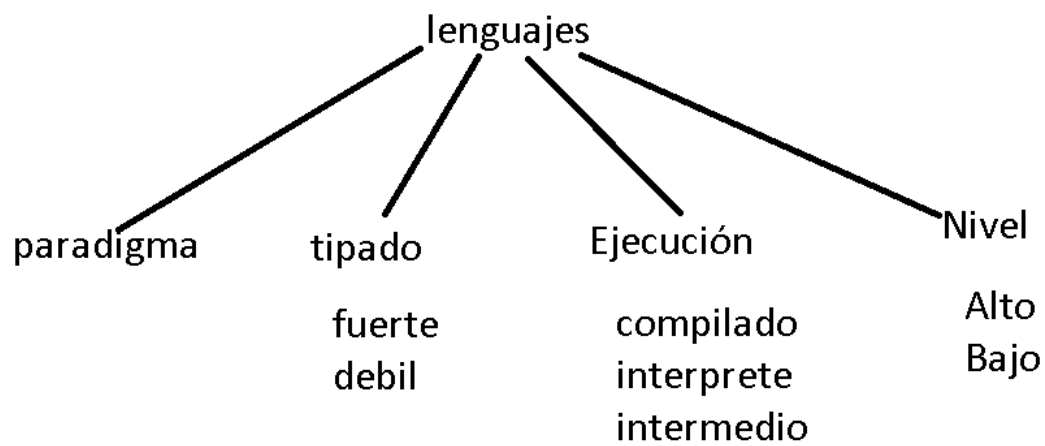
Es open source y se mantiene actualizado.

## Paradigmas de programación





En conclusión, sobre los lenguajes de programación



## Editores y entornos de desarrollo

**EDITOR VS IDE**

Con ambos puedes escribir código, pero ¿en qué se diferencian?

EDITOR	IDE
 Software ligero con ayudas para escribir código (resaltado de sintaxis, autocompletado, etc).	 Integra un editor con las herramientas que necesita un desarrollador (debugger, compilador, etc).
 Soporta <b>múltiples lenguajes</b> y tecnologías.	 Se especializa en <b>un lenguaje o tecnología</b> (Java, Python, Go, Android, etc).
 <b>Enfocado en archivos</b> (no tienen el concepto de proyecto).	 <b>Enfocado en proyectos completos.</b> Desde la primera línea hasta la salida a producción.
 <b>Puedes agregar plugins</b> para darle el poder de un IDE pero te toca configurar cada uno a mano.	 <b>Trae herramientas integradas y configuradas</b> (ej. Android Studio trae un emulador de Android).
<b>EJEMPLOS DE EDITORES</b> 	<b>EJEMPLOS DE IDES</b> 

Domina la tecnología con EDteam y #NuncaTeDetengas  
[ed.team/cursos](https://ed.team/cursos)

EDteam

## HERRAMIENTA SELECCIONADA

es: Visual Studio Code



## Tipos de errores

### 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.: Dejar 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.

## HERRAMIENTAS EN LÍNEA

- <https://pythontutor.com/>
- <https://cpp.sh/>

## PARA COMPARTIR CÓDIGO



Más imágenes

### Código compartido

En la industria de aviación civil, un acuerdo de código compartido es un acuerdo suscrito por dos o más aerolíneas para explotar conjuntamente una determinada ruta. De tal forma, todas las aerolíneas venden asientos de un mismo vuelo y este tiene varios números de vuelo distintos, uno para cada compañía. Wikipedia

<https://codeshare.io>

## Herramienta colaborativa



<https://replit.com/>

## MI PRIMER PROGRAMA

```
Semana01 > C++ Ejer01.cpp > main()
1  #include<iostream>
2  using namespace std;
3  int main(){
4      cout<<"It's, your first program"<<endl;
5      return 0;
6  }
```

Podemos agregar arte dentro del código utilizando: ASCII Art (<https://www.asciart.eu/> )

## OPERADORES EN C++

los operadores son símbolos especiales que permiten realizar operaciones en variables y valores. Hay diferentes tipos de operadores en C++, como operadores aritméticos, operadores de asignación, operadores de comparación, operadores lógicos, entre otros. Aquí tienes algunos ejemplos de operadores en C++:

### 1- Operadores aritméticos:

Suma: +

Resta: -

Multipliación: \*

División: /

Módulo: %

### 2- Operadores de asignación:

Asignación: =

Asignación con suma: +=                      (a=a+b) == (a+=b)

Asignación con resta: -=                      (a=a-b) == (a-=b)

Asignación con multiplicación: \*=                      (a=a\*b)==(a\*=b)

Asignación con división: /=                      (a=a/b)== (a/=b)

### 3- Operadores de comparación:

Igual a: ==

Diferente de: !=                      <>

Mayor que: >

Menor que: <

Mayor o igual que: >=

Menor o igual que: <=

### 4- Operadores lógicos:

AND lógico: &&

OR lógico: ||

NOT lógico: !

5- Operadores de incremento y decremento:

Incremento: ++ (agregar +1) (b=b+1==++b) b=5 → ++b → 6

++b

b++

Decremento: -- (disminuye -1) (b=b-1 == --b) b=10 → --b → 9

6- Operadores de acceso a miembros:

Acceso a miembro de objeto: objeto.miembro

Acceso a miembro de puntero a objeto: puntero->miembro

Estos son solo algunos ejemplos de operadores en C++. Cada operador tiene su propia función y sintaxis, y se utilizan en diferentes contextos para realizar operaciones específicas.

### Prioridades de operadores en C++

Aquí tienes una tabla que muestra la prioridad de algunos operadores comunes en C++:

Prioridad	Operadores
1	. - >
2	++ --
3	typeid dynamic_cast static_cast const_cast reinterpret_cast
4	* / %
5	+ -
6	<< >>
7	< > <= >=
8	== !=
9	&&
10	
11	? :
12	= += -= *= /= %=
13	,

El rol de paréntesis es utilizado para asignar operaciones y no dejar que el compilador utilice la tabla de prioridades.

```
#include <iostream>
using namespace std;
int main()
{
    int a=5,b=4,c=2;
    int resp;
    resp=((a*(b/c))/2);
    cout<<resp;
    return 0;
}
```

Operadores ternarios

```
cpp Copy code

condición ? expresión_si_cierto : expresión_si_falso;
```

Ejemplo:

```
#include <iostream>
using namespace std;
int main() {
    int alumnos = 10;

    cout << "Hay " << (alumnos > 15 ? "muchos alumnos" : "pocos alumnos") << endl;

    return 0;
}
```

**Determinación del signo de un número:**

```
cpp Copy code

int numero = -7;
std::string signo = (numero >= 0) ? "Positivo" : "Negativo";
// Se determina el signo del número
```

ÁMBITOS



Normalmente esta representado por las llaves ( )

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int numero = -7;
6      int p;
7      p+=numero;
8      {
9          int y=100;
10     }
11     int respuesta=p+numero+y;
12     {
13         int numero;
14     }
15     return 0;
16 }
```

y no esta en el ambito de las  
variables p y numero

## COMENTARIOS

- Una sola línea

```
//Hola mundo
```

- Un párrafo

```
/*
    FIIS
    -
    UNAS
*/
```



# TIPOS DE DATOS

Tabla 2.1. Tipos de datos simples de C++

Tipo básico	Tipo	Ejemplo	Tamaño en bytes	Rango. Mínimo..Máximo
Carácter	char	'C'	1	0..255
Entero	short	-15	2	-128..127
	int	1024	2	-32768..32767
	unsigned int	42325	2	0..65535
	long	262144	4	-2147483648..2147483637
Real	float	10.5	4	$3.4 \times (10^{-38}) \dots 3.4 \times (10^{38})$
	double	0.00045	8	$2.7 \times (10^{-308}) \dots 2.7 \times (10^{308})$
	long double	1e-8	8	igual que double

## ➔ TIPO DE DATOS PRIMITIVOS

### - Enteros (Integer Types):

int: Tipo de dato entero con signo.

short: Entero corto con signo.

long: Entero largo con signo.

unsigned int: Entero sin signo.

long long: Entero largo largo con signo.

### - Punto flotante (Floating-Point Types):

float: Número de punto flotante de precisión simple.

double: Número de punto flotante de doble precisión.

long double: Número de punto flotante de doble precisión extendida.

double numeroDecimal = 3.14;

float n=7.0;

float x=.0;

### - Caracteres (Character Types):

char: Carácter con signo.

unsigned char: Carácter sin signo.

wchar\_t: Carácter ampliado de ancho (wide character).

```

#include <iostream>
using namespace std;
int main()
{
    char p='X';
    int x=int(p);
    cout<<x<<"\t"<<p<<endl;

    int y=34;
    char q=(char)y;
    cout<<y<<"\t"<<q;
    return 0;
}

```

## - **Booleano:**

**bool:** Puede almacenar solo valores true o false.

```

1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      bool var1;
6      // True - False
7      //cualquier valor numerico - 0
8      bool var2=0;
9      if(0){
10         cout<<"TRUE";
11     }else{
12         cout<<"FALSE";
13     }
14     return 0;
15 }

```

## TIPO DE DATOS COMPUESTOS

### - **Arrays:** Colección de elementos del mismo tipo.

```

#include <iostream>
using namespace std;
int main()
{
    int n1,n2,n3,n4,n5,n6,n7,n8,n9,n10;
    n1=10;
    n2=12;
    n3=15;
    // Array
    int notas[10];
    notas[0]=5;
    notas[4]=7;
    return 0;
}

```

- **Structs: Agrupación de variables bajo un mismo nombre.**

```

#include <iostream>
using namespace std;
struct student
{
    string name;
    int code;
    int year;
    char sex;
};
int main()
{
    student e1;
    e1.name="Lois";
    e1.code=0020230471;
    e1.year=17;
    e1.sex='M';
    return 0;
}

```

- **Classes: Similar a struct, pero con la capacidad de encapsulación y herencia.**

```

#include <iostream>
using namespace std;

class student {
public:
    string name;
    int year;
    void student() {
        // código para estudiar
    }
};

int main()
{
    return 0;
}

```

Punteros: Almacenan direcciones de memoria.

```

1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int x=10;
6      int *p;
7      p=&x;
8      cout<<"Valor de X:\t"<<x<<endl;
9      cout<<"Direccion de X:\t"<<&x<<endl;
10     cout<<"P:\t"<<p<<endl;
11     cout<<"Valor de P:\t"<<*p<<endl;
12     *p=20;
13     cout<<"Valor de X:\t"<<x<<endl;
14     cout<<"Direccion de X:\t"<<&x<<endl;
15     return 0;
16 }

```

Enum: Definición de un conjunto de constantes con nombre.

```
1  #include <iostream>
2  using namespace std;
3  enum seasons
4  {
5      autumn,    winter,    spring,    summer
6  };
7  int main()
8  {
9      seasons estacionActual = winter;
10     switch (estacionActual)
11     {
12     case autumn:
13         cout << "Es otoño";
14         break;
15     case winter:
16         cout << "Es invierno";
17
18         break;
19     case spring:
20         cout << "Es primavera";
21         break;
22     case summer:
23         cout << "Es VERANO";
24         break;
25     }
26     return 0;
27 }
28
```

## NOTACIÓN CIENTÍFICA

Con enteros

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      float num1 = 900000000; //  $9 \times 10^8$ 
6      cout << "Número 1: " << scientific << num1 << endl;
7      cout << "Número 1: " << fixed << num1 << endl;
8  }
```

Número 1: 9.000000e+08

Número 1: 900000000.000000

Con decimales

```
#include <iostream>
using namespace std;
int main()
{
    float num1 = 1.5e-3; //  $1.5 \times 10^{-3}$ 
    cout << "Número 1: " << scientific << num1 << endl;
    cout << "Número 1: " << fixed << num1 << endl;
}
```

Número 1: 1.500000e-03

Número 1: 0.001500

# LECTURA Y ESCRITURA

## Entrada y salida estándar

se realiza a través de la pantalla. Se utilizan las funciones cin y cout respectivamente para llevar a cabo estas operaciones. Por ejemplo:

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

int main() {
    int num;
    cout << "Ingrese un número: ";
    cin >> num;
    cout << "El número ingresado es: " << num;
    return 0;
}
```

No olvidar colocar la librería “**iostream**”

Entendiéndose como librería: conjunto de recursos o funcionalidades que son utilizados para simplificar el desarrollo de software. Esto permite que los desarrolladores puedan enfocarse en la lógica específica del programa que están creando, en lugar de tener que preocuparse por implementar funciones básicas que ya están disponibles en una librería.

## Instrucciones de entrada y salida de datos

Se manifiestan 02 formas de ingresar o asignar datos a una variable

- **Cout**, muestra valores en consola
- **Cin.-** permite el ingreso de datos a una variable.

existen otras funciones de entrada y salida disponibles en la biblioteca estándar, tales como:

- `getline`: lee una línea completa de texto desde un flujo de entrada.
- `getchar`: lee un solo carácter desde un flujo de entrada.
- `putc`: escribe un carácter en un flujo de salida.
- `puts`: escribe una cadena de caracteres en un flujo de salida.
- `fstream`: proporciona una interfaz para la entrada y salida de archivos.
- `stringstream`: proporciona una interfaz para la lectura y escritura de cadenas de caracteres en memoria como si fueran archivos.

## scanf - printf

`scanf`: se utiliza para leer entradas formateadas desde la consola.

`printf`: se utiliza para imprimir texto formateado en la consola.

Por ejemplo:

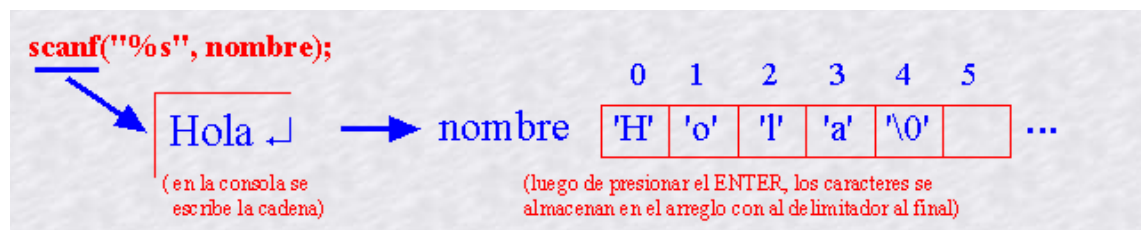
```
1  #include <stdio.h>
2
3  int main()
4  {
5      int n1, n2;
6      int resp = 0;
7
8      printf("ing. un numero 1:");
9      scanf("%d", &n1);
10     printf("ing. un numero 2:");
11     scanf("%d", &n2);
12
13     resp = n1 + n2;
14
15     printf("La suma de %d y %d es: %d\n", n1, n2, resp);
16     return 0;
17 }
```



```
#include <stdio.h>

int main() {
    int edad;
    printf("Ingresa tu edad: ");
    scanf("%d", &edad);
    printf("Tu edad es: %d\n", edad);
    return 0;
}
```

- **Scanf**.- permite el ingreso de datos desde la consola.



## gets - puts

Permiten el ingreso y la salida de cadenas de caracteres son las funciones gets y puts. La sintaxis de estas funciones es la siguiente:

```
char * gets(char *);
int puts (char *);
```

Mi segundo programa: sumar dos variables

```
1  #include<iostream>
2  using namespace std;
3  int main(){
4      //entrada
5      int a,b, suma;
6      //proceso inv (estudio)
7      cout<<"Intro. A:"<<endl;
8      cin>>a;
9      cout<<"Intro. B:"<<endl;
10     cin>>b;
11     suma=a+b;
12     //salida
13     cout<<"La suma de "<<a<<" + "<<b<<" es igual a "<<suma;
14     //prueba
15     if((a+b)==suma){cout<<"\nCo (const char [24])"Error!! en la operacion"
16     }else{cout<<"Error!! en la operacion";
17     }
18     return 0;
19 }
```

Comunicación

- <https://slack.com/intl/es-pe>
- <https://discord.com/>

Para la lluvia de ideas pueden utilizar

- <https://miro.com/es/>

## VARIABLES Y TIPOS DE DATOS

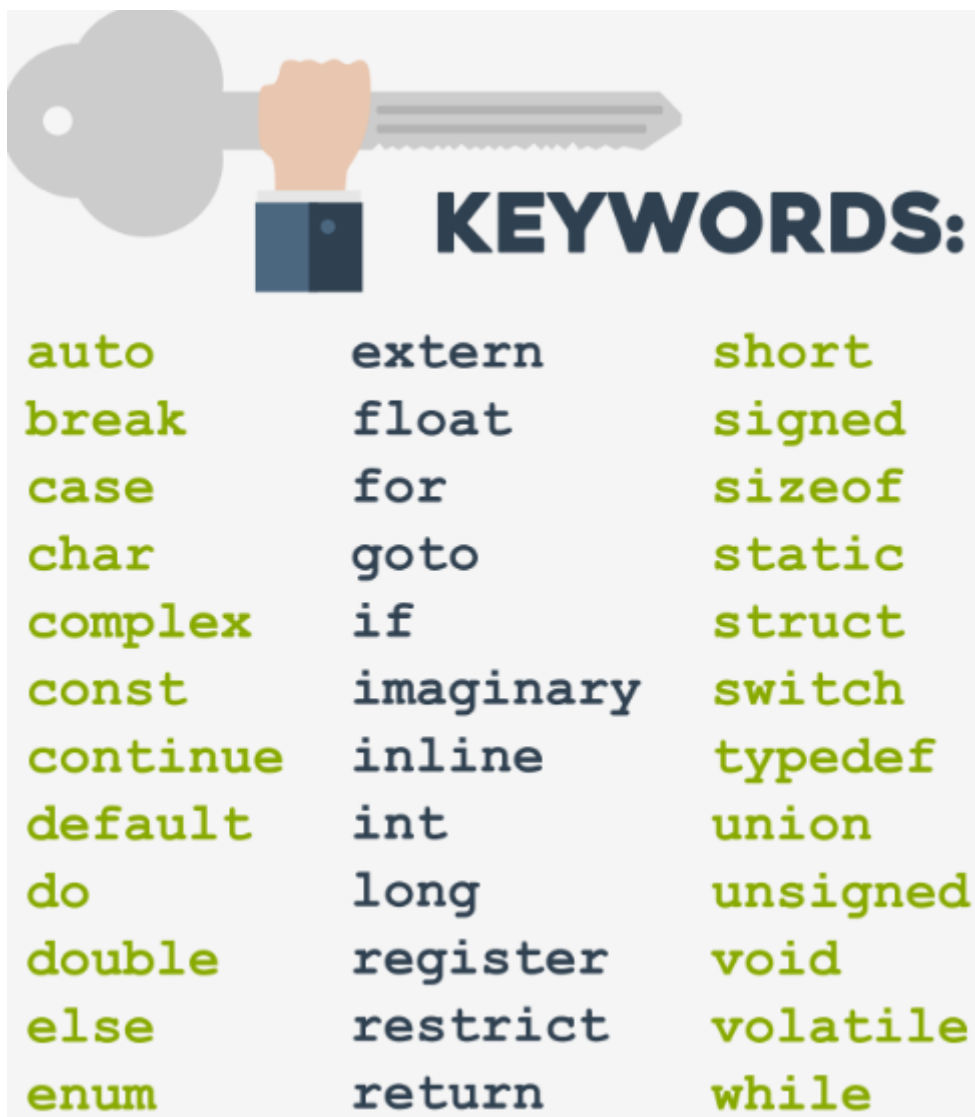
Las características para escribir una variable son las siguientes :

- el nombre de la variable debe estar compuesto por **letras latinas mayúsculas o minúsculas, dígitos y el carácter \_ (guión bajo)**;
- El nombre de la variable debe **comenzar con una letra**;
- el **carácter subrayado es una letra** (extraño pero cierto);
- las letras mayúsculas y minúsculas se tratan como diferentes (un poco diferente que en el mundo real: Alice y ALICE son los mismos nombres de pila, pero son dos nombres de variables diferentes, en consecuencia, dos variables **diferentes**);

### PALABRAS CLAVES

Son palabras parte del lenguaje de programación:

Int, float, double, ...



## ¿Qué son tipos de datos?

Es una clasificación que le damos a la información para hacer saber al compilador cómo va a ser escrita e interpretada.

### Tipos de datos primitivos

Nombre	Descripción	Tamaño	Rango
char	Caracter (código ASCII)	8 bits	Con signo: -128 ... 127 Sin signo: 0 ... 255
short int (short)	Número Entero corto	16 bits	Con signo: -32768 ... 32767 Sin signo: 0 ... 65535
int	Número Entero	32 bits	Con signo: -2147483648 ... 2147483647 Sin signo: 0 ... 4294967295
long int (long)	Número Entero largo	64 bits	Con signo: -9223372036854775808, 9223372036854775807 Sin signo: 0 ... 18446744073709551615
float	Número real	32 bits	$3,4 \times 10^{-38}$ ... $3,4 \times 10^{+38}$ (6 decimales)
double	Número real en doble precisión	64 bits	$1,7 \times 10^{-308}$ ... $1,7 \times 10^{+308}$ (15 decimales)
long double	Número real largo de doble precisión	80 bits	$3,4 \times 10^{-4932}$ ... $1,1 \times 10^{+4932}$
bool	Valor booleano	1 bit	true (VERDADERO) o false (FALSO)

### Ejemplo:

```
#include <iostream>

using namespace std;

int main()
{
    cout << 10 << endl;
    cout << 3.1416f << endl;
    cout << true << endl;
    cout << 'c' << endl;

    return 0;
}
```

### Alojando una variable en memoria

Una variable es un **espacio reservado** en memoria, definido por un **tipo de dato** y un **nombre asignado**, en el cual se puede **guardar un valor** y se puede **modificar**.

## Operadores aritméticos

Operador	Operación u Significado
+	Suma
-	Resta
*	Multiplicación
/	División
%	Modulo
=	Igual (Asignación de valor)
++	Incremento en 1
--	Decremento en uno
+=	Suma y asignación
-=	Resta y asignación
*=	Multiplicación y asignación
/=	División y asignación

## Apuntadores

Un apuntador es una variable que guarda una dirección de memoria.

La referencia a esta dirección de memoria puede ser usada para modificar el valor dentro ella..

```
int Direccion = 3509;
int* ApuntadorADir;
ApuntadorADir = &Direccion;
*ApuntadorADir = 3; | I
cout << Direccion << endl;
```

Ahora ingresamos datos <iostream>

### Cout

Comando de flujo de salida de datos (Generalmente en la consola)

### Cin

Comando de flujo de entrada de datos. (El programa espera hasta que los datos sean introducidos)

## Operadores lógicos

### Menor que (<)

1 < 10 (Verdadero)

### Mayor que (>)

1 > 10 (Falso)

### Igualacion (==)

100 == 100 (Verdadero)

### Menor que, igual que(<=)

10 <= 10 (Verdadero)

### Mayor que, igual que (>=)

11 >= 10 (Verdadero)

### No Igual (!=)

100 != 100 (Falso)

## ¿Qué son las condicionales?

Son estructuras de control que nos permiten manipular el flujo de nuestro programa dependiendo de ciertas condiciones establecidas por el programador.

## If – else

```
bool band;
if(band){

}else if(){

}else{

}
```

## Condicionales anidadas

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int x;
6      cout<<"ingresar valor";
7      cin>>x;
8      if(x==0)
9          cout<<"Es cero";
10         else if(x>0)
11             cout<<x<<" es Mayor a cero";
12         else
13             cout<<x<<" es Menor a cero";
14         return 0;
15     }
```

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int x;
6      cout<<"ingresar valor";
7      cin>>x;
8      if(x==0)
9          cout<<"Es cero";
10         if(x>0)
11             cout<<x<<" es Mayor a cero";
12         if(x<0)
13             cout<<x<<" es Menor a cero";
14         return 0;
15     }
```

## Hallar el mayor de 3 números

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int a, b, c;
6      int max;
7      cout << "Ingresa 3 numeros: " << endl;
8      cin >> a >> b >> c;
9      max = a;
10     if (max < b)
11         max = b;
12     if (max < c)
13         max = c;
14     cout << "El número mayor es " << max << endl;
15     return 0;
16 }
```

## Switch

```
int expression=0;
switch (expression)
{
case 1:
    break;
case 2:
    break;
default:
    break;
}
```



## Condicionales

Se utilizan cuando tomamos decisiones, por ejemplo; cuando ingresamos a un sistema de autenticación, se pide usuario y contraseña se necesita de una condicional para verificar lo datos ingresados.

- If – else
  - Tipos
    - Simples
    - Compuestos
    - Cascada
- Switch (se requiere de un selector)

## ¿Qué son loops? (Bucles)

Permite repetir una o varias veces un bloque de comandos. Se establecen condiciones para establecer el numero de veces que se repiten.

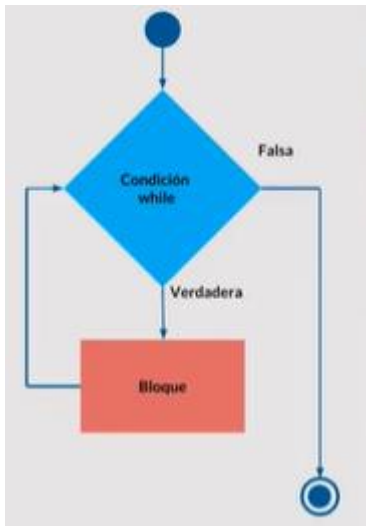
Evaluando la necesidad de utilizar bucles, se pide sumar 7 numeros:

```
1  #include <stdlib.h>
2  #include <time.h>
3  #include<iostream>
4  using namespace std;
5  int main()
6  {
7      int num1,num2,num3,num4,num5,num6,num7;
8      cout<<"ingresar Num";
9      cin>>num1;
10     cout<<"ingresar Num";
11     cin>>num2;
12     cout<<"ingresar Num";
13     cin>>num3;
14     cout<<"ingresar Num";
15     cin>>num4;
16     cout<<"ingresar Num";
17     cin>>num5;
18     cout<<"ingresar Num";
19     cin>>num6;
20     cout<<"ingresar Num";
21     cin>>num7;
22     int suma=num1+num2+num3+num4+num5+num6+num7;
23     cout<<"La suma es "<<suma;|
24 }
```

Todo esto se puede reducir en el siguiente código como se muestra a continuación: por ejemplo, solicitar 10 números.

```
1  #include <stdlib.h>
2  #include <time.h>
3  #include<iostream>
4  using namespace std;
5  int main()
6  {
7      int num1, suma=0;
8      for (int i = 0; i < 7; i++)
9      {
10         cout<<"ingresar Num";
11         cin>>num1;
12         suma+=num1;
13     }
14     cout<<"La suma es "<<suma;
15 }
```

- While



1- Utilizar un contador para crear un límite de ejecución con el bucle while

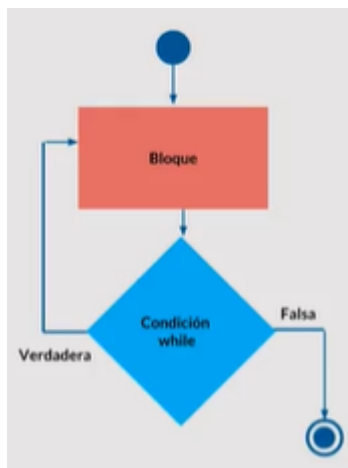
```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int counter=0;
6      while(counter<9){
7          cout<<"hello"<<endl;
8          ++counter;
9      }
10     return 0;
11 }
```

2- Utilizar un break para detener los bucles

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int counter = 0;
6      while (true)
7      {
8          if (counter == 9)
9              break;
10         cout << "hello" << endl;
11         ++counter;
12     }
13     return 0;
14 }
```

```
1  #include <stdlib.h>
2  #include <time.h>
3  #include<iostream>
4  using namespace std;
5  int main()
6  {
7      int num1, suma=0, cont=0;
8      while(cont<7){
9          cout<<"ingresar Num";
10         cin>>num1;
11         suma+=num1;
12         cont++;
13     }
14     cout<<"La suma es "<<suma;
15 }
```

- Do while



```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int n1, n2, answer;
6      char op;
7      do
8      {
9          cout << "write 02 number:" << endl;
10         cin >> n1 >> n2;
11         answer = n1 + n2;
12         cout << "the sum of " << n1 << " + " << n2 << " is equal to " << answer<<endl;
13
14         cout<<"¿Do you wish to continue? (s/n):";
15         cin>>op;
16         op=tolower(op);
17     }while(op=='s');
18     cout<<"see you soon!!";
19     return 0;
20 }
```

```

1  #include <stdlib.h>
2  #include <time.h>
3  #include<iostream>
4  using namespace std;
5  int main()
6  {
7      int num1, suma=0, cont=0;
8      do{
9          cout<<"ingresar Num";
10         cin>>num1;
11         suma+=num1;
12         cont++;
13     }while(cont<7);
14     cout<<"La suma es "<<suma;
15 }

```

- For: Tiene una estructura y tiene 3 separaciones



Ejemplo: Crear un factorial de un numero

```

1  #include<iostream>
2  using namespace std;
3  int main()
4  {
5      int n,fact=1;
6      cout<<"ingrese un Nro: ";
7      cin>>n;
8      //5!=5*4*3*2*1;
9      //0!=1
10     for (size_t i = 1; i <=n; i++)
11         fact*=i;
12     cout<<"el Factorial de "<<n<<" es "<<fact;
13 }

```

### ¿Qué son los arreglos?

Estos tienen un tratamiento especial y se mantienen las siguientes características:

- Almacenan un solo tipo de dato
- Tiene una dimensión finita

Se pueden declarar de dos formas:

- `vector<int> lista(5);` (para esto se utiliza librería "vector")
- `int lista[5];`

Declarar de forma explícita:

```

3  #include<iostream>
4  using namespace std;
5  int main()
6  {
7      string lista[5]={"Juan", "Manuel", "Luis", "Jose", "Antonio"};
8      for(string cad:lista)
9          cout<<cad<<"\t";
10
11 }

```

Ejemplo: Llenar el vector "lista" de dimensión 5, se pide: ingresar vector y mostrar.



```

1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  int main()
5  {
6      vector<int> lista(5);
7      for (size_t i = 0; i < lista.size(); i++)
8      {
9          cout<<"ingrese numero: ";
10         cin>>lista[i];
11     }
12     for (size_t i = 0; i < lista.size(); i++)
13         cout<<lista[i]<<"\t";
14 }

```

Ahora nos toca aprender a generar números aleatorios, esto se utilizará

```

1  #include <stdlib.h> ←
2  #include <time.h> ←
3  #include <iostream>
4  using namespace std;
5  int main()
6  {
7      int num, c;
8      srand(time(NULL)); ←
9      for(c = 1; c <= 10; c++)
10     {
11         num = 1 + rand() % (11 - 1);
12         cout << num<< " \n";
13     }
14     //system("pause");
15     return 0;
16 }

```

Este código será integrando a un vector.

Ejemplo: Llenar el vector "lista" de dimensión 5, los datos serán ingresado de forma aleatoria, se pide: ingresar vector y mostrar.

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  int main()
5  {
6      vector<int> lista(5);
7      srand(time(NULL));
8      for (size_t i = 0; i < lista.size(); i++)
9          lista[i]=1 + rand() % (11 - 1);
10     for (size_t i = 0; i < lista.size(); i++)
11         cout<<lista[i]<<"\t";
12 }
```

# ESTRUCTURAS

## Union

- es un tipo de datos derivado, como una estructura, con miembros que comparten el mismo espacio de almacenamiento.
- Fue el primer paso para continuar con las estructuras.

¿Qué es?.- Una unión es una estructura con características especiales. La diferencia radica en que cada uno de los campos definidos en ella comparten el mismo espacio de memoria. Esto quiere decir que, si definimos una variable de este tipo, a pesar de que tenga varios campos, sólo uno podrá almacenar datos a la vez.

## Struct

¿Qué es?

¿Cómo lo utilizamos?

## **ARREGLO CON VECTORES**

- Vectores.- colección de elementos, debemos definir el tamaño y tipo.

# **ORDENAMIENTO DE VECTORES**

Se desarrolla de 02 formas:

## **Iterativos**

para su ejecución se utiliza bucles

- Ordenamiento burbuja mejorado (Bubble sort)
- Ordenamiento inserción
- Ordenamiento selección

## **Recursivo**

-

## Constantes en C++

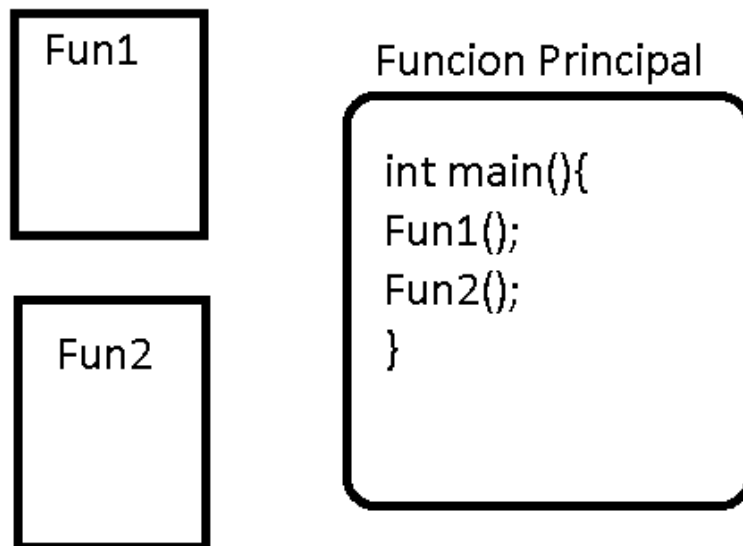
Son tipos de variables que no cambian su dato en ejecución, mal llamado contantes, esto se puede definir de dos formas:

```
1  #include <iostream>
2  #define PI 3.1416;
3  using namespace std;
4  int main()
5  {
6      const float R=2.5;
7      cout<<R<<endl;
8      cout<<PI;
9  }
```

Realmente al usar la instrucción **#define** en C++ no estamos creando una constante realmente, estamos creando una expresión, Si intentamos ejecutar el operador << justo después de PI no da un Error, esto sucede porque PI no es tratado exactamente como una variable cualquiera sino como una expresión, así que realmente aunque podemos usar #define para declarar constantes no es la mejor opción.

La instrucción **const** nos permite declarar constantes de una manera más adecuada y acorde.

## Funciones gráficamente



## Consideraciones:

- Se respeta el orden de ejecución
- Devuelve un tipo de dato

```
#include <iostream>
using namespace std;
void print(string msn){
    cout<<msn<<endl;
}
int main(){
    print("Hola");
    print("Buen dia");
    print("Como estas!");
    return 0;
}
```

Reto: crear un contexto donde se tome decisiones, en primer lugar el sistema debe describir el contexto, de acuerdo a las decisiones tomadas por el usuario llegara a un fin descrito.

# **FUNCIONES CON CADENA DE CARACTERES**

Librería: string.h

¿Qué es? Es la manipulación de cadena de caracteres

¿Utiliza alguna librería?

Funciones:

- Strlen(), longitud de una cadena.
- Strcpy(), copia una cadena en otra
- Strcmp(), compara cadenas
  - o Cad1>cad2 → mayor a cero
  - o Cad1<cad2 → menor a cero
  - o Cad1==cad2 → 0
- Strcat(), concatena cadenas

Se agrega un "n", para agregar cierto número de elementos de la cadena a comparar.

- Strncpy(), copia determinado numero de caracteres
- Strncmp(), comparar determinado numero de caracteres
- Strncat(), concatena determinado numero de caracteres
- Strtok(), busca dentro de una cadena de caracteres.



## CONVERSIONES

Conversiones implícitas

Conversiones explícitas

## **CADENAS**

```
string cad;
```

```
concatenar cadenas;
```

```
cad= cad + cad;
```

## FUNCIONES RECURSIVAS

Son aquellas que se llaman a sí mismo dentro de su propia definición, con el fin de resolver un problema de forma eficiente y elegante.

Ejemplo:

En este ejemplo, la función factorial es una función recursiva que toma un número  $n$  como argumento y devuelve su factorial. La función comprueba si  $n$  es igual a 0, en cuyo caso devuelve 1 (ya que  $0! = 1$ ). Si  $n$  no es igual a 0, la función llama a sí misma con el argumento  $n-1$  y multiplica el resultado por  $n$ . Esto se repite hasta que  $n$  es igual a 0, momento en el que la función comienza a devolver los resultados a la pila de llamadas, resolviendo así el problema original.

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

int factorial(int n) {
    if(n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}

int main() {
    int num;
    cout << "Ingresa un numero: ";
    cin >> num;
    cout << "El factorial de " << num << " es " << factorial(num) << endl;
    return 0;
}
```

En este ejemplo, la función fibonacci es una función recursiva que toma un número  $n$  como argumento y devuelve el  $n$ -ésimo término de la serie de Fibonacci. La función comprueba si  $n$  es menor o igual a 1, en cuyo caso devuelve  $n$  (ya que los dos primeros términos de la serie de Fibonacci son 0 y 1). Si  $n$  es mayor que 1, la función llama a sí misma con los argumentos  $n-1$  y  $n-2$  y devuelve la suma de ambos resultados. Esto se repite hasta que  $n$  es menor o igual a 1, momento en el que la función comienza a devolver los resultados a la pila de llamadas, resolviendo así el problema original.

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

int fibonacci(int n) {
    if (n <= 1) {
        return n;
    } else {
        return fibonacci(n-1) + fibonacci(n-2);
    }
}

int main() {
    int num;
    cout << "Ingresa un numero: ";
    cin >> num;
    cout << "El termino " << num << " de la serie de <br>Fibonacci es " << fibonacci(num);
    return 0;
}
```

# PROGRAMACIÓN ORIENTADA OBJETOS

POO: paradigma de programación que utiliza objetos, que son instancias de una clase y tienen atributos y métodos que describen comportamientos:

Cuatro conceptos principales:

- Abstracción: Simplifica y representa entidad del mundo real.
- Encapsulamiento: oculta los detalles de la clase, también usan interfaces.
- Herencia: reutilizar y extender código existente
- Polimorfismo: tratar de trabajar distintos métodos en una clase.

**Objetos:** es una instancia de la clase.

**Clases:** abstracción de elementos con características similares.

Aquí te dejo un ejemplo básico de cómo se podría implementar la POO en C++. Imaginemos que queremos modelar un objeto "persona" con nombre y edad. En este caso, podríamos crear una clase llamada Persona con los atributos nombre y edad y los métodos setNombre, setEdad, getNombre y getEdad, de la siguiente manera:

```
#include <iostream>
#include <string>
using namespace std;
class Persona {
    private:
        string nombre;
        int edad;
    public:
        void setNombre(string n) {nombre = n;}
        void setEdad(int e) {edad = e;}
        string getNombre() {return nombre;}
        int getEdad() {return edad;}
};
int main() {
    Persona p1;
    p1.setNombre("Juan");
    p1.setEdad(30);
    cout << "La persona se llama " << p1.getNombre()
    << " y tiene " << p1.getEdad() << " años." << endl;
    return 0;}
```

**En este ejemplo**, la clase Persona tiene dos atributos nombre y edad definidos como privados, lo que significa que sólo se pueden acceder a ellos desde dentro de la clase. Los métodos setNombre, setEdad, getNombre y getEdad son públicos y permiten modificar y acceder a los atributos de la clase desde fuera de ella.

En la función main, creamos un objeto p1 de la clase Persona, le asignamos un nombre y una edad utilizando los métodos setNombre y setEdad, y luego imprimimos la información de la persona utilizando los métodos getNombre y getEdad.

Este es un ejemplo muy básico de cómo se podría utilizar la POO en C++, pero en aplicaciones más complejas, la POO puede ser muy útil para organizar y estructurar el código de manera más eficiente.

## Abstracción

Simplificar y encapsular, permite crear clases y objetos con interfaces bien definidas y coherentes.

Las interfaces publicas son accesibles desde fuera de la clase y oculta los detalles de implementación interna.

En la siguiente figura, la clase Coche tiene una interfaz pública que incluye los métodos acelerar(), frenar() y mostrar\_velocidad\_actual(). Estos métodos permiten al usuario interactuar con el coche de una manera simplificada y abstracta, sin necesidad de conocer los detalles de implementación interna

```
class Coche {  
public:  
    Coche(string marca, string modelo);  
    void acelerar();  
    void frenar();  
    void mostrar_velocidad_actual();  
  
private:  
    string marca_;  
    string modelo_;  
    int velocidad_actual_;  
};
```

Ejercicios propuestos:

- 1- Una clase "Coche" que encapsula la complejidad de los detalles internos del motor, la transmisión y los sistemas eléctricos, proporcionando una interfaz pública fácil de usar para encender y apagar el motor, cambiar la marcha y controlar los faros y las luces de freno.
- 2- Una clase "Cuenta bancaria" que encapsula la complejidad de la gestión del dinero en una cuenta, proporcionando una interfaz pública fácil de usar para depositar y retirar fondos, así como para consultar el saldo actual.
- 3- Una clase "Vuelo" que encapsula la complejidad de los detalles de reserva, facturación y control de seguridad, proporcionando una interfaz pública fácil de usar para buscar y reservar vuelos, así como para consultar el estado del vuelo.
- 4- Una clase "Biblioteca" que encapsula la complejidad de la gestión de libros, proporcionando una interfaz pública fácil de usar para prestar y devolver libros, así como para buscar y reservar libros.

- 5- Una clase "Juego" que encapsula la complejidad de la lógica del juego, proporcionando una interfaz pública fácil de usar para iniciar y detener el juego, así como para gestionar la entrada del usuario y mostrar los resultados del juego.
- 6- Una clase "Empleado" que encapsula la complejidad de los detalles de la nómina y los beneficios, proporcionando una interfaz pública fácil de usar para consultar el salario, el historial laboral y las prestaciones.
- 7- Una clase "Base de datos" que encapsula la complejidad de la gestión de los datos, proporcionando una interfaz pública fácil de usar para insertar, actualizar y eliminar registros, así como para realizar consultas y generar informes.
- 8- Una clase "Red social" que encapsula la complejidad de la gestión de las conexiones y la comunicación entre usuarios, proporcionando una interfaz pública fácil de usar para buscar y agregar amigos, así como para publicar y comentar contenido.
- 9- Una clase "Robot" que encapsula la complejidad de la lógica de movimiento y control, proporcionando una interfaz pública fácil de usar para programar y controlar el movimiento del robot, así como para recibir y procesar datos de sensores.
- 10- Una clase "Juguete" que encapsula la complejidad de la fabricación y el ensamblaje de juguetes, proporcionando una interfaz pública fácil de usar para encender y apagar el juguete, así como para controlar las funciones especiales, como la música y las luces.

## Encapsulamiento

Ocultar la complejidad interna de una clase y exponer una interfaz pública bien definida para interactuar con el mundo exterior.

Capacidad de una clase de ocultar sus detalles internos de una clase (protegiéndolos de cambios no autorizados) y exponer una interfaz pública clara y bien definida para interactuar con el mundo exterior. el encapsulamiento se logra mediante el uso de modificadores de acceso como "public", "private" y "protected" para especificar la visibilidad de los miembros de la clase. Los miembros públicos son accesibles desde cualquier parte del código, mientras que los miembros privados y protegidos solo son accesibles desde dentro de la propia clase y las clases herederas, respectivamente.

Por ejemplo, la clase Coche encapsula los detalles internos del motor y la transmisión y proporciona una interfaz pública clara para interactuar con el coche. Los miembros velocidad y marcha son privados, lo que significa que solo se pueden acceder desde dentro de la propia clase, mientras que los métodos públicos como encenderMotor(), apagarMotor(), cambiarMarcha(), acelerar() y frenar() proporcionan una interfaz pública para interactuar con el coche. El programa principal utiliza estos métodos para encender el motor, cambiar la marcha, acelerar, frenar y apagar el motor. que encapsulan la complejidad interna de los detalles de implementación y proporcionan una interfaz pública más clara y fácil de usar para interactuar con el coche.

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

class Coche {
private:
    int velocidad;
    int marcha;

public:
    Coche() {
        velocidad = 0;
        marcha = 0;
    }

    void encenderMotor() {
        // código para encender el motor
        cout << "Motor encendido." << endl;
    }

    void apagarMotor() {
        // código para apagar el motor
        cout << "Motor apagado." << endl;
    }

    void cambiarMarcha(int nuevaMarcha) {
        // código para cambiar la marcha
        marcha = nuevaMarcha;
        cout << "Cambiando a la marcha " << nuevaMarcha << endl;
    }
}
```



```

void acelerar() {
    // código para acelerar
    velocidad += 10;
    cout << "Velocidad aumentada a " << velocidad << " km/h" << endl;
}

void frenar() {
    // código para frenar
    velocidad -= 10;
    cout << "Velocidad reducida a " << velocidad << " km/h" << endl;
}

};

int main() {
    Coche miCoche;

    miCoche.encenderMotor();
    miCoche.cambiarMarcha(1);
    miCoche.acelerar();
    miCoche.frenar();
    miCoche.apagarMotor();

    return 0;
}

```

En este ejemplo, la clase Empleado encapsula los detalles de un empleado, como su nombre, edad y salario. Los miembros de datos son privados, lo que significa que solo se pueden acceder desde dentro de la propia clase, y los métodos públicos getNombre(), getEdad(), getSalario() y setSalario() proporcionan una interfaz pública para interactuar con el objeto Empleado. El programa principal utiliza estos métodos para obtener y establecer los datos del empleado.

```

class Empleado {
private:
    string nombre;
    int edad;
    double salario;

public:
    Empleado(string nombre, int edad, double salario) {
        this->nombre = nombre;
        this->edad = edad;
        this->salario = salario;
    }

    string getNombre() {
        return nombre;
    }

    int getEdad() {
        return edad;
    }

    double getSalario() {
        return salario;
    }

    void setSalario(double salario) {
        this->salario = salario;
    }
};

int main() {
    Empleado empleado1("Juan Perez", 30, 1000.0);
    cout << "Nombre: " << empleado1.getNombre() << endl;
    cout << "Edad: " << empleado1.getEdad() << endl;
    cout << "Salario: " << empleado1.getSalario() << endl;

    empleado1.setSalario(1500.0);
    cout << "Nuevo salario: " << empleado1.getSalario() << endl;

    return 0;
}

```

## Herencia

Permite crear nuevas clases a partir de una clase existente. Hereda características y comportamientos.

La subclase hereda todos los atributos y métodos públicos y protegidos de la super clase (utiliza como base para crear nuevas clases)

```

#include <iostream>
#include <string>
using namespace std;
class Animal {
private:
    string nombre;
    int edad;
public:
    Animal(string n, int e):nombre(n), edad(e){}
};
class Perro:Animal{
private:
    string raza;
public:
    Perro(string n, int e, string r):Animal(n,e),raza(r){}
};
int main() {
    return 0;
}

```

## Polimorfismo

Se puede lograr a través de sobrecarga de funciones y sobre escritura de funciones virtuales. La sobrecarga de funciones permite definir múltiples funciones con el mismo nombre pero diferentes parámetros.

En la vida real nosotros podemos cumplir roles como hijos, padres y profesionales.

```

void suma(int a, int b) {
    cout << "La suma de " << a << " y " << b << " es " << a+b << endl;
}

void suma(float a, float b) {
    cout << "La suma de " << a << " y " << b << " es " << a+b << endl;
}

```

En este ejemplo, se han definido dos versiones de la función "suma" con diferentes tipos de parámetros. Dependiendo de los argumentos que se pasen a la función, el compilador seleccionará la versión adecuada.

<<Grafico>>

Sobre escritura de funciones virtuales, logra polimorfismo dinámico en tiempo de ejecución

```
#include <iostream>
using namespace std;
class Figura {
public:
    virtual float area() { return 0; }
};
class Circulo : public Figura {
private:
    float radio;
public:
    Circulo(float r) : radio(r) {}
    virtual float area() { return 3.1416 * radio * radio; }
};
class Rectangulo : public Figura {
private:
    float base, altura;
public:
    Rectangulo(float b, float h) : base(b), altura(h) {}
    virtual float area() { return base * altura; }
};
```

```
int main() {
    Figura* f;
    Circulo c(5);
    Rectangulo r(3, 4);
    f = &c;
    cout << "El área del círculo es " << f->area() << endl;
    f = &r;
    cout << "El área del rectángulo es " << f->area() << endl;
    return 0;
}
```

En este ejemplo, se han definido tres clases: "Figura", "Circulo" y "Rectangulo". La clase "Figura" tiene una función virtual "area" que se redefine en las clases "Circulo" y "Rectangulo". En la función "main", se crean objetos de las clases "Circulo" y "Rectangulo" y se asignan a un puntero de tipo "Figura". Al llamar a la función virtual "area" a través del puntero, se ejecuta la versión de la función correspondiente a la clase concreta del objeto, lo que permite calcular el área de cada figura de manera polimórfica.

Aquí hay algunas buenas prácticas para la programación orientada a objetos que pueden ayudarte a escribir un código más claro, modular y fácil de mantener:

1. Seguir el principio de responsabilidad única (SRP): cada clase debe tener una única responsabilidad o función específica.

2. Aplicar el principio de abierto/cerrado (OCP): las clases deben estar abiertas para extensión pero cerradas para modificación.
3. Utilizar la herencia con cuidado: utilizar la herencia sólo cuando sea necesario y evitar crear profundas jerarquías de herencia.
4. Preferir la composición sobre la herencia: en lugar de crear clases complejas mediante la herencia, se pueden construir objetos compuestos de varias clases más simples.
5. Mantener el acoplamiento bajo: minimizar la dependencia entre las clases, de modo que se puedan cambiar o reemplazar las clases sin afectar a otras partes del programa.
6. Utilizar interfaces y abstracciones: definir interfaces claras para las clases y programar en términos de esas interfaces, en lugar de depender directamente de las implementaciones concretas de las clases.
7. Utilizar nombres descriptivos para las clases y métodos: elegir nombres significativos que reflejen la función de la clase o del método.
8. Comentar el código: escribir comentarios claros y concisos que expliquen lo que hace el código, por qué se hizo así y cómo se utiliza.
9. Aplicar pruebas unitarias: probar cada clase o método individualmente para asegurarse de que funcionan correctamente.
10. Mantener el código limpio y organizado: escribir código legible y fácil de entender, utilizando un estilo de codificación consistente y siguiendo las convenciones de nomenclatura y formato.

## **Funciones Virtuales**

## STACK(PILA)

- Representa una pila de elementos.
- principio LIFO (Last In, First Out)
- Utilizar la librería `#include<stack>`

Stack <tipo\_dato> miPila;

### Métodos:

- `push()`: Inserta un elemento en la cima del Stack.
- `pop()`: Elimina el elemento superior (último elemento insertado) del Stack.
- `top()`: Devuelve una referencia al elemento superior (último elemento insertado) del Stack.
- `empty()`: Devuelve verdadero si el Stack está vacío, falso de lo contrario.
- `size()`: Devuelve la cantidad de elementos que hay en el Stack.

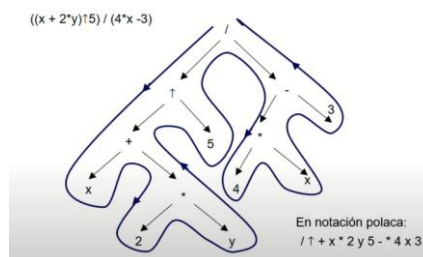
### Ejemplos:

- 1- Uso en procesamiento de expresiones matemáticas: Al procesar expresiones matemáticas como  $2 * (3 + 4) / 5$ , se puede utilizar una pila para almacenar los operandos y operadores en el orden correcto para su procesamiento posterior.
- 2- Control de historial de navegación en un navegador web: Para registrar el historial de navegación en un navegador web, se puede utilizar una pila para almacenar las URL de las páginas web visitadas. De esta forma, cuando el usuario desea volver a la página anterior, se puede utilizar la pila para recuperar la URL correspondiente.
- 3- Gestión de llamadas en un sistema operativo: En un sistema operativo, se puede utilizar una pila para manejar la gestión de llamadas. Cuando se hace una llamada, la información de la llamada se almacena en la pila. A medida que se completan las llamadas, la información de la llamada se elimina de la pila en orden inverso.
- 4- Evaluación de condiciones en juegos de mesa: En juegos de mesa como el ajedrez, se puede utilizar una pila para almacenar las condiciones de las jugadas anteriores. Esto permite a los jugadores deshacer las jugadas anteriores si es necesario.

Reversión de tareas en un software de diseño gráfico: En un software de diseño gráfico, una pila se puede utilizar para mantener un registro de las tareas realizadas por el usuario. Si el usuario desea deshacer una tarea, la pila se utiliza para revertir la tarea en orden inverso.

### Ejercicio:

- 1- Evaluación de expresiones aritméticas en notación polaca inversa utilizando una pila



- 2- Implementación de un sistema de historial de navegación en un navegador web utilizando una pila para almacenar las páginas visitadas.

- 3- Uso de una pila para implementar la funcionalidad de deshacer (undo) y rehacer (redo) en un editor de texto.
- 4- Implementación de una función que verifica si una cadena de paréntesis (por ejemplo, "((()))") está balanceada utilizando una pila.
- 5- Implementación de un algoritmo de búsqueda en profundidad (depth-first search) en un grafo utilizando una pila para almacenar los nodos a visitar.

## CONSTRUCTORES & DESTRUCTORES

Utilizadas para inicializar y liberar recursos de los objetos, respectivamente.

Ejemplo de un constructor para la clase "Persona" que toma dos argumentos (el nombre y la edad)

```
class Persona {
private:
    string nombre;
    int edad;
public:
    Persona(string n, int e) : nombre(n), edad(e) {}
    //...
};
```

O también:

```
class Persona {
private:
    string nombre;
    int edad;
    string direccion;
public:
    Persona() : nombre(""), edad(0), direccion("") {}
    Persona(string nombre, int edad, string direccion) :
        nombre(nombre), edad(edad), direccion(direccion) {}
};
```

Los destructores se llaman automáticamente cuando se destruye un objeto de una clase y se utiliza para liberar los recursos que se han asignado al objeto

```
class Persona {
private:
    string nombre;
    int edad;
public:
    Persona(string n, int e) : nombre(n), edad(e) {}
    ~Persona() {}
    //...
};
```

Las ventajas de utilizar constructores y destructores son:



- Inicialización segura de datos miembro: Los constructores aseguran que los datos miembro de un objeto se inicialicen correctamente antes de que se utilicen. Esto ayuda a prevenir errores y garantiza que el objeto esté en un estado válido desde el principio.
- Liberación automática de recursos: Los destructores aseguran que los recursos asignados a un objeto se liberen automáticamente cuando el objeto se destruye. Esto evita fugas de memoria y otros problemas relacionados con la gestión manual de la memoria.

Las desventajas de utilizar constructores y destructores son:

- Sobrecarga de procesamiento: Los constructores y destructores pueden añadir cierta sobrecarga de procesamiento a la creación y destrucción de objetos. En algunos casos, esta sobrecarga puede ser significativa y afectar al rendimiento del programa.
- Dificultad de depuración: Los constructores y destructores pueden introducir errores difíciles de depurar si no se utilizan correctamente. Es importante comprender cómo funcionan y cómo deben ser implementados adecuadamente.

En resumen, los constructores y destructores son funciones importantes en la programación orientada a objetos que permiten inicializar y liberar recursos de objetos de manera segura y eficiente.

## SISTEMAS DE PROTECCIÓN

El sistema de protección en C++ se utiliza para implementar los conceptos de encapsulamiento y herencia. Por ejemplo, los miembros privados de una clase se pueden utilizar para ocultar la información de la implementación interna de la clase y evitar que se acceda a ella desde fuera de la clase. Los miembros protegidos se pueden utilizar para permitir que las clases derivadas accedan a la información interna de la clase base.

Aquí hay un ejemplo que muestra el uso de los distintos sistemas de protección en C++:

```
class Animal {
public:
    void comer() {
        cout << "El animal está comiendo" << endl;
    }

protected:
    void dormir() {
        cout << "El animal está durmiendo" << endl;
    }

private:
    void mover() {
        cout << "El animal está moviéndose" << endl;
    }
};
```

```
class Perro : public Animal {
public:
    void ladrar() {
        cout << "El perro está ladrando" << endl;
    }
};
```

```
int main() {  
    Animal animal;  
    Perro perro;  
  
    animal.comer(); // Acceso público permitido  
    //animal.dormir(); // Acceso protegido no permitido desde fuera de la clase  
    //animal.mover(); // Acceso privado no permitido desde fuera de la clase  
  
    perro.comer(); // Acceso público heredado permitido  
    //perro.dormir(); // Acceso protegido no permitido desde fuera de la clase  
    //perro.mover(); // Acceso privado no permitido desde fuera de la clase  
  
    perro.ladRAR(); // Acceso público permitido desde la clase derivada  
    return 0;  
}
```

En este ejemplo, la clase Animal tiene tres miembros: comer(), dormir() y mover(). comer() es un miembro público, dormir() es un miembro protegido y mover() es un miembro privado. La clase Perro es una clase derivada de Animal y hereda comer() como miembro público.

En resumen, el sistema de protección en C++ es fundamental para la implementación del encapsulamiento y la herencia en la programación orientada a objetos. Permite controlar el acceso a los miembros de una clase y garantizar la integridad de los datos.

# FUNCIONES Y CLASES AMIGAS

## Funciones Amigas

es una función que tiene acceso a los miembros privados y protegidos de una clase. Para declarar una función como amiga de una clase, se utiliza la palabra clave friend en la declaración de la función dentro de la clase.

Las funciones amigas son útiles en situaciones donde se necesita acceder a los miembros privados de una clase desde una función que no es miembro de la clase.

Una de las ventajas de las funciones amigas es que pueden acceder a los miembros privados y protegidos de una clase, lo que permite trabajar con los datos de la clase sin tener que hacer que los miembros sean públicos o proporcionar métodos de acceso. Sin embargo, el uso excesivo de funciones amigas puede violar el principio de encapsulamiento, lo que puede dificultar la comprensión y el mantenimiento del código.

### 1- Suma de dos objetos de la clase "Fraccion"

```
1  class Fraccion {
2      private:
3          int numerador;
4          int denominador;
5      public:
6          Fraccion(int n, int d) {}
7          numerador = n;
8          denominador = d;
9      }
10     friend Fraccion suma_fracciones(const Fraccion& f1, const Fraccion& f2) {
11         int n = f1.numerador * f2.denominador + f2.numerador * f1.denominador;
12         int d = f1.denominador * f2.denominador;
13         return Fraccion(n, d);
14     }
15 };
16
17 int main() {
18     Fraccion f1(1, 2);
19     Fraccion f2(3, 4);
20     Fraccion resultado = suma_fracciones(f1, f2);
21     return 0;
22 }
```

### 2- Función para imprimir los valores de los miembros privados de la clase Persona

```

class Persona {
    private:
        string nombre;
        int edad;
    public:
        Persona(string n, int e) {
            nombre = n;
            edad = e;
        }
        friend void imprimir_datos(const Persona& p) {
            cout << "Nombre: " << p.nombre << endl;
            cout << "Edad: " << p.edad << endl;
        }
};

int main() {
    Persona p("Juan", 25);
    imprimir_datos(p);
    return 0;
}

```

- 3- Función para comparar dos objetos de la clase Rectangulo y determinar cuál tiene un área mayor:

```

1  class Rectangulo {
2      private:
3          int base;
4          int altura;
5      public:
6          Rectangulo(int b, int a) {
7              base = b;
8              altura = a;
9          }
10         friend Rectangulo comparar_rectangulos(const Rectangulo& r1, const Rectangulo& r2) {
11             int area1 = r1.base * r1.altura;
12             int area2 = r2.base * r2.altura;
13             return (area1 > area2) ? r1 : r2;
14         }
15     };
16
17     int main() {
18         Rectangulo r1(5, 10);
19         Rectangulo r2(8, 4);
20         Rectangulo mayor = comparar_rectangulos(r1, r2);
21         return 0;
22     }

```

Función para calcular la distancia entre dos puntos en un espacio tridimensional utilizando la clase Punto:

```

1  class Punto {
2      private:
3          int x;
4          int y;
5          int z;
6      public:
7          Punto(int x, int y, int z) {
8              this->x = x;
9              this->y = y;
10             this->z = z;
11         }
12         friend float distancia_entre_puntos(const Punto& p1, const Punto& p2) {
13             int dx = p1.x - p2.x;
14             int dy = p1.y - p2.y;
15             int dz = p1.z - p2.z;
16             return sqrt(dx*dx + dy*dy + dz*dz);
17         }
18     };
19     int main() {
20         Punto p1(1, 2, 3);
21         Punto p2(4, 5, 6);
22         float distancia = distancia_entre_puntos(p1, p2);
23         return 0;
24     }

```

Función para modificar atributos privados de una clase Personaje

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4  class Personaje {
5  public:
6      friend void modificar(Personaje &,int , int );
7      Personaje():ataque(0),defensa(0){}
8      Personaje(int a, int b) : ataque(a), defensa(b){}
9      void mostrar(){
10         cout<<"Ataque: "<<ataque<<endl;
11         cout<<"defensa: "<<defensa<<endl;
12     }
13
14     private:
15         int ataque;
16         int defensa;
17 };
18
19 void modificar(Personaje &p,int a, int b){
20     p.ataque=a;
21     p.defensa=b;
22 }
23
24 int main() {
25     Personaje p(5,3);
26     modificar(p,10,15);
27     p.mostrar();
28     return 0;
29 }
30

```

Veamos el último ejemplo:

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

class MySecretClass {
private:
    int secretNum;
public:
    MySecretClass() {
        secretNum = 1234;
    }
    friend void printSecretNum(MySecretClass obj);
};

void printSecretNum(MySecretClass obj) {
    cout << "The secret number is: " << obj.secretNum << endl;
}

int main() {
    MySecretClass obj;
    printSecretNum(obj);
    return 0;
}
```

Ventajas de funciones amigas:

- Permiten el acceso a miembros privados de una clase desde funciones externas, lo que puede simplificar el código y hacerlo más legible.
- Aceleran la ejecución del programa, ya que evitan el uso de métodos públicos de acceso a los datos.
- Pueden utilizarse para restringir el acceso a datos privados, permitiendo que solo ciertas funciones tengan acceso a ellos.

Desventajas de funciones amigas:

- Las funciones amigas rompen el encapsulamiento de la clase, lo que puede hacer que el código sea menos seguro y más difícil de mantener.
- Si se utilizan mal, pueden permitir que se modifiquen los datos de una clase de manera inesperada, lo que puede provocar errores difíciles de detectar.
- Pueden dificultar la reutilización del código, ya que las funciones amigas están diseñadas específicamente para trabajar con una clase en particular y pueden no funcionar correctamente con otras clases.

Debemos tener en cuenta que las funciones amigas deben utilizarse con precaución en la construcción de un algoritmo, ya que rompen el encapsulamiento y pueden introducir problemas de seguridad y mantenimiento.

## Funciones Externas Amigas

Son funciones que se declaran fuera de la clase, pero tienen acceso a los miembros privados y protegidos de la clase. Aquí hay algunos ejemplos de funciones externas amigas en C++.

Ejemplo 1, Función para imprimir los valores de los miembros privados de una clase:

```
class MiClase {
private:
    int valor1;
    int valor2;
    friend void imprimirValores(MiClase obj);
};

void imprimirValores(MiClase obj) {
    cout << "Valor1: " << obj.valor1 << endl;
    cout << "Valor2: " << obj.valor2 << endl;
}
```

Ejemplo2, Función para calcular la media de los valores de los miembros privados de una clase:

```
class MiClase {
private:
    int valor1;
    int valor2;
    friend double mediaValores(MiClase obj);
};

double mediaValores(MiClase obj) {
    return (obj.valor1 + obj.valor2) / 2.0;
}
```

## Funciones amigas en otras clases.

Las funciones amigas en otras clases en C++ son funciones que tienen acceso a los miembros privados y protegidos de una clase diferente a la que están declaradas. Aquí hay algunos ejemplos de funciones amigas en otras clases en C++.

Ejemplo1, Función amiga en otra clase para calcular la suma de dos objetos de una clase:



```

class MiClase {
private:
    int valor1;
    int valor2;
    friend class OtraClase;
};

class OtraClase {
public:
    MiClase sumarObjetos(MiClase obj1, MiClase obj2) {
        MiClase resultado;
        resultado.valor1 = obj1.valor1 + obj2.valor1;
        resultado.valor2 = obj1.valor2 + obj2.valor2;
        return resultado;
    }
};

```

Ejemplo 2, Función amiga en otra clase para imprimir los valores de los miembros privados de una clase:

```

class MiClase {
private:
    int valor1;
    int valor2;
    friend class OtraClase;
};

class OtraClase {
public:
    void imprimirValores(MiClase obj) {
        cout << "Valor1: " << obj.valor1 << endl;
        cout << "Valor2: " << obj.valor2 << endl;
    }
};

```

## Clases amigas

En C++, las clases amigas son clases que tienen acceso a los miembros privados y protegidos de otra clase. Una clase amiga se declara dentro de la clase que se va a beneficiar de su amistad y puede acceder a sus miembros privados y protegidos como si fueran miembros de su propia clase

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4  class Amigo {
5  public:
6      void funcionAmiga(Clase amiga); // Función amiga
7  };
8
9  class Clase {
10 private:
11     int datoPrivado = 42;
12     friend class Amigo; // Clase amiga
13 };
14
15 void Amigo::funcionAmiga(Clase amiga) {
16     cout << "El dato privado de Clase es: " << amiga.datoPrivado << endl;
17 }
18
19 int main() {
20     Clase c;
21     Amigo a;
22     a.funcionAmiga(c); // Imprime: El dato privado de Clase es: 42
23     return 0;
24 }

```

Las clases amigas pueden ser útiles en situaciones en las que se necesita acceder a los miembros privados o protegidos de una clase desde otra clase que no es su amiga, pero que debe interactuar con ella de alguna manera. Sin embargo, el uso excesivo de clases amigas puede violar los principios de encapsulamiento y puede hacer que el código sea más difícil de mantener y modificar en el futuro. Por lo tanto, es importante usar las clases amigas con precaución y sólo en situaciones en las que realmente sean necesarias.

- 1- Ejemplo: Supongamos que tenemos una clase CuentaBancaria que tiene información privada de una cuenta, como el saldo actual y el número de cuenta. Si queremos crear una clase Banco que tenga acceso a esta información para realizar operaciones como depósitos y retiros, podemos hacer que la clase Banco sea amiga de la clase CuentaBancaria

```

class CuentaBancaria {
private:
    double saldo;
    int numCuenta;
    friend class Banco;
public:
    // constructor, métodos, etc.
};

class Banco {
public:
    void depositar(CuentaBancaria &cuenta, double monto) {
        cuenta.saldo += monto;
    }
    void retirar(CuentaBancaria &cuenta, double monto) {
        cuenta.saldo -= monto;
    }
};

```

## MODIFICADORES PARA MIEMBROS

### Funciones inline

permite al compilador optimizar el código al expandir el cuerpo de la función directamente en el lugar donde se llama en lugar de generar una llamada a función.

```

#include <iostream>

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

int main() {
    int x = 5, y = 7;
    std::cout << "Sum: " << sum(x, y) << std::endl;
    return 0;
}

```

Ventajas:

- Mejora el rendimiento del programa, ya que elimina el costo de la llamada a la función.
- Reduce el tiempo de ejecución, ya que no hay necesidad de saltar a otra parte del programa para ejecutar la función.
- Es especialmente útil para funciones pequeñas y simples, como funciones de acceso a datos o de utilidad.

- Se pueden definir en un archivo de cabecera y, por lo tanto, se pueden incluir en múltiples archivos de origen.

Desventajas:

- El tamaño del código binario puede aumentar, ya que el compilador debe insertar el código de la función en el lugar donde se llama.
- Si se cambia el cuerpo de la función, se deben volver a compilar todos los archivos que incluyen la definición de la función.
- Las funciones inline no se pueden depurar fácilmente, ya que no tienen una dirección de entrada única.

Es importante tener en cuenta que la decisión de utilizar una función inline debe basarse en su frecuencia de uso y en la necesidad de optimizar el rendimiento del programa.

Ejemplo 1:

```
inline double area_circulo(double radio) {  
    return 3.14159 * radio * radio;  
}
```

Ejemplo 2:

```
inline double promedio(int arr[], int n) {  
    int suma = 0;  
    for (int i = 0; i < n; i++) {  
        suma += arr[i];  
    }  
    return suma / n;  
}
```

Ejemplo 3:

```
inline int potencia(int base, int exponente) {  
    int resultado = 1;  
    for (int i = 0; i < exponente; i++) {  
        resultado *= base;  
    }  
    return resultado;  
}
```

Ejemplo 4:

```
inline int factorial(int n) {  
    int resultado = 1;  
    for (int i = 1; i <= n; i++) {  
        resultado *= i;  
    }  
    return resultado;  
}
```

Ejemplo 5,

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

class Estudiante {
public:
    Estudiante (int i,string n) {codigo = i; nombre = n;};/*
    implícitamente (inline)*/
    string getNombre();
    int getCodigo();
private:
    string nombre;
    int codigo;
};
inline string Estudiante::getNombre(){/* explícitamente (inline)*/
    return nombre;
}
inline int Estudiante::getCodigo(){
    return codigo;
}
int main(){
    Estudiante st(11,"jose");
    cout <<"codigo: "<<st.getCodigo()<<endl;
    cout<<"nombre: "<<st.getNombre()<<endl;
    return 0;
}
```

## Niveles De Acceso

Estos modificadores son public, private y protected.

- public: son accesibles desde cualquier parte del programa, es decir, cualquier objeto puede acceder a ellos directamente.
- private: son accesibles desde dentro de la clase en la que se declararon. Esto significa que los objetos de otras clases no pueden acceder a ellos directamente.
- protected: son similares a los miembros privados, pero tienen la diferencia de que son accesibles por las clases derivadas.

```

class Persona {
    private:
        int edad; // Atributo privado

    public:
        string nombre; // Atributo público

    protected:
        string direccion; // Atributo protegido

    public:
        void set_edad(int e) {
            edad = e;
        }

        int get_edad() {
            return edad;
        }
};

```

## const

Una variable o función no debe ser modificada. Cuando se declara una variable miembro de una clase como const, esto significa que su valor no puede cambiar después de su inicialización en el constructor de la clase.

**Funciones constantes.**- es una función que promete no modificar el objeto al que se llama. Es decir, una función constante no puede modificar los miembros de datos de la clase.

```

class Ejemplo {
public:
    int getX() const {
        return x;
    }
private:
    int x;
};

```

En este ejemplo, getX() es una función constante que devuelve el valor del miembro x de la clase Ejemplo. La palabra clave const después de la lista de argumentos indica que la función no modificará el objeto al que se llama.

**Clases constantes.** - es una clase cuyos miembros de datos no se pueden modificar después de su inicialización.

```
class Ejemplo {  
public:  
    Ejemplo(int x) : x(x) {}  
    int getX() const {  
        return x;  
    }  
private:  
    const int x;  
};
```

## COMPONENTES ESTÁTICOS

### Static

En C++, los componentes estáticos se pueden definir de varias formas:

- **Variables estáticas:** son variables que se declaran dentro de una función o clase, pero su ámbito de vida se extiende a toda la duración del programa. Se inicializan una sola vez al inicio de la ejecución y no pueden ser modificadas después.
- **Funciones estáticas:** son funciones que se declaran dentro de una clase y se pueden invocar sin necesidad de crear un objeto de la clase. Tienen un alcance limitado a la clase donde se declaran y no pueden ser modificadas en tiempo de ejecución.
- **Métodos estáticos:** son métodos que se declaran dentro de una clase y se pueden invocar sin necesidad de crear un objeto de la clase. Al igual que las funciones estáticas, su alcance se limita a la clase donde se declaran y no pueden ser modificados en tiempo de ejecución.

Una variable static tiene el mismo valor para todas las instancias de la clase y se puede acceder a ella utilizando el nombre de la clase en lugar de una instancia específica. Una función static también pertenece a la clase en lugar de a una instancia y se puede llamar usando el nombre de la clase.

Veamos un ejemplo: En la función main(), se crean tres instancias de MyClass. Cada instancia tiene un valor de id diferente, pero todas comparten el mismo valor de count, ya que es un miembro de clase static. Luego se llama a la función printInfo() en cada instancia para imprimir su id y el valor actual de count.

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

class MyClass {
public:
    static int count; // miembro de clase
    const int id; // miembro de instancia constante

    MyClass(int id) : id(id) { // constructor
        count++; // incrementar el contador de instancias
    }

    void printInfo() const { // función miembro constante
        cout << "ID: " << id << ", Count: " << count << endl;
    }
};

int MyClass::count = 0; // inicialización del miembro de clase

int main() {
    MyClass obj1(1);
    MyClass obj2(2);
    MyClass obj3(3);

    obj1.printInfo(); // ID: 1, Count: 3
    obj2.printInfo(); // ID: 2, Count: 3
    obj3.printInfo(); // ID: 3, Count: 3

    return 0;
}
```



## **PUNTEROS**

## **OBJETOS DENTRO DE OBJETOS C++**

# JERARQUÍA DE CLASES

## Herencia

Tipos:

- Extensión
- Agregación

3 niveles de acceso:

- Public
- Private
- Protected

## Herencia abstracta

Una clase abstracta se define utilizando al menos una función miembro pura (función virtual pura), lo que hace que la clase sea abstracta y no se puedan crear objetos de ella.

Para definir una clase abstracta en C++, se utiliza el modificador virtual y se asigna la función miembro pura utilizando la sintaxis `virtual tipo_retorno nombre_funcion() = 0;`. La función miembro pura no tiene implementación en la clase abstracta y debe ser implementada en las clases derivadas.

Aquí tienes un ejemplo de una clase abstracta "Figura" que tiene una función miembro pura "calcularArea()" y dos clases derivadas "Rectangulo" y "Triangulo" que implementan esa función:

```
#include <iostream>

class Figura {
public:
    virtual double calcularArea() = 0;
};

class Rectangulo : public Figura {
private:
    double base, altura;
public:
    Rectangulo(double b, double a) : base(b), altura(a) {}

    double calcularArea() {
        return base * altura;
    }
};
```

```

class Triangulo : public Figura {
private:
    double base, altura;
public:
    Triangulo(double b, double a) : base(b), altura(a) {}

    double calcularArea() {
        return (base * altura) / 2;
    }
};

int main() {
    Rectangulo rectangulo(5, 3);
    Triangulo triangulo(4, 6);

    Figura* figura1 = &rectangulo;
    Figura* figura2 = &triangulo;

    std::cout << "Área del rectángulo: " << figura1->calcularArea() << std::endl;
    std::cout << "Área del triángulo: " << figura2->calcularArea() << std::endl;

    return 0;
}

```

En este ejemplo, la clase "Figura" es una clase abstracta con la función miembro pura "calcularArea()". Las clases "Rectangulo" y "Triangulo" son clases derivadas de "Figura" y proporcionan una implementación de "calcularArea()". En la función principal, se crean objetos de las clases derivadas y se utilizan punteros de tipo "Figura" para llamar a la función "calcularArea()" de cada objeto. Esto es posible gracias al polimorfismo, ya que el tipo estático del puntero es "Figura" pero el tipo dinámico es el de la clase derivada correspondiente.

## Compatibilidad de tipos

- <Dynamic\_cast>
- <static\_cast>

## Objetos como parámetros

Clases de almacenamiento:

- Variable local
- Variable Global

## Parámetros de una función

<funciones amigas>

### **Pasar un objeto por valor**

- Los cambios en el objeto solo afectan a la copia local de la función

### **Pasar un objeto de una subclase**

## HERENCIA MÚLTIPLE

Permite que una clase herede de varias clases a la vez. En C++, una clase puede heredar de múltiples clases mediante la declaración de la herencia separada por comas. Por ejemplo:

```
class CelularElectronico : public Electronico, public Celular {  
    // definición de la clase  
};
```

la clase "CelularElectronico" heredará todas las funciones y variables públicas de las clases "Electronico" y "Celular".

**IMPORTANTE:** la herencia múltiple puede aumentar la complejidad y la dificultad de mantenimiento del código, por lo que es importante utilizarla con precaución y solo cuando sea necesaria.

hay otros lenguajes de programación que admiten la herencia múltiple. Algunos ejemplos son:

- Python: Python admite la herencia múltiple. En Python, una clase puede heredar atributos y métodos de varias clases base.
- C#: C# admite la herencia múltiple mediante el uso de interfaces. Al igual que en Java, una clase puede implementar múltiples interfaces para heredar comportamiento de varias clases.
- Ruby: Ruby admite la herencia múltiple de manera similar a C++. Una clase puede heredar atributos y métodos de varias clases base.

## ERRAR NO ES DE HUMANOS

- Leyes de Murphy Fatal:
  - o No hay código sin errores, existe código libre de errores ha sido suficientemente probado
  - o Ley de la persistencia de los problemas: Los Problemas ni se crean, ni se destruyen, sólo se transforman

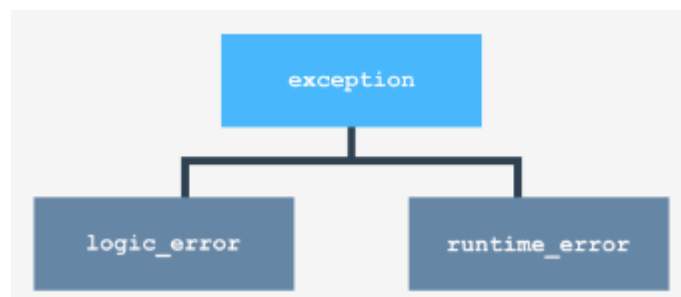
¿Como salir de apuros?

Existen diferentes formas de salir de apuros cuando ocurren errores en C++, entre ellas:

- Depuración: Se trata de la técnica más común para encontrar y solucionar errores en el código. Consiste en utilizar herramientas específicas, como un depurador, para identificar la fuente del error y corregirlo directamente en el código.
- Excepciones: Las excepciones permiten al programa detectar y responder a errores de manera más elegante y estructurada. En lugar de simplemente detener la ejecución del programa, las excepciones pueden capturarse y procesarse para intentar resolver el problema o proporcionar una salida adecuada.
- Logging: El registro de eventos o logging es una técnica que consiste en registrar los eventos del programa en un archivo o base de datos. Esto puede ayudar a los desarrolladores a entender mejor cómo se está ejecutando el programa, identificar errores y tomar medidas para solucionarlos.
- Pruebas: Las pruebas automatizadas pueden ayudar a encontrar errores antes de que el programa se lance en producción. Esto puede ahorrar tiempo y recursos al detectar errores temprano en el proceso de desarrollo y corregirlos antes de que afecten a los usuarios finales.
- Gestión de errores: Es importante diseñar el código de manera que los errores puedan ser gestionados adecuadamente. Esto puede incluir la verificación de entradas de usuario, el uso de valores por defecto y la manipulación cuidadosa de los recursos para minimizar los errores.

Hablemos de excepciones ahora:

El lenguaje C++ proporciona algunas clases especializadas reunidas en una estructura jerárquica que refleja las diferentes naturalezas de las diferentes excepciones



Dos clases diferentes se derivan de la clase de excepción. El primero, llamado `logic_error`, está destinado a representar excepciones conectadas a la lógica del programa, es decir, el algoritmo, su implementación, validez de datos y cohesión. Podemos decir que este tipo de excepción se lanza a niveles más altos de abstracción de programas.

La segunda clase, llamada `runtime_error`, se utiliza para identificar excepciones lanzadas debido a accidentes "inesperados" como la falta de memoria. Usamos la palabra `runtime_error` porque se convierte en un oxímoron en este contexto: no podemos decir que un evento es inesperado cuando nos estamos preparando para servirlo.

En C++, se manejan las excepciones mediante la sintaxis "try-catch". Un bloque "try" es utilizado para encerrar el código que puede lanzar una excepción, mientras que uno o más bloques "catch" son utilizados para manejar las excepciones que pueden ser lanzadas por ese código.

Un ejemplo sencillo de uso de excepciones en C++ es el siguiente:

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

int main() {
    try {
        int num1, num2;
        cout << "Introduzca dos números enteros: ";
        cin >> num1 >> num2;
        if (num2 == 0) {
            throw "División por cero";
        }
        int resultado = num1 / num2;
        cout << "El resultado de la división es: " << resultado << endl;
    }
    catch (const char* mensaje) {
        cout << "Error: " << mensaje << endl;
    }
    return 0;
}
```

En este ejemplo, el bloque "try" encierra el código que pide al usuario que introduzca dos números enteros, y que realiza la división de esos números. Si el segundo número es cero, se lanza una excepción con un mensaje de error ("División por cero"). El bloque "catch" es utilizado para capturar esa excepción y mostrar el mensaje de error correspondiente.

En C++, hay dos tipos de excepciones: las excepciones predefinidas y las excepciones definidas por el usuario.

- **Excepciones predefinidas:** son las excepciones integradas en el lenguaje C++. Algunos ejemplos son:

`std::bad_alloc`: se lanza cuando no se puede asignar memoria dinámicamente.

`std::out_of_range`: se lanza cuando se intenta acceder a un elemento fuera del rango válido de una estructura de datos, como un vector o una matriz.

`std::logic_error`: se lanza cuando se produce un error en la lógica del programa, como una división por cero o una operación inválida.



- **Excepciones definidas por el usuario:** son las excepciones que se crean en el propio programa de C++. El usuario puede definir una clase de excepción personalizada para manejar errores específicos que puedan ocurrir en su programa.

Por ejemplo, si se está desarrollando un programa para una tienda de electrónica, se puede crear una clase de excepción personalizada llamada "ProductoNoDisponible" que se lanza cuando un cliente intenta comprar un producto que está fuera de stock. El código para la excepción podría ser algo así:

```
class ProductoNoDisponible : public std::exception {
public:
    virtual const char* what() const throw() {
        return "Producto no disponible";
    }
};
```

Este código define una clase llamada ProductoNoDisponible que hereda de la clase std::exception. También incluye una función miembro llamada "what" que devuelve una cadena de texto que describe el error.

Algunas posibles desventajas del uso de excepciones en C++ son:

- **Mayor complejidad del código:** el uso de excepciones puede aumentar la complejidad del código, especialmente si se manejan múltiples tipos de excepciones y se implementa un manejo adecuado de errores.
- **Sobrecarga de procesamiento:** si se lanzan muchas excepciones en un programa, puede haber un impacto significativo en el rendimiento, ya que la ejecución de excepciones puede requerir una gran cantidad de procesamiento adicional.
- **Interrupción del flujo de control:** el uso de excepciones puede interrumpir el flujo normal de ejecución del programa y puede hacer que el código sea más difícil de seguir y depurar.
- **Posibilidad de errores lógicos:** el manejo inadecuado de excepciones puede llevar a errores lógicos en el programa, lo que puede ser difícil de detectar y solucionar.

En general, aunque las excepciones pueden ser útiles para manejar errores y simplificar el código en algunas situaciones, también pueden agregar complejidad y sobrecarga de procesamiento, por lo que es importante usarlas con cuidado y en situaciones apropiadas.

## Principales clases de errores

Clase "domain\_error"

Clase "invalid\_argument"

Clase "length\_error"

Clase "out\_of\_range"

Clase "runtime\_error"

Clase "length\_error"

Clase "range\_error"

## Stack

## Plantillas

Las plantillas se definen utilizando la palabra clave "template" seguida de los parámetros de plantilla. Los parámetros de plantilla pueden ser tipos o valores constantes. Por ejemplo:

```
template <typename T>
class MiClase {
public:
    T obtenerDato() const;
    void establecerDato(const T& dato);
};

template <typename T>
T MiClase<T>::obtenerDato() const {
    // Implementación de la función
}

template <typename T>
void MiClase<T>::establecerDato(const T& dato) {
    // Implementación de la función
}
```

En el ejemplo anterior, se define una clase llamada "MiClase" utilizando una plantilla. El parámetro de plantilla "T" se utiliza para representar un tipo genérico. Los métodos de la clase, como "obtenerDato" y "establecerDato", se implementan dentro y fuera de la declaración de la clase utilizando la sintaxis de plantilla.

El uso de la plantilla se realiza cuando se instancia la clase con un tipo específico:

```
MiClase<int> objetoEntero; // Instancia de la clase para el tipo int
objetoEntero.establecerDato(10);
int dato = objetoEntero.obtenerDato();
```

Las plantillas también se pueden utilizar para definir funciones genéricas. Por ejemplo:

```
template <typename T>
T suma(const T& a, const T& b) {
    return a + b;
}

int resultadoEntero = suma(2, 3);           // suma para enteros
float resultadoFloat = suma(2.5, 3.7);     // suma para flotantes
```

En este ejemplo, la función "suma" utiliza una plantilla para permitir la suma de diferentes tipos de datos. La función se puede llamar con enteros, flotantes u otros tipos compatibles.

Ventajas de las plantillas en C++:

1. **Reutilización de código:** Las plantillas permiten escribir código genérico que puede ser utilizado con diferentes tipos de datos. Esto facilita la reutilización del código y evita la necesidad de escribir implementaciones separadas para cada tipo de dato.
2. **Flexibilidad:** Las plantillas ofrecen flexibilidad al permitir la creación de estructuras de datos y algoritmos genéricos. Puedes crear contenedores, funciones y clases que se adapten a diferentes tipos de datos sin tener que repetir el código.
3. **Eficiencia:** Las plantillas en C++ se resuelven en tiempo de compilación, lo que significa que el código generado para cada tipo de dato es específico y no se incurre en la sobrecarga de tiempo de ejecución asociada a la programación orientada a objetos.
4. **Abstracción:** Las plantillas permiten la abstracción de conceptos comunes y la encapsulación de la lógica en un solo lugar. Esto facilita la comprensión y el mantenimiento del código.

#### Desventajas de las plantillas en C++:

1. **Código más complejo:** El uso de plantillas puede aumentar la complejidad del código debido a la necesidad de lidiar con la sintaxis y las reglas específicas de las plantillas. Esto puede dificultar la comprensión y el mantenimiento del código.
2. **Tiempo de compilación más largo:** El uso de plantillas puede aumentar el tiempo de compilación, especialmente cuando se utilizan tipos de datos complejos o se instancian múltiples tipos de plantillas.
3. **Mensajes de error confusos:** Los mensajes de error relacionados con plantillas pueden ser complicados y difíciles de interpretar. La naturaleza genérica de las plantillas puede hacer que los errores sean difíciles de rastrear y corregir.
4. **En algunos casos, el compilador puede generar múltiples copias de código para cada tipo de dato utilizado en una plantilla.** Esto puede aumentar el tamaño del archivo ejecutable y afectar el rendimiento del programa.

#### Recomendaciones:

Aquí tienes algunos consejos y buenas prácticas para utilizar plantillas en C++ de manera efectiva:

- **Mantén las plantillas simples:** Intenta mantener las plantillas lo más simples y genéricas posible. Evita agregar lógica compleja o dependencias específicas del dominio dentro de las plantillas. Esto facilitará la comprensión y el mantenimiento del código.
- **Nombra las plantillas de manera descriptiva:** Utiliza nombres descriptivos para tus plantillas y parámetros de plantilla. Esto ayudará a otros desarrolladores a comprender el propósito y el uso de la plantilla.
- **Proporciona documentación clara:** Documenta adecuadamente tus plantillas, incluyendo información sobre los requisitos de los parámetros de plantilla, el comportamiento esperado y cualquier restricción o limitación. Esto facilitará su uso por parte de otros desarrolladores.
- **Valida los parámetros de plantilla:** Si es necesario, realiza validaciones en tiempo de compilación para garantizar que los tipos de datos utilizados con las plantillas cumplan con los requisitos esperados. Esto puede ayudar a detectar errores y garantizar un comportamiento predecible.

- Evita la duplicación innecesaria de código: Si encuentras que estás escribiendo múltiples plantillas con lógica similar, considera la posibilidad de factorizar ese código en una plantilla común o utilizar técnicas de herencia o composición para compartir la implementación.
- Prueba exhaustivamente las plantillas: Realiza pruebas exhaustivas de tus plantillas con diferentes tipos de datos para asegurarte de que funcionen correctamente en todos los casos. Ten en cuenta los casos límite y los tipos de datos inesperados.
- Utiliza especialización de plantillas de manera adecuada: Si necesitas un comportamiento especializado para un tipo de dato específico, considera utilizar especialización de plantillas en lugar de agregar lógica compleja dentro de la plantilla genérica. Esto mejorará la legibilidad y el rendimiento del código.

Ejemplos:

- 1- Plantilla de función para encontrar el valor máximo:

```
template <typename T>
T encontrarMaximo(const T& a, const T& b) {
    return (a > b) ? a : b;
}

int maximoEntero = encontrarMaximo(5, 10);
double maximoDouble = encontrarMaximo(3.7, 2.5);
```

- 2- Plantilla de clase contenedor genérico:

```
template <typename T>
class Contenedor {
private:
    T dato;
public:
    Contenedor(const T& valor) : dato(valor) {}
    T obtenerDato() const { return dato; }
};

Contenedor<int> contenedorEntero(5);
Contenedor<std::string> contenedorString("Hola");
```

- 3- Plantilla de función para calcular el promedio de un arreglo:

```

template <typename T, int size>
T calcularPromedio(const T (&arreglo)[size]) {
    T suma = 0;
    for (int i = 0; i < size; ++i) {
        suma += arreglo[i];
    }
    return suma / size;
}

int arregloEntero[] = {1, 2, 3, 4, 5};
double arregloDouble[] = {2.5, 3.7, 1.8, 4.2};
int promedioEntero = calcularPromedio(arregloEntero);
double promedioDouble = calcularPromedio(arregloDouble);

```

4- Plantilla de función para intercambiar dos valores:

```

template <typename T>
void intercambiar(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}

int a = 5, b = 10;
intercambiar(a, b);

```

5- Plantilla de función para encontrar el elemento máximo en un contenedor:

```

template <typename T>
T encontrarMaximo(const T& contenedor) {
    T maximo = contenedor[0];
    for (int i = 1; i < contenedor.size(); ++i) {
        if (contenedor[i] > maximo) {
            maximo = contenedor[i];
        }
    }
    return maximo;
}

std::vector<int> vec = {1, 5, 3, 7, 2};
int maximo = encontrarMaximo(vec);

```

## EXPLICIT EN C++

Imagina que tienes una clase llamada "MiClase" que tiene un constructor que recibe un entero. Al marcar este constructor como "explicit", le estás diciendo al compilador que no permita conversiones automáticas implícitas de enteros a objetos de tipo "MiClase". Esto significa que no puedes simplemente asignar un entero a un objeto "MiClase" sin utilizar una sintaxis explícita.

Por ejemplo, supongamos que tenemos el siguiente código:

```
class MiClase {
public:
    explicit MiClase(int valor) : dato(valor) {}
    int obtenerDato() const { return dato; }
private:
    int dato;
};

void funcionEjemplo(const MiClase& objeto) {
    // Código
}

int main() {
    MiClase objeto1(5); // Constructor llamado explícitamente
    int dato = objeto1.obtenerDato();

    // MiClase objeto2 = 10; // Error: conversión implícita no permitida

    MiClase objeto3 = MiClase(10); // OK: conversión explícita permitida

    funcionEjemplo(objeto3); // OK: paso explícito del objeto como parámetro

    return 0;
}
```

En este código, puedes crear un objeto "MiClase" llamado "objeto1" y pasarle el valor 5 al constructor. Sin embargo, no puedes hacer algo como "MiClase objeto2 = 10;" porque la conversión implícita de un entero a "MiClase" no está permitida debido a la palabra clave "explicit".

Sin embargo, aún puedes crear un objeto "MiClase" utilizando una conversión explícita, como en "MiClase objeto3 = MiClase(10);". Aquí, estás indicando de manera explícita que deseas convertir el entero 10 en un objeto "MiClase".

En general, el uso de "explicit" en un constructor ayuda a evitar conversiones automáticas inesperadas y hace que el código sea más claro y fácil de entender.





Operator+

# SOLID

Los principios SOLID son un conjunto de principios de diseño orientados a objetos que promueven la creación de código limpio, modular y fácil de mantener. Estos principios se basan en la idea de escribir código que sea flexible, escalable y que pueda adaptarse a cambios en los requisitos o funcionalidades sin causar efectos secundarios indeseables.

A continuación, te explicaré brevemente cada uno de los principios SOLID y cómo se aplican en C++:

- **Principio de Responsabilidad Única (Single Responsibility Principle, SRP):** Una clase debe tener una única responsabilidad y solo debe tener una razón para cambiar. Esto significa que cada clase debe estar enfocada en una única tarea o funcionalidad. En C++, puedes asegurarte de cumplir con este principio dividiendo la funcionalidad en clases separadas, donde cada clase tiene una responsabilidad bien definida.
- **Principio de Abierto/Cerrado (Open/Closed Principle, OCP):** Las entidades de software deben estar abiertas para su extensión pero cerradas para su modificación. Esto significa que debemos diseñar nuestras clases y módulos de manera que puedan ser extendidos sin modificar su código fuente existente. En C++, esto se puede lograr utilizando herencia, interfaces y polimorfismo para permitir agregar nuevas funcionalidades sin alterar el código existente.
- **Principio de Sustitución de Liskov (Liskov Substitution Principle, LSP):** Las instancias de una clase base deben poder ser reemplazadas por instancias de sus clases derivadas sin alterar la funcionalidad del programa. Esto significa que cualquier clase derivada debe ser capaz de ser utilizada en lugar de la clase base sin causar errores o comportamientos inesperados. En C++, esto se logra siguiendo las reglas de herencia y asegurándose de que los objetos derivados se comporten correctamente en todos los contextos en los que se usen.
- **Principio de Segregación de Interfaces (Interface Segregation Principle, ISP):** Los clientes no deben depender de interfaces que no utilizan. En lugar de tener una única interfaz grande y compleja, es mejor tener varias interfaces más pequeñas y específicas. De esta manera, las clases solo implementarán las interfaces que necesiten realmente. En C++, puedes utilizar herencia múltiple o crear interfaces más especializadas para seguir este principio.
- **Principio de Inversión de Dependencia (Dependency Inversion Principle, DIP):** Los módulos de alto nivel no deben depender de módulos de bajo nivel, ambos deben depender de abstracciones. Además, las abstracciones no deben depender de los detalles, sino que los detalles deben depender de las abstracciones. Este principio promueve el uso de interfaces o clases base abstractas para la comunicación entre módulos, en lugar de depender directamente de implementaciones concretas. En C++, esto se logra utilizando punteros, referencias o smart pointers para referenciar las abstracciones en lugar de las implementaciones concretas.

La aplicación de los principios SOLID en C++ ayuda a crear un código modular, flexible y de alta calidad. Estos principios fomentan el diseño orientado a objetos, la reutilización de código y la separación de preocupaciones, lo que facilita la escalabilidad y el mantenimiento del código a medida que los proyectos crecen y evolucionan.

### Ejemplo de Principio de Responsabilidad Única:

```
class DatabaseConnector {
public:
    void connect() {
        // Código para establecer la conexión con la base de datos
    }
    void disconnect() {
        // Código para cerrar la conexión con la base de datos
    }
    void executeQuery(const std::string& query) {
        // Código para ejecutar una consulta en la base de datos
    }
};

class DataValidator {
public:
    bool isValidEmail(const std::string& email) {
        // Código para validar si una dirección de correo electrónico es válida
    }

    bool isValidPhoneNumber(const std::string& phoneNumber) {
        // Código para validar si un número de teléfono es válido
    }
};

class ReportGenerator {
public:
    void generatePDFReport(const std::vector<std::string>& data) {
        // Código para generar un informe en formato PDF con los datos proporcionados
    }
    void generateCSVReport(const std::vector<std::string>& data) {
        // Código para generar un informe en formato CSV con los datos proporcionados
    }
};
```

En este ejemplo, tenemos tres clases diferentes, cada una con su propia responsabilidad:

- La clase DatabaseConnector se encarga de establecer y cerrar la conexión con una base de datos, así como de ejecutar consultas en ella.
- La clase DataValidator se encarga de validar diferentes tipos de datos, como direcciones de correo electrónico y números de teléfono. Contiene métodos específicos para realizar estas validaciones.
- La clase ReportGenerator se encarga de generar informes en diferentes formatos, como PDF y CSV, utilizando los datos proporcionados. Tiene métodos dedicados a generar informes en cada formato.

Cada clase tiene una única responsabilidad y se enfoca en su tarea específica. Esto facilita el mantenimiento del código y la reutilización de estas clases en otros contextos. Si se necesita cambiar o mejorar la funcionalidad de alguna de estas responsabilidades, solo se tiene que modificar la clase correspondiente sin afectar a las demás.

## OPERATOR EN C++

puedes sobrecargar los operadores existentes o definir nuevos operadores personalizados para tus tipos de datos personalizados mediante el uso de funciones miembro o funciones amigas. Esto se conoce como sobrecarga de operadores.

La sobrecarga de operadores te permite definir el comportamiento de los operadores en relación con tus tipos de datos personalizados, lo que hace que tu código sea más expresivo y legible. Al sobrecargar un operador, puedes definir cómo se comporta al operar con objetos de tu clase.

Aquí hay algunos ejemplos de operadores que se pueden sobrecargar en C++:

- Operadores aritméticos: +, -, \*, /, %
- Operadores de asignación: =, +=, -=, \*=, /=
- Operadores de comparación: ==, !=, >, <, >=, <=
- Operadores lógicos: &&, ||, !
- Operadores de incremento/decremento: ++, --
- Operadores de acceso a elementos: [], ()
- Operadores de flujo de entrada/salida: <<, >>

Aquí hay un ejemplo de sobrecarga del operador de suma (+) para una clase Vector que representa un vector matemático:

```
class Vector {
public:
    double x, y;

    Vector(double _x, double _y) : x(_x), y(_y) {}

    Vector operator+(const Vector& other) const {
        return Vector(x + other.x, y + other.y);
    }
};

int main() {
    Vector v1(1.0, 2.0);
    Vector v2(3.0, 4.0);

    Vector sum = v1 + v2; // Utiliza el operador sobrecargado de suma

    return 0;
}
```

En este ejemplo, la clase Vector sobrecarga el operador + para permitir la suma de dos objetos Vector. La función miembro operator+ define cómo se realiza la suma de los componentes x e y de los vectores y devuelve un nuevo objeto Vector que representa la suma de los dos vectores originales.

La sobrecarga de operadores te permite personalizar el comportamiento de los operadores en tu código, lo que puede hacerlo más intuitivo y legible. Sin embargo, debes usar la sobrecarga

de operadores con precaución y seguir las convenciones y buenas prácticas para evitar confusiones y mantener la claridad en tu código.

Por ejemplo, podemos definir el operador + para nuestra clase Persona, de modo que la adición de dos objetos Persona devuelva un nuevo objeto que represente la suma de las edades de ambas personas:

```
#include<iostream>
class Persona {
public:
    Persona(int edad) : edad(edad) {}

    Persona operator+(const Persona& other) const {
        return Persona(edad + other.edad);
    }

    int getEdad() const { return edad; }
private:
    int edad;
};

int main() {
    Persona persona1(25);
    Persona persona2(30);
    Persona persona3 = persona1 + persona2;

    std::cout<< "La suma de las edades es: " << persona3.getEdad() << std::endl;

    return 0;
}
```

- 1- Operador de igualdad (==) para comparar dos objetos de una clase Punto que representan coordenadas x e y:

```

class Punto {
public:
    Punto(int x, int y) : x(x), y(y) {}

    bool operator==(const Punto& other) const {
        return (x == other.x) && (y == other.y);
    }

private:
    int x;
    int y;
};

int main() {
    Punto punto1(2, 3);
    Punto punto2(2, 3);

    if (punto1 == punto2) {
        std::cout << "Los puntos son iguales" << std::endl;
    } else {
        std::cout << "Los puntos son diferentes" << std::endl;
    }

    return 0;
}

```

- 2- Operador de suma (+) para concatenar dos objetos de la clase Cadena que representan cadenas de texto:

```

#include<iostream>
class Cadena {
public:
    Cadena(const std::string& texto) : texto(texto) {}

    Cadena operator+(const Cadena& other) const {
        return Cadena(texto + other.texto);
    }

    const std::string& getTexto() const {
        return texto;
    }

private:
    std::string texto;
};

int main() {
    Cadena cadena1("Hola, ");
    Cadena cadena2("mundo!");
    Cadena resultado = cadena1 + cadena2;

    std::cout << "La cadena resultante es: " << resultado.getTexto() << std::endl;

    return 0;
}

```

- 3- Operador de preincremento (++) para incrementar en uno el valor de un objeto de la clase Contador:

```

#include<iostream>
class Contador {
public:
    Contador(int valor) : valor(valor) {}

    Contador& operator++() {
        ++valor;
        return *this;
    }

    int getValor() const {
        return valor;
    }

private:
    int valor;
};

int main() {
    Contador contador(5);
    ++contador;

    std::cout << "El valor del contador es: " << contador.getValor() << std::endl;

    return 0;
}

```

- 4- Operador de asignación (=) para copiar el valor de un objeto de la clase Fecha en otro objeto de la misma clase:

```
#include<iostream>
class Fecha {
public:
    Fecha(int dia, int mes, int anio) : dia(dia), mes(mes), anio(anio) {}

    Fecha& operator=(const Fecha& other) {
        if (this != &other) {
            dia = other.dia;
            mes = other.mes;
            anio = other.anio;
        }
        return *this;
    }
private:
    int dia;
    int mes;
    int anio;
};

int main() {
    Fecha fecha1(10, 5, 2022);
    Fecha fecha2(25, 12, 2022);
    fecha2 = fecha1;

    return 0;
}
```

- 5- Sobrecarga del operador de suma (+) para dos objetos de una clase "Vector":



```

class Vector {
    int x, y;
public:
    Vector() {}
    Vector(int a, int b) : x(a), y(b) {}

    Vector operator+(const Vector& v) {
        Vector result;
        result.x = x + v.x;
        result.y = y + v.y;
        return result;
    }

    void display() {
        cout << "Vector (" << x << ", " << y << ")" << endl;
    }
};

int main() {
    Vector v1(2, 3), v2(4, 5);
    Vector v3 = v1 + v2;
    v3.display();
    return 0;
}

```

- 6- Sobrecarga del operador de pre-incremento (++) para un objeto de una clase "Contador":

```
class Contador {
    int count;
public:
    Contador() : count(0) {}

    Contador operator++() {
        ++count;
        return *this;
    }

    void display() {
        cout << "Contador: " << count << endl;
    }
};

int main() {
    Contador c1, c2;
    ++c1;
    ++c1;
    c1.display();
    ++c2;
    c2.display();
    return 0;
}
```

## ESCRITURA DE ARCHIVOS

se realiza a través de objetos de la clase ofstream, que proporcionan una serie de métodos para abrir, escribir y cerrar archivos.

Se ha creado un objeto llamado "archivo" de la clase ofstream y se ha abierto el archivo "datos.txt" para escritura. El método is\_open() se utiliza para verificar que el archivo se ha abierto correctamente.

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

int main() {
    ofstream archivo("datos.txt");
    if (!archivo.is_open()) {
        // manejo de error
    }
    //...
    archivo.close();
    return 0;
}
```

Para escribir utilizamos el operador '<<'

```
archivo << "Hola, mundo!" << endl;
```

se ha escrito la cadena "Hola, mundo!". También se ha utilizado el manipulador endl para agregar un salto de línea.

También no olvidarse de cerrar el archivo y liberar recursos asociados.

```
archivo.close();
```

Ejemplos:

se ha definido una estructura Persona que contiene un nombre y una edad. Se crea una estructura con los valores "Juan" y 25, y se ha escrito en un archivo binario llamado "personas.dat" utilizando el método write(). Para escribir la estructura, se ha utilizado la función reinterpret\_cast() para convertir el puntero del objeto Persona a un puntero a char.

```

#include <fstream>
using namespace std;

struct Persona {
    string nombre;
    int edad;
};

int main() {
    Persona p = {"Juan", 25};
    ofstream archivo("personas.dat", ios::binary);
    archivo.write(reinterpret_cast<const char*>(&p), sizeof(Persona));
    archivo.close();
    return 0;
}

```

Profundicemos un poco más:

```

#include <iostream>
#include <fstream>

using namespace std;

struct Persona {
    string nombre;
    int edad;
};

int main() {
    Persona p = {"Juan", 30};

    ofstream archivo("datos.txt");
    if (!archivo.is_open()) {
        cout << "Error al abrir el archivo" << endl;
        return 1;
    }

    archivo.write((char*)&p, sizeof(p));

    archivo.close();

    return 0;
}

```

Ejemplo 2: Escribir un array de enteros (valores del 1 al 5) en un archivo “números.txt”, utilizando for

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

int main() {
    int numeros[] = {1, 2, 3, 4, 5};
    ofstream archivo("numeros.txt");
    for (int i = 0; i < 5; i++) {
        archivo << numeros[i] << " ";
    }
    archivo.close();
    return 0;
}
```

Profundicemos un poco más:

```
#include <iostream>
#include <fstream>
#include <vector>

using namespace std;

int main() {
    vector<int> vec = {1, 2, 3, 4, 5};

    ofstream archivo("datos.txt");
    if (!archivo.is_open()) {
        cout << "Error al abrir el archivo" << endl;
        return 1;
    }

    for (auto i : vec) {
        archivo << i << " ";
    }

    archivo.close();

    return 0;
}
```

### Ejemplo 3: Escribir en un archivo desde una cadena de caracteres

```
#include <iostream>
#include <fstream>
#include <cstring>

using namespace std;

int main() {
    const char* cadena = "Hola, mundo!";

    ofstream archivo("datos.txt");
    if (!archivo.is_open()) {
        cout << "Error al abrir el archivo" << endl;
        return 1;
    }

    archivo.write(cadena, strlen(cadena));

    archivo.close();

    return 0;
}
```

### Ejemplo 04: Escribir en un archivo utilizando el operador de flujo

```
#include <iostream>
#include <fstream>

using namespace std;

int main() {
    ofstream archivo("datos.txt");
    if (!archivo.is_open()) {
        cout << "Error al abrir el archivo" << endl;
        return 1;
    }

    archivo << "Hola, mundo!" << endl;

    archivo.close();

    return 0;
}
```



Ejercicios: Desarrollar un juego que permita jugar con la PC el juego de Papel, piedra y tijera



```
js Juego1.js > ...
1  var jugador = 'papel';
2  var pc = 'papel';
3  var resultado;
4
5  function ganador (){
6      if(jugador === pc){
7          return resultado = 'empate';
8      }else if((jugador=='piedra' && pc=='papel') || (jugador=='papel'&&pc=='tijera') || (jugador=='tijera'&&pc=='piedra')){
9          return resultado = 'Gana PC';
10     }else if((jugador == 'piedra' && pc=='tijera') || (jugador=='papel'&&pc=='piedra') || (jugador=='tijera'&&pc=='papel')){
11         return resultado = 'Gana Jugador';
12     }else{
13         return resultado = 'Ingrese valores correctos';
14     }
15 }
16 console.log(ganador());
```

Con switch, piedra papel y tijera

```
js switch.js
1  let jugador = "papel"
2  let enemigo = "tijera"
3
4  function resultado() {
5      if (jugador === enemigo) {
6          console.log("Empate");
7      }else{
8          switch (true) {
9              case jugador == "tijera" && enemigo == "papel":
10                 case jugador == "piedra" && enemigo == "tijera":
11                 case jugador == "papel" && enemigo == "piedra":
12                     console.log("Ganaste");
13                     break;
14                 default:
15                     console.log("Perdiste");
16                     break;
17             }
18         }
19     }
20
21     resultado()
```



## FUNCIONES MAS UTILIZADAS

**sizeof()**.- número de bytes que ocupa cualquier estructura de datos

```
#include <stdio.h>
#define NAME_LENGTH 10
#define TABLE_SIZE 100
#define UNITS_NUMBER 10

struct unit
{ /* Define a struct with an internal union */
    int x;
    float y;
    double z;
    short int a;
    long b;
    union
    { /* Union with no name because it is internal to the
struct */
        char name[NAME_LENGTH];
        int id;
        short int sid;
    } identifier;
};

int main(int argc, char *argv[])
{
    int table[TABLE_SIZE];
    struct unit data[UNITS_NUMBER];

    printf("%d\n", sizeof(struct unit)); /* Print size of
structure */
    printf("%d\n", sizeof(table));      /* Print size of
table of ints */
    printf("%d\n", sizeof(data));       /* Print size of
table of structs */

    return 0;
}
```

**Getch()**.-

**System("cls")**.-

## **Getline.-**

```
string str1;  
cout<<"ingrese su nombre";  
getline(cin, str1);
```

# PROYECTO DE CAMPO

## Proyecto Estructurados

**Proyecto 1:** Se tiene una tienda con nombre del negocio, eslogan, dirección y RUC, el sistema realiza la compra, venta y stock de productos, se pide:

- Se debe de crear un modulo de configuración
- Se maneja roles (Administrador, cajero)
  - o Administrador
    - Ventas diarias por caja
    - Crear, listar y ver productos
  - o Cajero
    - Desarrolla venta (boleta)
    - Listar y ver productos
- Seguridad (Permitir 3 intentos)
  - o Cada rol tiene usuario y contraseña
- Productos
  - o Crear, listar y ver productos

**Proyecto 2:** Se pide una estadística de ventas, para ver el estado del negocio (Gastos, compro y ventas).

- Gestión de configuración (Editar nombre del negocio, eslogan, dirección, número de serie y RUC)
- Seguridad (Manejo de errores)
- Gestionar boletas (CRUD- Crear, editar, actualizar y eliminar)
- Reportes (Gastos, compras y ventas)

**Proyecto 3:** Ahora tenemos un Minimarket, se pide automatizar sus ventas y gestionar sus productos, utilizando una boleta electrónica.

- Manejar roles
  - o Dueño
    - Gestión de producto (CRUD)
    - Gestionar Usuario (CRUD)
    - Gestionar las ventas (CRUD)
  - o Usuario
    - Realizar venta (Fecha, productos y totales)
    - Gestionar clientes (Nombre, dirección y DNI)
    - Cierre de caja diaria
- Trabajar con los impuestos
- Reporte
  - o Ventas diarias (Fechas)
  - o Mejor vendedor
- Tener productos, clientes, ventas y usuario en archivos TXT

**Problema 4:** desarrollar una calculadora científica, se tiene que introducir a una temática de matemática.

- Operaciones básicas (+, -, \*, /)
- Fórmulas matemáticas (áreas, perímetros, volúmenes)
- Conversión de unidades (metros, pulgadas, centímetros)
- Ecuaciones de primer grado
- Gráficos básicos (Circunferencia, rectángulo y triángulo)

Observación: estas pueden ser orientadas a operaciones desarrolladas en la carrera de sistemas, por ejemplo: Conversiones de base (octales, binarias y hexadecimales)

**Problema 5:** se tiene una tienda de venta de autos para llevar un sistema de cuentas de ingreso (ventas) y egresos (pagos), es importante conocer cual es el punto de equilibrio para que una empresa pueda pagar a sus empleados y servicios.

- Configurar para la empresa (negocio, datos de la empresa)
- Gestión de administrador
  - o Genera reportes financieros
  - o Reporte de ventas
  - o Reporte de gastos
- Gestión de empleados (uso de accesos, nombre, sueldo)
  - o Pago mensual de comisionado (empleado)
- Gestión de autos (marca, modelo, precio y comisión)

**Problemas 6:** Mejorar la experiencia usuaria al interactuar con los servicios gubernamentales, brindar información sobre los requisitos y si el beneficiario es apto.

- Registro y accesibilidad de servicios
- Postulantes

**Problemas 7:** Catalogo de vehículos (Carros y motos)

- Gestión cliente ()
- Gestión administradora ()
- Gestión vehículo (marca, modelo, color, año, tipo)



## **Proyectos orientados a objetos**

## **Evaluaciones**

**Test**

## Quiz



## **BIBLIOGRAFÍA**

### **Plataforma OpenEGD**

### **Libros**

### **Web**

- <http://agora.pucp.edu.pe/inf2170681/>

## **ANEXOS**

**Rubrica para proyecto final**

**Formato del proyecto 50%**


**Formato del informe final del proyecto**

## MATERIAL DE CERTIFICACIÓN

About Terms and Conditions Support Search Buy Vouchers

**X OPENEDG**

Projects Products & Services Partners



# STUDY. CERTIFY. SUCCEED.

Join the global community of developers and change-makers, and make an impact!

<https://openedg.org/>

## ERRORES DE SOFTWARE

UNITED ARTISTS

Longer video of 'Ariane 5' Rocket firs... Ver más ta... Compartir



MÁS VÍDEOS

00:00:17

YouTube

<https://www.rtve.es/noticias/20140604/error-software-convirtio-lanzamiento-espacial-carisimos-fuegos-artificiales/948262.shtml>

