

UNITEC^{MR}

Universidad Tecnológica de México

Estructura de datos

Séptimo Cuatrimestre

Clave de la asignatura: ICM008



Datos del docente

Nombre: Miguel Ángel González Fabiani
Estudios:

- Licenciado en Diseño para la Comunicación en medios digitales
- Maestro en Diseño Multimedia
- Especialización en Aplicaciones Móviles y Web

Experiencia:

- Docencia
- Agencias de publicidad y periodísticas, sector educativo.
- Desarrollo de interfaces para dispositivos móviles



Evaluación



Criterio a evaluar	Porcentaje	Total
Evaluación parcial		
Examen parcial	20%	40%
Prácticas	20%	
Evaluación final		
Examen parcial	25%	50%
Prácticas	25%	
Actividades de aprendizaje	10%	

¿Qué es una estructura de datos?

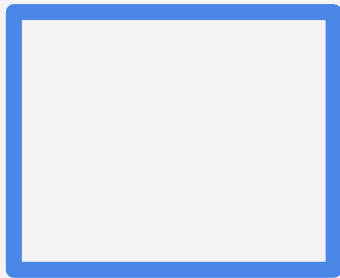
- Los datos a procesar por un computador se clasifican en: simples y estructurados. **Los datos simples** ocupan **solo una casilla de memoria**, por lo tanto una variable simple hace referencia a un único valor a la vez.
- Los **datos estructurados** se caracterizan **por tener un nombre** (el identificador de la variable estructurada) con el cual se hace referencia a un **grupo de casillas de memoria**. Es decir, un dato estructurado tiene varios componentes. Cada uno de los componentes puede ser a su vez un dato simple o estructurado.

DEFINICIÓN

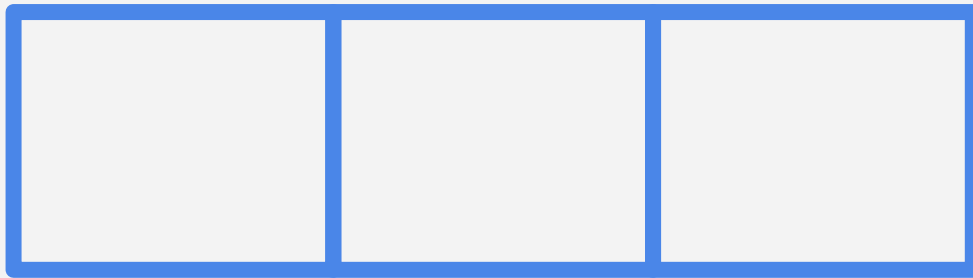
Una estructura de datos es un conjunto de elementos, que se relacionan entre sí y que se pueden operar como un todo.



Diferencia entre dato simple y estructurado



DATO SIMPLE



DATO ESTRUCTURADO

Clasificación de las estructuras de datos



Elementos de un programa en C++

A continuación se muestran ejemplos de identificadores válidos:

A	Nombre	Nombre_alumno	X1
a	_codigo	CODIGO	resultado_3

A continuación se muestran ejemplos de identificadores no válidos:

Identificador	Explicación
3id	El primer carácter debe ser una letra
Alumno#	El carácter # no es permitido
Codigo alumno	El espacio en blanco no es permitido
Codigo-alumno	El carácter - no es permitido
“alumno”	El carácter “ no es permitido

Tipos de datos

En un programa siempre se procesan datos, los cuales pueden ser de distinta naturaleza. Dependiendo de su tipo, se representará y almacenará el dato en la memoria de la computadora de una manera específica; es decir, el tipo de dato determina la cantidad de memoria requerida para almacenarlo.

Los tipos de datos básicos (o primitivos) son:

- **Enteros (Int)**
- **Float**
- **Double**
- **Char**
- **Booleanos**

Tipo	Descripción	Cantidad de memoria requerida	Rango
void	Vacío o valor null		
char	Almacena un carácter	1 byte	-128 a 127
unsigned char	Almacena un carácter o un valor sin signo	1 byte	0 a 255
int	Define un valor numérico entero	2 bytes	-32767 a 32768
unsigned int	Como short int pero sin signo	2 bytes	0 a 65,535
short int	Entero corto	2 bytes	-32767 a 32768
float	En punto flotante (fracción o entero con exponente)	4 bytes	3.4x10-38 a 1.7x10+38
double	Doble del tamaño del float	8 bytes	1.7x10-308
long int	Entero con signo del doble del tamaño del int	4 bytes	2,147,483,647
unsigned long	Como long int pero sin signo	4 bytes	0 a 4, 294, 967, 295
long double	Increments el tamaño del double	10 bytes	3.4x10-4932

Variables y Constantes

Variables

Para poder ejecutar un programa, es necesario que los datos estén almacenados junto con las instrucciones, en la memoria. Muchas veces dichos datos son **proporcionados por el usuario del programa durante la ejecución del mismo** o son resultado del procesamiento de otros datos.

Una variable es un **espacio de memoria que el programador reserva con el fin de almacenar esos datos desconocidos cuando empieza la ejecución de un programa** o que pueden ir cambiando durante ese proceso.

Constantes

Las constantes son **elementos frecuentemente utilizados en los programas y es un espacio en la memoria que recibe un valor por primera vez** y no se modifica durante la ejecución del programa.

En C++, las CONSTANTES se crean utilizando la directiva del procesador **#define** solo se utiliza para declarar constantes simbólicas y crear macros o bien, la palabra reservada **const**.

Palabras reservadas

Las palabras reservadas de C++ son aquellas cuyo significado se encuentra definido en el lenguaje; es decir, ya tienen uso específico.

Las palabras reservadas de C++ son:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Comentarios

Los comentarios son cadenas de caracteres o texto que describen partes del programa que el programador desea explicar; dicho texto no es parte del programa fuente, si no una descripción del mismo.

Para poder usar los comentarios en C++ y que el compilador no los considere instrucciones del programa, estos deberán estar encerrados de la siguiente manera

```
/*...texto...*/
```



Entradas y salidas en C++

Un programa es un **conjunto de instrucciones que la computadora ejecuta con el fin de obtener un resultado o bien la solución a un problema determinado**. Casi siempre este resultado se obtiene a partir del procesamiento de los datos.

El lenguaje C++ cuenta con las funciones de `scanf()` y `printf()` para entrada y salida de datos respectivamente, las cuales se pueden utilizar agregando el archivo de cabecera **`#include <studio.h>`**

La computadora dispone de diversos medios para proporcionar la salida de datos como la impresora, archivos o el más utilizado: el monitor.

Su sintaxis es la siguiente:

- **`Printf ("texto, cadena de control de tipo", argumentos);`**





01

Funciones

- Conocer las funciones.
- Explicar la definición de función, y la declaración de función.
- Identificar el prototipo de funciones y la aplicación de funciones.

1. Funciones

¿Qué es una función?

En programación, una función es como una subrutina o un procedimiento que realiza una tarea específica. Es un bloque de código que se puede definir una vez y luego llamar desde diferentes partes del programa tantas veces como sea necesario. Esto permite:

- **Modularizar el código:** Dividir un programa en partes más pequeñas y manejables, facilitando su comprensión y mantenimiento.
- **Reutilizar código:** Evitar repetir el mismo código varias veces, lo que hace el código más conciso y eficiente.
- **Organizar el código:** Agrupar tareas relacionadas en funciones, mejorando la estructura del programa.

La función puede ser invocada, o llamada, desde cualquier lugar del programa. Los valores que se **pasan a la función son los argumentos**, cuyos tipos deben ser compatibles con los tipos de los parámetros en la definición de la función.

No hay ningún límite práctico para la longitud de la función, pero un buen diseño tiene como objetivo funciones que realizan una sola tarea bien definida. **Los algoritmos complejos deben dividirse en funciones más sencillas** y fáciles de comprender.

Las funciones pueden ser **sobrecargadas**, lo que significa que diferentes versiones de una función pueden **compartir el mismo nombre si difieren por el número y/o tipo de parámetros formales**.

1. Funciones

¿Para qué sirven las funciones?

- **Calcular valores:** Por ejemplo, calcular el área de un círculo, el factorial de un número, etc.
- **Realizar operaciones:** Cómo ordenar una lista, buscar un elemento en un arreglo, etc.
- **Mostrar información en pantalla:** Imprimir mensajes, resultados de cálculos, etc.
- **Interactuar con el usuario:** Leer datos desde el teclado, mostrar menús, etc.

Ventajas de usar funciones:

- **Mayor claridad y legibilidad:** El código se vuelve más fácil de entender al dividirlo en funciones con nombres descriptivos.
- **Reutilización:** Las funciones pueden ser llamadas desde diferentes partes del programa, evitando la duplicación de código.
- **Facilidad de depuración:** Si hay un error en una función, es más fácil aislarlo y corregirlo.
- **Modularidad:** Permite dividir un programa grande en módulos más pequeños, facilitando su desarrollo y mantenimiento.

Estructura básica de un programa en C++

Tipo de dato: El tipo de valor que la función devuelve. Si no devuelve nada, se utiliza **void**.

Nombre de función: Un identificador único que se usa para llamar a la función.

Lista de parámetros:

Variables que se pasan a la función como entrada.

Cuerpo de la función: Las instrucciones que realiza la función.

Valor de retorno: El valor que la función devuelve al finalizar.

```
tipo_de_dato nombre_funcion(lista_de_parámetros) {  
    // Cuerpo de la función  
    // Instrucciones que realizan la tarea  
    return valor_de_retorno; // Opcional, si la función devuelve un valor  
}
```

```
int sumar(int a, int b) {  
    int resultado = a + b;  
    return resultado;  
}
```

Estructura de C++

Como una cuestión de buena forma de programación, el siguiente ordenamiento de instrucciones deberá formar la estructura básica alrededor de la cual se construyen todos los programas en C++.

```
/* ----- LIBRERIAS Y MACROS ----- */
#include <iostream>
using namespace std;
/* ----- */

/* ----- VARIABLES GLOBALES ----- */
/* ----- */

/* ----- PROTOTIPO DE FUNCIONES ----- */
/* ----- */

int main()
{
    /* ----- VARIABLES LOCALES ----- */
    /* ----- */
}

/* ----- DEFINICIÓN DE FUNCIONES ----- */
/* ----- */
```

1.1. Definición de funciones

Las funciones son una herramienta fundamental en la programación que permite organizar el código, mejorar su legibilidad y reutilizabilidad. Al dividir un programa en funciones, se facilita la comprensión y el mantenimiento del mismo.

1.2. Declaración de función

En programación, **una declaración de función, también conocida como prototipo de función**, es como una tarjeta de presentación que le damos al compilador sobre una función que vamos a utilizar.

Esta declaración le informa al compilador el nombre de la función, el tipo de datos que devuelve (tipo de retorno) y los tipos de datos de los parámetros que recibe.

¿Para qué sirve una declaración de función?

- **Organización del código:** Ayuda a estructurar el código, indicando al compilador qué funciones existen y cómo se utilizan antes de que se definan completamente.
- **Evita errores de sintaxis:** Al declarar una función, el compilador puede verificar si la función se está utilizando correctamente en términos de nombre, número y tipos de parámetros.
- **Permite la compilación separada:** Las declaraciones de función permiten dividir un programa en múltiples archivos, facilitando la organización y el mantenimiento del código.

1.3. Prototipo de funciones

En C++ es obligatorio **declarar una función antes de ser usada**, esta declaración se conoce como **prototipo**. Los prototipos de una función contienen la misma cabecera de la función a excepción de que esta lleva punto y coma al final, por ejemplo:

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

Este es un prototipo de la función suma, podemos notar que sólo colocamos la cabecera de la función sin definirla, es decir, sin cerrar dentro de llaves líneas de código, además, la finalizamos con punto y coma.

1.4. Aplicaciones de funciones

Inline

- **¿Qué es?**
 - Indica al compilador que intente insertar el cuerpo de una función directamente en el lugar donde se llama, en lugar de generar un salto a una ubicación separada de memoria.
- **Beneficios:**
 - **Reducción de sobrecarga de llamadas:** Elimina la sobrecarga asociada con las llamadas a funciones, lo que puede mejorar el rendimiento en funciones pequeñas y simples.
 - **Optimización:** Permite al compilador realizar optimizaciones más agresivas dentro de la función en línea.
- **Cuándo usar:**
 - Funciones cortas y simples que se llaman con frecuencia.
 - Funciones que no tienen efectos secundarios significativos (como modificar variables globales).

1.4. Aplicaciones de funciones

`noexcept`

- **¿Qué es?**
 - Indica que una función no lanzará ninguna excepción.
- **Beneficios:**
 - **Optimización:** Permite al compilador realizar optimizaciones más agresivas, ya que no tiene que preocuparse por el manejo de excepciones.
 - **Seguridad:** Ayuda a prevenir errores en tiempo de ejecución causados por excepciones no manejadas.
- **Cuándo usar:**
 - Funciones que no pueden fallar o que no tienen sentido lanzar una excepción.
 - Funciones que llaman a otras funciones que también son `noexcept`.

1.4. Aplicaciones de funciones

constexpr

- **¿Qué es?**
 - Indica que una función puede ser evaluada en tiempo de compilación si todos sus argumentos son conocidos en ese momento.
- **Beneficios:**
 - **Optimización:** Permite realizar cálculos en tiempo de compilación, lo que puede mejorar el rendimiento y reducir el tamaño del código.
 - **Seguridad:** Ayuda a prevenir errores en tiempo de ejecución causados por valores no inicializados.
- **Cuándo usar:**
 - Funciones que realizan cálculos simples y deterministas.
 - Inicialización de variables de tipo constante.

1.4. Aplicaciones de funciones

Diferencias y Combinaciones

- **inline vs noexcept:** `inline` se enfoca en la optimización del código, mientras que `noexcept` se enfoca en la seguridad y la prevención de excepciones. Ambas pueden utilizarse juntas.
- **constexpr vs inline:** `constexpr` implica `inline` de forma implícita, ya que una función `constexpr` debe ser evaluada en tiempo de compilación y, por lo tanto, debe ser insertada en el código donde se llama.
- **Combinaciones:** Es posible combinar las tres palabras clave: `constexpr inline noexcept`.

Práctica

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

//Prototipo de Funcion

```
int MaxBetweenTwoNumbers(int n1, int n2);
```

```
int main()
```

```
{
```

```
    int n1,n2;
```

```
    cout<<"Digite primer numero: ";
```

```
    cin>>n1;
```

```
    cout<<"Digite segundo numero: ";
```

```
    cin>>n2;
```

```
    cout << "El mayor es: " <<
```

```
    MaxBetweenTwoNumbers(n1, n2) << endl;
```

```
}
```

```
int MaxBetweenTwoNumbers(int n1, int n2)
```

```
{
```

```
    int max = 0;
```

```
    if(n1 > n2)
```

```
    {
```

```
        max = n1;
```

```
    }
```

```
    else
```

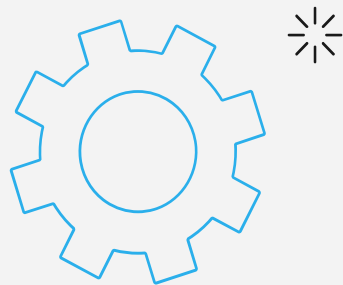
```
    {
```

```
        max = n2;
```

```
    }
```

```
    return max;
```

```
}
```



Práctica

Escriba una función llamada mult() que acepte dos números en punto flotante como parámetros, multiplique estos dos números y despliegue el resultado.

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

//Prototipo de Funcion

```
void pedirDatos();
void mult(float x,float y);
```

```
float n1,n2;
```

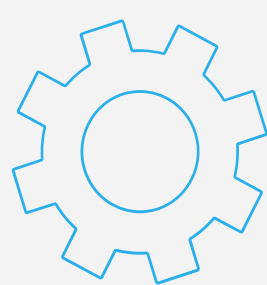
```
int main()
{
    pedirDatos();
    mult(n1,n2);

    return 0;
}
```

```
void pedirDatos()
{
    cout<<"Digite 2 numeros: ";
    cin>>n1>>n2;
}

void mult(float x,float y)
{
    float multiplicacion = n1 * n2;

    cout<<"La multiplicacion es:
"<<multiplicacion<<endl;
}
```



Entregable 1

¿Qué es una función y un apuntador?, explique un ejemplo de cada uno en lenguaje de programación C++.

Integrar los siguientes aspectos:

- Portada
- Desarrollo
- Conclusión
- Referencias bibliográficas

Bibliografía

- Zohonero Martínez, I. & Joyanes Aguilar, L. (2008). *Estructuras de datos en Java*. McGraw-Hill España.
- Ruiz Rodríguez, R. (2009), Fundamentos de la programación orientada a objetos: una aplicación a las estructuras de datos en Java.
- Malik, D. S. (2013), Estructuras de datos con C++ (2a. ed.). Cengage Learning México.
- Joyanes Aguilar, L. (2006), Programación en C++: Algoritmos, Estructuras de datos y objetos (2a. ed.). McGraw-Hill España.



02

Apuntadores

- Describir la definición de apuntador.
- Diferenciar el operador de dirección de memoria y el operador de indirección de memoria y Operador NULL.
- Conocer el apuntador de apuntadores y como enviar y recibir apuntadores en una función.

2. Apuntadores

Un puntero es una variable que almacena la dirección de memoria de un objeto. Los punteros se usan ampliamente en **C y C++** para tres propósitos principales:

para asignar nuevos objetos en el montón,

para pasar funciones a otras funciones

para iterar los elementos de matrices u otras estructuras de datos.

En la programación de estilo C, se **usan punteros sin procesar para todos estos escenarios**. Sin embargo, los punteros sin procesar son el origen de muchos errores de programación graves. Por lo tanto, su uso se desaconseja fuertemente, excepto cuando proporcionan una ventaja significativa de rendimiento y no hay ambigüedad en cuanto a qué puntero es el puntero propietario que es responsable de eliminar el objeto. C++ moderno proporciona punteros inteligentes para asignar objetos, iteradores para recorrer estructuras de datos para pasar funciones. Mediante el uso de estas instalaciones de lenguaje y biblioteca en lugar de punteros sin procesar, hará que el programa sea más seguro, más fácil de depurar y más fácil de entender y mantener.

2.1. Definición de apuntadores

Un puntero es un tipo de **variable**. Almacena la **dirección de un objeto en memoria y se usa para tener acceso a ese objeto**.

Imaginemos que tu casa tiene varias habitaciones. Cada habitación tiene una dirección específica. Los punteros en C++ funcionan de manera similar, pero en lugar de habitaciones, pensamos en espacios de memoria en tu computadora.

Un puntero es como una pequeña nota adhesiva que puedes pegar en un lugar específico de tu casa (o en este caso, en la memoria de tu computadora). En esa nota, escribes la dirección exacta de otra habitación

¿Para qué sirven los punteros?

- **Encontrar cosas rápidamente:** Si sabes la dirección de una habitación (gracias a la nota adhesiva), puedes ir directamente a ella sin tener que buscarla en toda la casa.
- **Compartir información:** Puedes darle la nota adhesiva a alguien más para que también pueda encontrar esa habitación.
- **Hacer cambios:** Si encuentras algo que quieres modificar en la habitación, puedes usar la nota adhesiva para llegar directamente a ella y hacer los cambios necesarios.

2.1. Definición de apuntadores

¿Por qué son útiles?

Los punteros son muy poderosos en C++ porque te permiten:

- **Manejar memoria de forma eficiente:** Puedes crear y liberar memoria de manera muy precisa.
- **Crear estructuras de datos más complejas:** Como listas enlazadas, árboles, etc.
- **Pasar argumentos a funciones de forma más flexible:** Puedes modificar los valores originales de las variables dentro de una función.

Importante:

- **Cuidado con los punteros:** Si no los usas correctamente, pueden causar errores difíciles de encontrar, como pérdidas de memoria o accesos a zonas de memoria no válidas.
- **Punteros inteligentes:** En C++ moderno, existen punteros inteligentes que ayudan a evitar muchos de estos problemas.

2.2. Operador de dirección de memoria

Una dirección de memoria contiene un byte de información, una variable dependiendo de su tipo puede ocupar uno o más bytes. Un apuntador solo almacena la dirección del primer byte de la variable sin importar que esta sea de más de 1 byte.

Operador & (Ampersand):

- **Obteniendo la dirección de memoria:** Cuando se coloca antes de una variable, el operador & devuelve la dirección de memoria donde se almacena el valor de esa variable. Es como pedir la dirección exacta de una casa.
- **Declaración de referencias:** En C++, el ampersand también se utiliza para declarar referencias.

2.3. Operador de **indirección de memoria**

El Operador de **Indirección (*)**, toma la dirección de una variable y regresa el dato que contiene esa dirección.

Operador * (Asterisco):

- **Dereferenciación:** Cuando se coloca antes de un puntero, el operador * accede al valor almacenado en la dirección de memoria a la que apunta ese puntero. Es como ir a la casa cuya dirección tienes y ver lo que hay dentro.

Conceptos adicionales:

- **Punteros nulos:** Un puntero que no apunta a ninguna dirección de memoria válida se llama puntero nulo.
- **Punteros colgantes:** Un puntero colgante apunta a una memoria que ya ha sido liberada.
- **Aritmética de punteros:** Puedes realizar operaciones aritméticas básicas con punteros (suma, resta), pero debes tener cuidado para no acceder a zonas de memoria no válidas.

¿Qué son los punteros?

Un puntero es una **variable**. Pero no guarda un valor concreto sino la dirección donde se encuentra otro valor (**"el valor apuntador"**)

Contenido de la variable:
"hola"

Dirección de memoria:
31C4

Variable puntero Variable apuntada



¿Qué son los punteros?

Los punteros son una característica fundamental en C++ que permiten una **gestión eficiente y detallada de la memoria** y otros recursos del *hardware*. Los punteros son variables cuya función principal es **almacenar direcciones de memoria**.

¿Qué puede haber en las direcciones de memoria?

- Variables primitivas
- Arrays (Arreglos)
- Clases
- Otro puntero (Apuntador de apuntadores)
- Funciones

Es una variable que va almacenar la dirección de memoria donde se está guardando otra variable.

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    int numero = 5; // Creamos una variable
```

```
    // Creamos un puntero que apunta a numero
```

```
    int *puntero_a_numero = &numero;
```



```
    cout << "El valor de numero es: " << numero << endl;
```

```
    cout << "La dirección de memoria de numero es: " <<
```

```
&numero << endl;
```

```
    cout << "El valor al que apunta el puntero es: " <<
```

```
*puntero_a_numero << endl;
```

```
    return 0;
```

```
}
```

Además de imprimir el número, dame la posición de memoria exacta donde se está guardando dicho número.

Va a guardar la **dirección de memoria** de una variable entera.

Práctica

Comprobar si un número es par o impar, y señalar la posición de memoria donde se está guardando el número.

```
#include<iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int numero, *dir_numero;
```

```
    //Pedimos el numero al usuario
```

```
    cout<<"Digite un numero: ";
```

```
    cin>>numero;
```

```
    //Guardando la posición de memoria
```

```
    dir_numero = &numero;
```

```
    if(*dir_numero%2==0)
```

```
    {
```

```
        cout<<"El numero: "<<*dir_numero<<" es par"<<endl;
```

```
        cout<<"Posicion: "<<dir_numero<<endl;
```

```
    }
```

```
    else{
```

```
        cout<<"El numero: "<<*dir_numero<<" es
```

```
        impar"<<endl;
```

```
        cout<<"Posicion: "<<dir_numero<<endl;
```

```
    }
```

```
    return 0;
```

```
}
```

Práctica

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

void pedirDatos();
void contarVocales(char *);
char palabraUsuario[30];

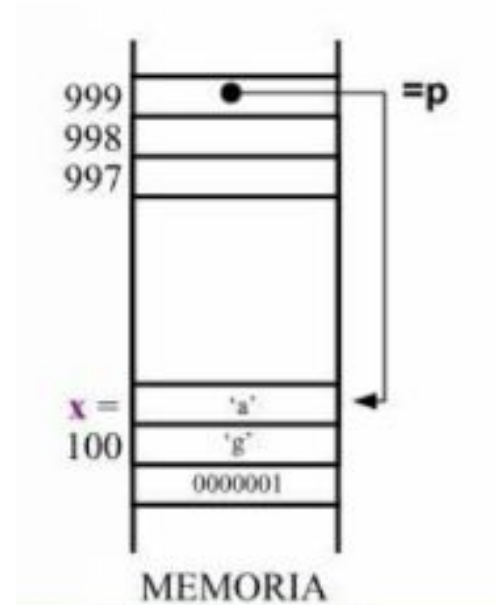
int main()
{
    pedirDatos();
    //Llamada a la funcion para contar vocales del nombre
    contarVocales(palabraUsuario);
}

void pedirDatos()
{
    cout<<"Digite una palabra: ";
    cin.getline(palabraUsuario,30,'\n');
}
```

```
void contarVocales(char *palabra)
{
    int contA=0,contE=0,contI=0,contO=0,contU=0;
    //mientras nombre sea diferente de nulo '\0'
    while(*palabra)
    {
        switch(*palabra)
        {
            case 'A': contA++;break;
            case 'E': contE++;break;
            case 'I': contI++;break;
            case 'O': contO++;break;
            case 'U': contU++;break;
        }
        palabra++;
    }
    //Imprimiendo el conteo de cada vocal
    cout<<"\nNumero de vocales A: "<<contA<<endl;
    cout<<"Numero de vocales E: "<<contE<<endl;
    cout<<"Numero de vocales I: "<<contI<<endl;
    cout<<"Numero de vocales O: "<<contO<<endl;
    cout<<"Numero de vocales U: "<<contU<<endl;
}
```


2.4. Operador NULL

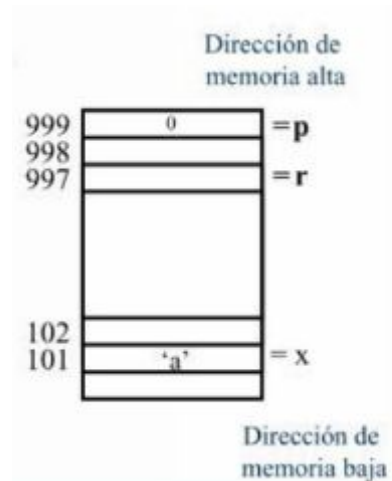
En C++, el valor **NULL** se utiliza para indicar que un **puntero no apunta a ninguna dirección de memoria válida**. Es decir, es como una etiqueta que dice "este puntero no está apuntando a nada en este momento". El valor NULL **es muy útil para la construcción de estructuras de datos dinámicas, como las listas enlazadas, matrices, etc.** Se pueden producir valores nulos al hacer referencia a campos del mensaje que no existen, al acceder a columnas de base de datos para las que no se ha facilitado ningún dato y al utilizar la palabra clave NULL, que suministra un valor literal nulo.



Inicializar un puntero a una dirección nula

¿Cómo se utiliza NULL?

- **Inicialización de punteros:** Al declarar un puntero, es buena práctica inicializarlo con **NULL** para indicar claramente que no apunta a nada hasta que se le asigne una dirección válida.
- **Comprobación de punteros nulos:** Antes de acceder a la memoria a través de un puntero, siempre debes comprobar si es nulo.
- **Retorno de funciones:** Algunas funciones pueden devolver punteros nulos para indicar que no se encontró el valor buscado o que ocurrió un error.



Inicializar un puntero a una dirección nula

Ejemplo de inicialización de un puntero a NULL

```
int *ptr = NULL;
if (ptr != NULL)
{
    // Acceder a la memoria a través de ptr
    *ptr = 10;
}
else
{
    // Manejar el caso en el que ptr sea nulo
    cout << "El puntero es nulo" << endl;
}
```

Ejemplo de retorno de funciones

```
int *buscarElemento(int x, int y)
{
    // Si x es menos a y
    if (x < y)
    {
        // Devolver NULL
        return NULL;
    }
    else
    {
        // Devolver la dirección de la variable x
        return &x;
    }
}
```

2.5. Apuntador de apuntadores

Un apuntador de apuntadores, como su nombre lo indica, es un puntero que apunta a otro puntero. Es decir, es una variable que almacena la dirección de memoria de otro puntero.

¿Para qué sirven? Se utilizan principalmente en:

- **Matrices de punteros:** Cuando se necesita crear arreglos de cadenas o de otros tipos de datos dinámicos, se suelen utilizar matrices de punteros. Cada elemento de la matriz es un puntero que apunta a una cadena o a un bloque de memoria.
- **Estructuras de datos dinámicas:** En estructuras como listas enlazadas, árboles, etc., se utilizan apuntadores para conectar los nodos. Un apuntador de apuntadores puede ser útil para modificar los enlaces entre los nodos.
- **Funciones que modifican punteros:** Algunas funciones pueden recibir un puntero a un puntero como argumento para poder modificar el puntero original dentro de la función.

2.6. Enviar y recibir apuntadores en una función

¿Por qué enviar y recibir punteros?

La principal razón para enviar y recibir punteros en una función es **modificar el valor de una variable original dentro de esa función**. Cuando pasamos una variable por valor, se crea una copia de esa variable dentro de la función, y cualquier cambio realizado en esa copia no afecta a la variable original. Sin embargo, al pasar un puntero, estamos pasando la dirección de memoria de la variable original, lo que significa que cualquier modificación realizada a través del puntero se reflejará en la variable original.

2.6. Enviar y recibir apuntadores en una función

¿Cómo funciona?

1. Pasar un puntero a una función:

- Se declara un parámetro de tipo puntero en la definición de la función.
- Al llamar a la función, se pasa la dirección de la variable usando el operador `&`.

2. Modificar el valor a través del puntero:

- Dentro de la función, se utiliza el operador de referenciación (`*`) para acceder al valor almacenado en la dirección de memoria apuntada por el puntero.
- Cualquier cambio realizado a este valor se refleja en la variable original.

Práctica

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

// Declaración de la función
void Imprime()
{
    cout << "Imprimiendo un mensaje" << endl;
}

int main()
{
    // Declaración del puntero a función
    void (*ptr_funcion)() = Imprime;

    // Llamada a la función a través del puntero
    ptr_funcion();
}
```

1. Descripción de punteros a funciones:
 - Un puntero a función se declara con la siguiente sintaxis: `tipo_retorno (*nombre_puntero)(parámetros)`.
 - `tipo_retorno` es el tipo de dato que la función devuelve.
 - `nombre_puntero` es el nombre del puntero a función.
 - `parámetros` son los argumentos que la función acepta.
2. Ejemplo de uso de punteros a funciones: Supongamos que tenemos una función llamada `imprime` que simplemente muestra un mensaje en la consola. Queremos crear un puntero a esta función y luego invocarla.

Práctica

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

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

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

```
void operacion(int a, int b, int (*pFunc)(int, int));
```

```
int main()
{
    int x = 10, y = 5;
    operacion(x, y, Suma); // Imprime: Resultado: 15
    operacion(x, y, Resta); // Imprime: Resultado: 5
    return 0;
}
```

Supongamos que tenemos una función operación que toma dos enteros y un puntero a función como argumentos. La función operación ejecutará la operación especificada por el puntero a función en los dos enteros.

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

```
int Resta(int a, int b) { return a - b; }
```

```
void operacion(int a, int b, int (*pFunc)(int, int))
{
    int resultado = pFunc(a, b);
    cout << "Resultado: " << resultado << endl;
}
```


Estructuras

En programación, una estructura (o struct en muchos lenguajes como C y C++) es una forma de agrupar variables de diferentes tipos bajo un mismo nombre. Es como crear un tipo de dato personalizado, adaptado a las necesidades específicas de tu programa.

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

```
struct Persona
{
    string nombre;
    int edad;
    float altura;
};
```

```
int
{
    Persona persona1;
    persona1.nombre = "Juan";
    persona1.edad = 30;
    persona1.altura = 1.75;
}
```

main()

Práctica

Objetivo:

El objetivo de este ejercicio es crear un programa en C++ que simula una tienda de armas en un juego de rol. El programa permite al jugador ver las armas disponibles, comprarlas con su oro y salir de la tienda.

Paso 1: Definir la estructura 'Arma'

1. Crear una estructura llamada `Arma` que contenga los siguientes miembros:
 - `nombre`: Un string para almacenar el nombre del arma.
 - `precio`: Un entero para almacenar el precio del arma en monedas de oro.
 - `daño`: Un entero para almacenar el daño que inflige el arma.

Paso 2: Definir las armas disponibles

1. Dentro de la función `main()`, declarar un array de tipo `Arma` llamado `armas`.
2. Inicializar el array `armas` con los datos de las armas disponibles. En este caso, el código proporciona tres armas: "Espada oxidada", "Escudo mágico" y "Arco élfico".
3. Calcular el número de armas disponibles utilizando la expresión `sizeof(armas) / sizeof(Arma)`. Almacenar este valor en la variable `cantidadArmas`.

Paso 3: Definir el oro inicial del jugador

1. Declarar una variable entera llamada `oroJugador` para almacenar el oro inicial del jugador.
2. Asignar un valor inicial al `oroJugador`, por ejemplo, 30.
3. Declarar un puntero `*oroActual` y asignar el valor de `oroJugador`

Paso 4: Bucle principal del juego

1. Implementar un bucle `while(true)` para representar el ciclo principal del juego.
2. Dentro del bucle, mostrar un mensaje de bienvenida a la tienda de armas y mostrar el oro actual del jugador.

Paso 5: Mostrar las armas disponibles

1. Almacenar el arreglo de armas en un puntero de tipo `Armas` llamado `armasEnVenta`
2. Recorrer `armasEnVenta` utilizando un bucle `for`.
3. Dentro del bucle, mostrar el número, nombre, precio y daño de cada arma.

Paso 6: Mostrar el menú de opciones

1. Mostrar un menú con las siguientes opciones:
 2. Comprar arma
 3. Salir
2. Solicitar al jugador que ingrese una opción utilizando `cin >> opcion`.

Paso 7: Procesar la opción seleccionada

1. Utilizar una instrucción `switch` para procesar la opción seleccionada por el jugador:
 - **Caso 1: Comprar arma:**
 - Mostrar un nuevo menú con las armas disponibles para comprar.
 - Solicitar al jugador que ingrese el número del arma que desea comprar.
 - Validar la entrada del jugador para asegurarse de que sea un número válido dentro del rango de armas disponibles.
 - Obtener el arma seleccionada del array `armas` utilizando el índice ingresado por el jugador menos 1 (ya que los índices de los arrays comienzan en 0).
 - Verificar si el jugador tiene suficiente oro para comprar el arma.
 - Si tiene suficiente oro, restar el precio del arma del `oroActual` y mostrar un mensaje de compra exitosa.
 - Si no tiene suficiente oro, mostrar un mensaje indicando que no tiene suficiente dinero.
 - **Caso 2: Salir:**
 - Mostrar un mensaje de despedida y finalizar el programa utilizando `return 0;`.
 - **Caso default:**
 - Mostrar un mensaje indicando que la opción ingresada es inválida.



03

Recursividad

- Definición de recursividad
- Encontrar caso base
- Gráficamente como funciona la recursividad
- La función factorial como operación recursiva
- Representación recursiva de los números de Fibonacci

3.1. Definición de Recursividad

Un concepto que siempre le cuesta bastante a los programadores que están empezando es el de recursión o **recursividad** (se puede decir de las dos maneras).

Aunque es un concepto que puede llegar a ser muy complejo, en esencia es muy sencillo.

¿Qué es la recursividad?

Imagina un espejo colocado frente a otro espejo. Cada espejo refleja la imagen del otro, creando una secuencia infinita de imágenes. La recursividad en programación funciona de manera similar. Una función recursiva es aquella que se llama a sí misma dentro de su propia definición.

¿Cómo funciona?

Una función recursiva debe cumplir dos condiciones esenciales:

1. **Caso base:** Es la condición que detiene la recursión. Cuando se alcanza este caso, la función deja de llamarse a sí misma y devuelve un resultado.
2. **Caso recursivo:** Es la llamada a la función misma, pero con un argumento que se acerca más al caso base.

Práctica

```
// Declaración de la función recursiva para calcular el factorial
long long factorial(int n);
```

```
int main() {
    int numero;

    cout << "Ingresa un número positivo: ";
    cin >> numero;

    if (numero < 0) {
        cout << "Error: El factorial de un número negativo no existe." << endl;
    } else {
        long long resultado = factorial(numero);
        cout << "El factorial de " << numero << " es: " << resultado << endl;
    }

    return 0;
}

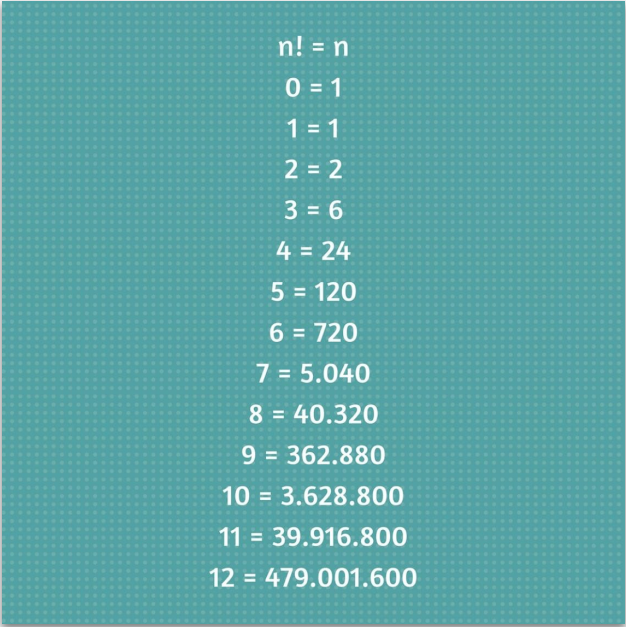
int factorial(int n) {
    if (n > 1) {
        return n * factorial(n - 1);
    } else {
        return 1;
    }
}
```

Recursividad: Es aquella función que se llama a ella misma y tiene un caso base y un caso general.

Ejercicio: Sacar el factorial de un número.

3!= 3*2!

El factorial de un número entero positivo se define como el producto de todos los números naturales anteriores o iguales a él.



n!	=	n
0	=	1
1	=	1
2	=	2
3	=	6
4	=	24
5	=	120
6	=	720
7	=	5.040
8	=	40.320
9	=	362.880
10	=	3.628.800
11	=	39.916.800
12	=	479.001.600

Práctica

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

// Declaración de funciones
// Función para verificar si un número es par
bool Par(int num);
// Función para verificar si un número es impar
bool Impar(int num);
```

```
int main()
{
    int numero = 5;

    if (Impar(numero))
    {
        cout << numero << " es impar." << endl;
    }
    else
    {
        cout << numero << " es par." << endl;
    }

    return 0;
}
```

```
// Función para verificar si un número es par
bool Par(int num)
{
    if (num == 0)
    {
        return true;
    }
    else
    {
        return Impar(num - 1);
    }
}
```

```
// Función para verificar si un número es impar
bool Impar(int num)
{
    if (num == 0)
    {
        return false;
    }
    else
    {
        return Par(num - 1);
    }
}
```


¿Por qué usar recursividad?

Solución elegante para problemas recursivos: Muchos problemas matemáticos y de programación tienen una definición recursiva natural.

Código más conciso: A menudo, el código recursivo es más corto y fácil de entender que una solución iterativa.

Estructuras de datos recursivas: La recursividad es fundamental para trabajar con estructuras de datos como árboles y listas enlazadas.

Desventajas de la recursividad

Mayor consumo de memoria: Cada llamada recursiva ocupa espacio en la pila.

Puede ser menos eficiente: En algunos casos, una solución iterativa puede ser más rápida.

Dificultad para depurar: Los errores en funciones recursivas pueden ser más difíciles de encontrar.

Cuándo usar recursividad

Problemas que se pueden dividir en subproblemas más pequeños del mismo tipo.

Estructuras de datos recursivas (árboles y grafos).

Cuando la solución recursiva es más clara y concisa que una iterativa.

3.3 Gráficamente cómo funciona la recursividad

Toda definición recursiva establece un proceso que debe venir dado por tres reglas.

Regla Base

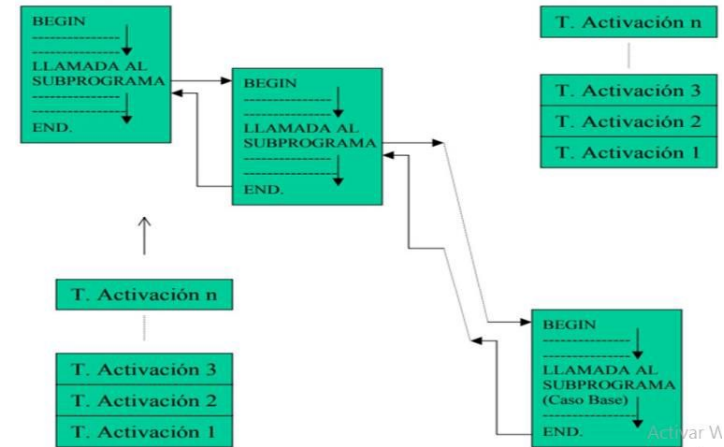
Es la que indica cuáles son los ejemplos o casos particulares que cumplen la definición. Es imprescindible para hacer una definición recursiva.

Regla de Exclusión

Es un conjunto de reglas que indica cuándo un objeto no se ajusta al concepto en términos de la regla base y la regla recursiva. Normalmente esta parte va implícita y es opcional.

Regla Recursiva

Realmente es un conjunto de reglas que aplicándolas establece cuándo un objeto responde a la definición, reglas en las que deben de aparecer de nuevo el concepto a definir.

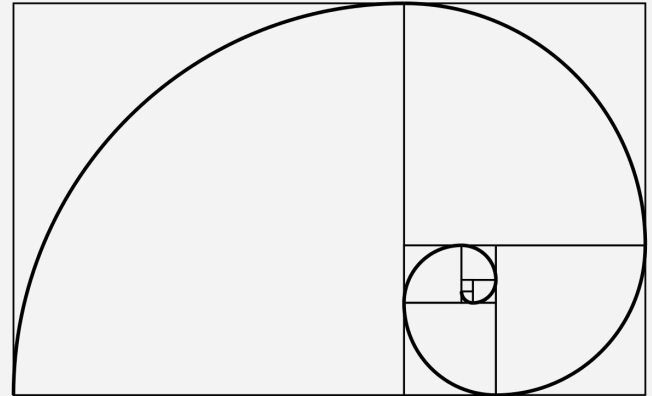


3.5. Representación recursiva de los números de Fibonacci

La serie de Fibonacci es una secuencia de números en la que cada número es la suma de los dos anteriores. Comienza con los números 0 y 1. Por ejemplo: 0, 1, 1, 2, 3, 5, 8, 13, ...

La Recursividad en la Serie de Fibonacci

La definición misma de la serie de Fibonacci se presta de manera natural a una solución recursiva. Podemos definir una función que calcule el n-ésimo número de Fibonacci de la siguiente manera.



Práctica

Realice una función recursiva para la serie Fibonacci

Nota: La serie de Fibonacci está formada por la secuencia de números:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34...

```
#include<iostream>
using namespace std;
int fibonacci(int n);
int main()
{
    int nElementos;
    //Pedimos un numero entero positivo
    do
    {
        cout<<"Digite el numero de elementos: ";
        cin>>nElementos;
    }while(nElementos <= 0);

    //Mandamos llamar a la función pero de forma iterativa para imprimir todos los elementos
    cout<<"Serie Fibonacci: ";
    for(int i=0;i<nElementos;i++){
        cout<<fibonacci(i)<<" , ";
    }

    return 0;
}
```

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

Práctica

OBJETIVO:

Simular el tiro parabólico de un balón de fútbol, considerando la potencia del tiro, el ángulo de lanzamiento y la gravedad, utilizando el lenguaje de programación C++.

NOTAS: En la siguiente práctica no hay variables definidas, son solo las instrucciones que debes seguir para realizar el ejercicio. Deben de pensar en qué **TIPO Y NOMBRE DE LAS VARIABLES**.

TIPO DE VARIABLE: int, char, string, etc.

Paso 1: Variables y constantes necesarias:

- Variable que almacene la potencia de tiro (entre 0 y 100).
- Variable que almacene el ángulo del tiro (en grados entre 0 y 90).
- Definir la variable de gravedad (9.81).
- Puntero que almacene la posición horizontal (x) del balón (inicialmente 0).
- Puntero que almacene la posición vertical del balón (inicialmente 0).
- Variable que almacena el tiempo transcurrido desde el lanzamiento del balón (inicialmente 0.1).

Práctica

Paso 2: Implementar la función **TrajectoryCalculation**:

Esta función recursiva calculará la trayectoria del balón para un intervalo de tiempo determinado.

- **parámetros:** puntero a la posición horizontal (x), puntero a la posición vertical (y), tiempo, potencia de tiro, ángulo y gravedad.
 - **variables:** Velocidad de desplazamiento en x y velocidad de desplazamiento en Y..
1. Calcular las componentes horizontales (velocityX) y verticales (velocityY) de la velocidad inicial utilizando las fórmulas:
 - a. Velocidad en x = potencia de tiro * $\cos(\text{ángulo} * M_PI / 180)$;
 - b. Velocidad en y = potencia de tiro * $\sin(\text{ángulo} * M_PI / 180)$;
 2. Actualizar la posición horizontal del balón utilizando la fórmula:
 - a. *posicion horizontal = velocidad en x * tiempo;
 3. Actualizar la posición vertical del balón utilizando la fórmula:
 - a. *posicion vertical = velocidad en y * tiempo - 0.5 * gravedad * tiempo * tiempo;
 4. Verificar si el balón ha llegado al suelo. Si es así, imprimir un mensaje indicando que el balón ha llegado al final del tiro y terminar la función.
 5. Si el balón no ha llegado al suelo, llamar recursivamente a la función **TrajectoryCalculation** para calcular la trayectoria para el siguiente paso de tiempo (time + 0.1).

Práctica

Paso 3: Implementar `main`:

Esta función recursiva calculará la trayectoria del balón para un intervalo de tiempo determinado.

1. Incluir la librería: `#include <math.h>`
2. Solicitar al usuario la potencia del tiro
3. Validar que la potencia esté entre 0 y 100.
4. Solicitar al usuario el ángulo del tiro.
5. Validar que el ángulo esté entre 0 y 90 grados..
6. Llamar a la función `TrajectoryCalculation` para calcular la trayectoria del balón.

Paso 4: NO OLVIDES COMENTAR TU CÓDIGO, SE TOMARÁ EN CUENTA LA COMPRENSIÓN DEL MISMO.



04

Estructuras

- Conocer las Estructuras
- Definición de estructura
- Ejemplos de estructura
- Typedef
- Operador punto
- Operador flecha

4.0. Estructuras

¿Qué es una estructura?

En programación, una estructura (o struct en muchos lenguajes como C y C++) es una forma de agrupar variables de diferentes tipos bajo un mismo nombre. Es como crear un tipo de dato personalizado, adaptado a las necesidades específicas de tu programa.

¿Para qué sirven las estructuras?

Imagina que quieres representar a una persona. Una persona tiene nombre, edad, altura, etc. En lugar de declarar cada una de estas variables por separado, puedes crear una estructura llamada "Persona" y agrupar todas estas propiedades dentro de ella.

4.0. Estructuras

¿Por qué usar estructuras?

- **Organización del código:** Agrupar datos relacionados mejora la legibilidad y mantenibilidad del código.
- **Creación de tipos de datos personalizados:** Puedes definir estructuras para representar cualquier concepto que requiera múltiples datos.
- **Paso de parámetros a funciones:** Puedes pasar estructuras a funciones como un solo argumento.
- **Retorno de valores de funciones:** Algunas funciones pueden retornar estructuras como resultado.

4.1. Definición de estructura

Una estructura es un **tipo de dato compuesto que permite almacenar un conjunto de datos de diferente tipo**.

Los datos que contiene una estructura pueden ser de tipo simple (números enteros o de coma flotante etc.) o a su vez de tipo compuesto (estructuras, listas, etc.).

struct

es una palabra reservada de C que indica que los **elementos que vienen agrupados** a continuación entre llaves componen una estructura

Es una colección de uno o más tipos de elementos denominados campos, cada uno de los cuales puede ser un tipo de dato diferente.

nombre_estructura:

identifica el tipo de dato que se describe y del cual se podrán declarar variables. Se especifica entre corchetes para indicar su opcionalidad

miembro1, "miembro2..."

Son los elementos que componen la estructura de datos, deben ser precedidos por el tipo_dato al cual pertenecen

```
struct Nivel {  
    string nombre;  
    string fondo;  
    int num_enemigos;  
    int tiempo;  
    bool jefe;  
};
```

4.2. Ejemplos de estructura

Consideremos la información de una fecha. Una fecha consiste de el **día, el mes, el año** y posiblemente el día en el año y el nombre del mes. Declaramos toda esa información en una estructura del siguiente modo:

```
struct fecha {  
    int dia;  
    int mes;  
    int anio;  
    int dia_del_anio;  
    char nombre_mes[9];  
};
```

Un ejemplo de estructura que incluye otras estructuras es la siguiente estructura **persona** que incluye la estructura fecha:

```
struct persona {  
    char nombre[tamano_nombre];  
    char direccion[tamano_dir];  
    long codigo_postal;  
    long seguridad_social;  
    double salario;  
    fecha cumpleaños;  
    fecha contrato;  
};
```

Supongamos que se desea almacenar los datos de una colección de discos compactos (CD) de música. Estos datos pueden ser:

- Título
- Artista
- Número de canciones
- Precio
- Fecha de compra

```
struct coleccion_CD{  
    string titulo;  
    string artista;  
    int num_canciones;  
    float precio;  
};
```

¿Cómo se declara una estructura en C++?

```
struct <nombreDeLaEstructura>{  
    <tipoDeDato>    <nombreDeCampo>  
    <tipoDeDato>    <nombreDeCampo>  
    <tipoDeDato>    <nombreDeCampo>  
    <tipoDeDato>    <nombreDeCampo>  
    <tipoDeDato>    <nombreDeCampo>  
    ...  
};
```

```
struct coleccion_CD{  
    string titulo;  
    string artista;  
    int num_canciones;  
    float precio;  
};
```

```
int main(){  
    struct coleccion_CD CD1,CD2,CD3;
```


Práctica

En este ejemplo, se define una estructura llamada `Nivel` que almacena los datos de un nivel de un juego de plataformas, como el nombre, el fondo, el número de enemigos, el tiempo límite, etc.

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

struct Nivel
{
    string nombre; // el nombre del nivel
    string fondo; // el nombre del archivo de imagen que contiene el fondo del
nivel
    int num_enemigos; // el número de enemigos que hay en el nivel
    int tiempo; // el tiempo límite en segundos para completar el nivel
    bool jefe; // un valor booleano que indica si el nivel tiene un jefe final o no
};

void MostrarNivel(Nivel n)
{
    cout << "Nombre del nivel: " << n.nombre << endl;
    cout << "Fondo del nivel: " << n.fondo << endl;
    cout << "Número de enemigos: " << n.num_enemigos << endl;
    cout << "Tiempo límite: " << n.tiempo << " segundos" << endl;
    cout << "Jefe final: " << (n.jefe ? "Sí" : "No") << endl;
}
```

```
int main()
{
    Nivel n1 = {"Nivel 1", "bosque.png", 10, 60, false};
    MostrarNivel(n1);
}
```

4.3. Typedef

Un Aliado para la Claridad en C++

¿Qué es `typedef`?

En C++, `typedef` es una palabra clave que se utiliza para crear un nuevo nombre para un tipo de dato existente. Esencialmente, es un alias o sinónimo para un tipo de dato.

¿Por qué usar `typedef`?

El principal objetivo de `typedef` es mejorar la legibilidad y la portabilidad de tu código. Aquí hay algunas razones comunes para usarlo:

1. **Creación de tipos personalizados:** Puedes crear nombres más descriptivos para tipos de datos estándar.
2. **Simplificación de declaraciones:** Puedes usar `typedef` para simplificar declaraciones complejas.
3. **Portabilidad:** Si estás trabajando con diferentes plataformas o compiladores, `typedef` puede ayudarte a garantizar la portabilidad de tu código. Puedes definir tipos de datos que sean consistentes en diferentes entornos.

4.3. Typedef

¿Qué es `typedef`?

En C++, `typedef` es una palabra clave que se utiliza para crear un nuevo nombre para un tipo de dato existente. Esto puede ser útil para mejorar la legibilidad y la reutilización del código.

¿Cómo se usa `typedef` con estructuras?

Cuando se combina con estructuras, `typedef` permite crear un alias o sinónimo para el tipo de estructura. Esto significa que puedes usar el nuevo nombre en lugar del nombre original de la estructura.

¿Por qué usar `typedef` con estructuras?

- **Mejora la legibilidad:** Un nombre descriptivo para un tipo de estructura puede hacer que el código sea más fácil de entender.
- **Facilita la reutilización:** Si necesitas usar la misma estructura en diferentes partes de tu código, puedes definir un alias y usarlo en todos los lugares.
- **Simplifica la declaración de variables:** Puedes declarar variables de tipo estructura usando el alias en lugar del nombre completo de la estructura.

```
struct Persona {  
    string nombre;  
    int edad;  
    float altura;  
};  
  
typedef Persona PersonaType;  
  
int main() {  
    PersonaType persona1;  
    // ...  
}
```

4.4. Operador punto

¿Qué es el operador punto (.) en estructuras?

En programación, especialmente en lenguajes como C y C++, el operador punto (.) se utiliza para acceder a los miembros (variables o funciones) de una estructura. Es decir, nos permite seleccionar y manipular los datos que se encuentran dentro de una estructura.

¿Cómo funciona?

Imagina una estructura como una caja que contiene varias cosas. El operador punto es como una llave que te permite abrir la caja y tomar un elemento específico.

```
struct Persona {  
    string nombre;  
    int edad;  
    float altura;  
};  
  
int main() {  
    Persona persona1;  
    persona1.nombre = "Juan";  
    persona1.edad = 30;  
    persona1.altura = 1.75;  
  
    cout << "El nombre de la persona es: " << persona1.nombre << endl;  
}
```

4.5. Operador flecha

¿Qué es el operador flecha (->)?

En programación, especialmente en lenguajes como C y C++, el operador flecha (->) se utiliza para acceder a los miembros de una estructura cuando se tiene un **puntero** a esa estructura. Es decir, nos permite manipular los datos de una estructura a través de su dirección de memoria.

¿Por qué usar el operador flecha?

- **Punteros a estructuras:** Cuando trabajas con punteros a estructuras, el operador flecha es esencial para acceder a los miembros.
- **Dinámica de memoria:** En muchas estructuras de datos dinámicas (listas enlazadas, árboles), los elementos se almacenan en la memoria utilizando punteros. El operador flecha te permite manipular estos elementos.
- **Funciones que retornan punteros:** Si una función retorna un puntero a una estructura, debes usar el operador flecha para acceder a los miembros del objeto al que apunta.

Operador	Uso
. (punto)	Se utiliza para acceder a los miembros de una estructura directamente . Se aplica cuando tienes una variable de tipo estructura.
-> (flecha)	Se utiliza para acceder a los miembros de una estructura a través de un puntero a esa estructura.

Práctica

1. Incluir las librerías necesarias, como `iostream` y `string`.
2. Definir la estructura `Personaje` con los campos `nombre`, `nivel`, `experiencia`, `salud` y `ataque`.
3. Definir la función `subirNivel` que reciba un puntero a un `Personaje` y modifique sus campos según la fórmula.
 - a. La fórmula para subir de nivel es la siguiente: el nivel se incrementa en uno, la experiencia se pone a cero, la salud se multiplica por 1.1, y el ataque se suma 5.
4. Crear una variable de tipo `Personaje` e inicializarla con algunos valores.
5. Mostrar en pantalla los datos del personaje antes de subir de nivel.
6. Llamar a la función `subirNivel` pasando la dirección de la variable del personaje.
7. Mostrar en pantalla los datos del personaje después de subir de nivel.

```
Nombre: Mario
Nivel: 1
Experiencia: 50
Salud: 100
Ataque: 10
-----
Nombre: Mario
Nivel: 2
Experiencia: 0
Salud: 110
Ataque: 15
```

Práctica

1. Crea una estructura que represente un objeto de un videojuego, con los atributos que consideres necesarios (por ejemplo, nombre, tipo, valor, peso, etc.).
2. Usa typedef para crear un alias para esa estructura, por ejemplo, objeto.
3. Crea una función que reciba dos objetos y que devuelva el objeto más valioso (el que tenga mayor valor o menor peso, según el criterio que elijas).

```
El objeto mas valioso entre la espada y el escudo es: Espada  
El objeto mas valioso entre la pocion y el anillo es: Anillo  
El objeto mas valioso entre la espada y el anillo es: Espada
```

Práctica

1. Crea una estructura llamada “Enemigo”:
 - a. La clase debe tener los siguientes atributos:
 - i. nombre (cadena de caracteres): El nombre del enemigo.
 - ii. nivel (entero): El nivel del enemigo.
 - iii. salud (flotante): La salud del enemigo.
2. Crea un método llamado “Atacar”:
 - a. Este método debe recibir un valor numérico que represente el daño infligido al enemigo y una variable de tipo enemigo.
 - b. Resta el valor del daño a la salud del enemigo.
 - c. Si la salud del enemigo llega a cero o menos, muestra un mensaje indicando que el enemigo ha sido derrotado.
3. En el programa principal (función main):
 - a. Crea un objeto de tipo “Enemigo” llamado jefeFinal.
 - b. Asigna valores iniciales al objeto (por ejemplo, nombre: “Dragón”, nivel: 10, salud: 200).
 - c. Llama al método Atacar con un valor de daño (por ejemplo, 50).
 - d. Muestra la salud actual del enemigo.
 - e. Si la salud es menor o igual a cero, muestra un mensaje de victoria.

Práctica

En este ejemplo, creamos una clase Personaje que tiene atributos como nombre, nivel y salud. El método MostrarInfo() utiliza el operador flecha para acceder a los miembros de la instancia del personaje.

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

```
// Estructura para representar un personaje de videojuego
struct Personaje
```

```
{
    string nombre;
    int nivel;
    float salud;
};
```

```
// Método para mostrar información del personaje
```

```
void MostrarInfo(Personaje *p)
{
    cout << "Nombre: " << p->nombre << "\n";
    cout << "Nivel: " << p->nivel << "\n";
    cout << "Salud: " << p->salud << "\n";
}
```

```
int main()
```

```
{
    // Crear un objeto de tipo Personaje
    Personaje jugador;
    jugador.nombre = "Heroe";
    jugador.nivel = 5;
    jugador.salud = 100.0f;
```

```
    cout << "Información del jugador:\n";
    MostrarInfo(&jugador);
```

```
}
```

Práctica

1. Define una estructura llamada Jugador que tenga dos miembros: nombre de tipo string y monedas de tipo int.
2. Crea una función llamada AumentarMonedas que reciba un puntero a una estructura Jugador y un número entero n. La función debe aumentar el valor de monedas del jugador en n usando el operador flecha.
3. Crea una función llamada MostrarJugador que reciba un puntero a una estructura Jugador y muestre su nombre y sus monedas en la pantalla usando el operador flecha.
4. En la función principal (main), crea una variable de tipo Jugador e inicializa sus miembros con valores de tu elección. Luego, crea un puntero de tipo Jugador que apunte a la dirección de memoria de la variable. Finalmente, llama a las funciones AumentarMonedas y MostrarJugador pasando el puntero como argumento y observa el resultado. Debes mostrar las monedas antes de aumentar su valor y después.