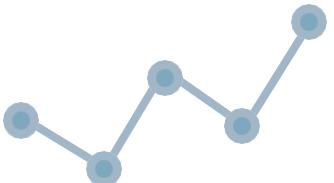


LENGUAJES DE PROGRAMACIÓN I

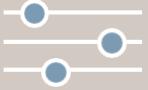




BIENVENIDA

Bienvenido(a) a la asignatura *Lenguajes de Programación I*, con la cual iniciarás tus estudios del lenguaje C++. Este lenguaje es muy importante porque se utiliza en áreas en donde el alto desempeño es una cuestión de gran relevancia. En otras palabras, C++ es una de las mejores opciones para aplicaciones de software rápidas.

Actualmente, C++ es un lenguaje que se utiliza mucho en los motores de videojuegos o navegadores. Por tanto, aprender este lenguaje es la puerta de entrada hacia varias industrias. Finalmente, al aprender este lenguaje se te hará más sencillo moverte a otros similares.



LIBROS RECOMENDADOS

- Pandey, H. M. (2015). *Object-Oriented Programming C++ Simplified*. University Science Press.
- Horton, I. & van Weert, P. (2018). *Beginning C++17: From Novice to Professional*. Apress.



1
UNIDAD

FUNDAMENTOS DEL LENGUAJE



1.1

ENTORNO DE DESARROLLO



TEMARIO



1.2

PALABRAS RESERVADAS Y SINTAXIS





INTRODUCCIÓN

La asignatura *Lenguajes de Programación I* tiene como objetivo el análisis del paradigma estructurado de programación, así como el conocimiento de la sintaxis de un lenguaje orientado a objetos (C++). Esto con la finalidad de solucionar problemas de la vida real, tanto administrativos como informáticos.

En esta primera unidad aprenderás más acerca de los entornos de desarrollo. Por otra parte, conocerás las palabras reservadas y la sintaxis de este lenguaje.

COMPETENCIAS A DESARROLLAR



The background features a hand holding a stylus, pointing at a smartphone screen. The screen displays a 3D grid interface with several data points labeled with values like 212.4 and 50. The overall theme is technology and data analysis.



El alumno será capaz de comprender el proceso de desarrollo de un programa en C++.



El alumno será capaz de aplicar la sintaxis de los operadores básicos del lenguaje C++.

ENTORNO DE DESARROLLO

C++ fue diseñado a mediados de los años 80 por Bjarne Stroustrup, con la intención de extender el exitoso lenguaje de programación C, con mecanismos que permitieran la manipulación de objetos.

En la mayoría de los casos, la sintaxis de instrucción de C++ es idéntica a la de ANSI C. La diferencia principal es que en C solo se permiten declaraciones al principio de un bloque. Por su parte, C++ agrega la instrucción de declaración, que elimina eficazmente esta restricción.



ENTORNO DE DESARROLLO

Los Entornos de Desarrollo Integrados (IDE) para C++ hacen uso de un concepto de **proyecto**.

Un proyecto en un IDE cualquiera es un *contenedor* global. En este, podemos incluir o crear todos los archivos necesarios que tengan relación con una aplicación de software que estemos desarrollando: clases, interfaces, archivos de texto, imágenes, paquetes, entre otros.



ENTORNO DE DESARROLLO

A continuación, verás una lista de IDE que podemos utilizar para programar en C++:

EN WINDOWS PUEDES USAR:	EN LINUX PUEDES USAR:
Visual C++	Gedit
Visual Studio	Geany
Notepad++	Kate
DevC++	KDevelop
Code::Blocks	Eclipse
Eclipse	Code::Blocks
Ultimate++	Ultimate++
Kdevelop	CodeLite



```
import java.util.ArrayList; import java.io.File;
import java.util.Scanner; import java.io.IOException;
import java.io.File;
import java.io.IOException; import java.util.Arrays;
import java.util.Arrays;

public class AirlineProblem {ss AirlineProblem {
    public static void main(String[] args){
        Scanner scannerToReadFile = null;id main(String[] args){
        try{
            scannerToReadFile = new Scanner(new File("airlines.txt")); id main(String[] args){
        } catch(NoSuchElementException e){try{
            System.out.println("Could not connect to file airlines.txt"); System.out.println("Scanner(new File(")
        } catch(NoSuchElementException e){System.out.println("Could not connect to file airlines.txt"); System.out.println("Scanner(new File(")
        } if(scannerToReadFile != null){catch(NoSuchElementException e){
            ArrayList<Airline> airlinesPartnersNetwork = new ArrayList<Airline>;
            Airline newline; System.out.println("Could not connect to file
            String lineFromFile; System.exit(0);
            newline = new Airline(scannerToReadFile.nextLine());
            airlinesPartnersNetwork.add(newline);
            System.out.println("String lineFromFile;
            System.out.println(airlinesPartnersNetwork);
            Scanner keyboard = new Scanner(System.in);
            String[] airlineNames;
```

ENTORNO DE DESARROLLO

Los IDE anteriores te servirán para lo siguiente:

○ **Escribir el código fuente.** Mismo que tendrá una extensión .cpp.

○ **Compilar el código fuente.** Esto es, traducirlo a un lenguaje comprensible por el ordenador. Los archivos resultantes se denominan archivos objeto. Se identifican mediante la extensión .ou .obj.

ENTORNO DE DESARROLLO

○ **Enlazar los archivos compilados.** Un programa puede estar repartido entre distintos archivos. Cada uno de los cuales se habrá compilado independientemente. Por ello, antes de ejecutar el programa, es necesario unir o ligar los archivos objeto (.o u .obj) que lo componen. Es se hace con un enlazador (*linker*). El resultado final es el programa ejecutable, cuya extensión suele ser .exe.



VIDEO

- Te invitamos a ver los siguientes videos:



PALABRAS RESERVADAS Y SINTAXIS

Las palabras reservadas son identificadores predefinidos que tienen significados especiales. Estos no pueden utilizarse como identificadores creados por el usuario en los programas. Las palabras reservadas de C++ pueden agruparse en 3 grupos.

El primer grupo contiene palabras del lenguaje C que también se usan en C++:

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

PALABRAS RESERVADAS Y SINTAXIS

El segundo grupo contiene palabras que no provienen de C. Por tanto, solo son utilizadas en C++:

asm	dynamic_cast	class	reinterpret_cast	try	bool
new	static_cast	typeid	catch	false	operator
typename	namespace	friend	private	this	using
inline	public	throw	const_cast	delete	mutable
true	wchar_t	explicit	template	virtual	protected

PALABRAS RESERVADAS Y SINTAXIS

El tercer grupo consta de las palabras reservadas que han sido añadidas como alternativas para algunos operadores de C++. Estas palabras ayudan a que los programas sean más fáciles de escribir y leer:

and	dynamic_cast	compl	not_eq
not_eq	not_eq	xor_eq	and_eq
bitor	not	or	xor



PALABRAS RESERVADAS Y SINTAXIS

Los **comentarios** son líneas de código que no son tomadas en cuenta por el compilador en el momento de ejecutar nuestra aplicación. Por lo tanto, no están sujetas a restricciones de sintaxis ni nada similar.

El uso principal de las líneas de comentario es mantener un orden y hacer más entendible el código. Esto es especialmente útil cuando este es leído por alguien distinto a los programadores originales.

PALABRAS RESERVADAS Y SINTAXIS

Existen **dos tipos de comentarios** en el lenguaje de programación C++:

- **Comentarios de una sola línea.** Pueden ser colocados en cualquier parte y comienzan con un doble slash “//”.

Ejemplo: int i=0;
//Declaración de variable

- **Comentarios multilínea.** Van encerrados entre “/*” y “*/”. Estos comentarios son similares a los anteriores, pero deben tener un comienzo y un final:

Ejemplo: float b;
/*Esta sentencia define el tipo de dato de la variable*/



PALABRAS RESERVADAS Y SINTAXIS

Un **tipo de dato** define un dominio de valores, así como las operaciones que se pueden realizar con estos. En C++ se dispone de unos cuantos tipos de datos predefinidos (simples). Sin embargo, se permite al programador crear otro tipo de datos.

Cabe mencionar que C++ es un lenguaje orientado a objetos y posee una enorme cantidad de librerías o bibliotecas que podemos usar. Estas nos proporcionan tipos de datos adicionales; no obstante, estos tipos de datos son más complejos.

PALABRAS RESERVADAS Y SINTAXIS

La siguiente tabla indica los **tipos de datos sencillos**:

TIPO	DESCRIPCIÓN	TAMAÑO	DOMINIO
Int	Números enteros	2 bytes (16 bits)	Son todos los números enteros entre los valores -32.768 y 32.767
Float	Números reales	4 bytes	Son todos los números reales que contienen una coma decimal comprendidos entre los valores: $3.4 \times [10]^{(-38)}$ y $3.4 \times [10]^{38}$
Double	Números reales más grandes que float	8 bytes	Son todos los números reales que contienen una coma decimal comprendidos entre los valores: $1.7 \times [10]^{(-308)}$ y $1.7 \times [10]^{308}$
Bool	Valores lógicos	1 bytes	Dos únicos valores : {true, false}
Char	Caracteres y cualquier cantidad de 8 bits	1 bytes	Dígitos, letras mayúsculas, letras minúsculas y signos de puntuación.
Void	El tipo vacío. Sirve para indicar que una función no devuelve valores.		

PALABRAS RESERVADAS Y SINTAXIS

Los tipos de datos son importantes para poder declarar variables de manera correcta.

Una **variable** es un **espacio reservado en el ordenador** para contener valores que pueden cambiar durante la ejecución de un programa. Los tipos determinan cómo se manipulará la información contenida en esas variables.

La sintaxis para una variable es la siguiente:

[modificadores] [tipo de dato] [nombre de la variable] [=] [valor];

PALABRAS RESERVADAS Y SINTAXIS

El siguiente es un ejemplo de cómo asignar valores a variables:

```
#include <iostream>
using namespace std;
Int main()
{
    char x ='a'; //Declaramos y asignamos en la misma linea
    int num; //Declaramos el entero en una linea
    num = 5; //Le asignamos una valor en otra linea
    int num2 = 8; //Asignación y declaración al tiempo
    float numero; //Un número decimal
    numero = 3.5; //Le asignamos un valor decimal
```

PALABRAS RESERVADAS Y SINTAXIS

```
float res = numero + num2; //Sumamos dos variables y las  
asignamos a res//3.5 + 8 = 11.5  
res = res + num; //Al valor actual de res le sumamos el valor  
de num //11.5 + 5 = 16.5  
bool valor = false; //Variable booleana  
valor = true; //Puede ser true o false  
res = res*2; //Duplicamos el valor de res 16.5*2 = 33  
cout << res << endl; //Mostramos el valor de res por pantalla  
return 0;  
}
```

PALABRAS RESERVADAS Y SINTAXIS

Las **constantes** son muy útiles para especificar el tamaño de un vector, así como para alguna variable de tamaño específico. Las constantes se declaran después de las librerías y antes de las funciones.

En C++ se pueden **definir constantes de dos maneras:**

- Con el comando: #define nombre_constante valor
- Con la palabra clave: const



PALABRAS RESERVADAS Y SINTAXIS

La declaración de constantes utilizando **#define** debe realizarse después de **#include** y antes de declarar funciones. Por ejemplo:

```
#include <iostream>
using namespace std;
#define PI 3.1416; //Definimos una constante de Pi
int main()
{
    cout << "Mostrando el valor de PI:" << PI;
    return 0;
}
```

PALABRAS RESERVADAS Y SINTAXIS

La instrucción *const* nos permite declarar constantes en el interior del main. Por ejemplo:

```
#include <iostream>
using namespace std;
int main()
{
    const float PI = 3.1416;
    //Definimos una constante llamada PI
    cout << "Mostrando el valor de PI:" << PI << endl;
    return 0;
}
```



PALABRAS RESERVADAS Y SINTAXIS

Un **operador** es un elemento del programa que se aplica a uno o a varios operadores en una expresión o instrucción.

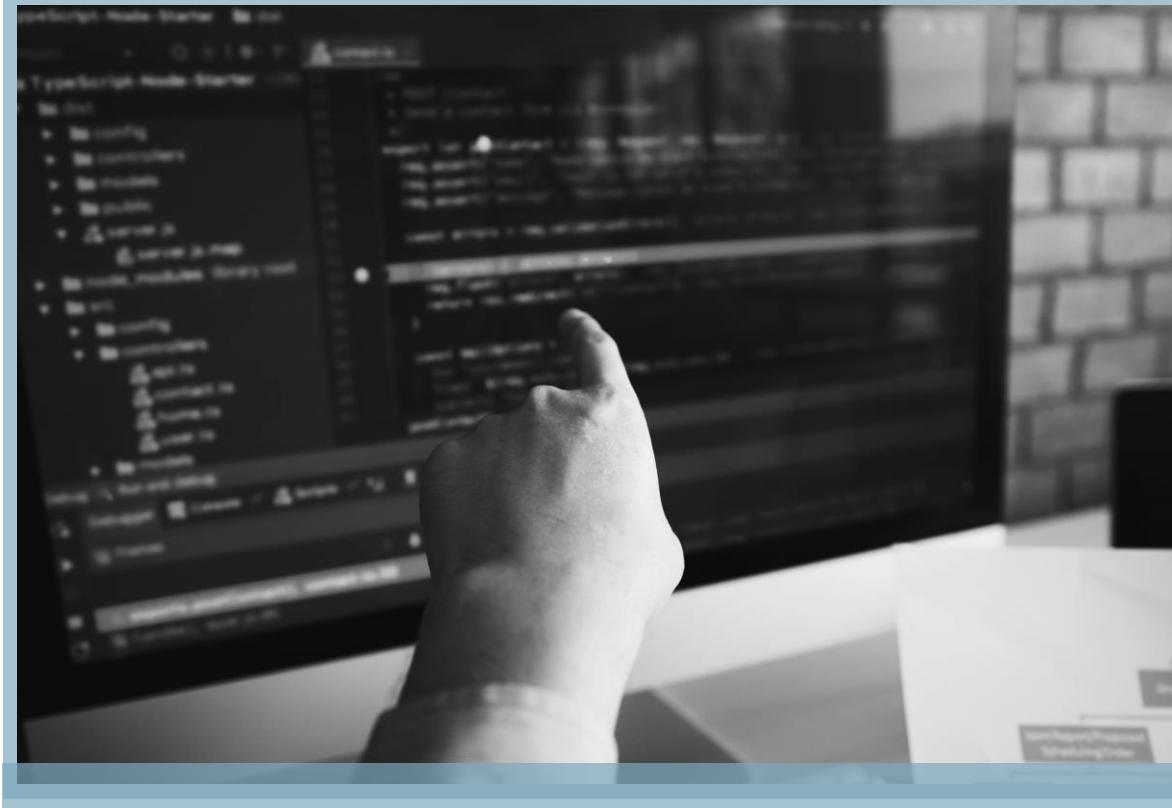
Los operadores que toman un operando (como el operador de incremento “`++`” o “`new`”) se conocen como operadores **unarios**.

Los operadores que toman dos operandos (como los aritméticos “`+`”, “`-`”, “`*`”, “`/`”) se conocen como **binarios**.

PALABRAS RESERVADAS Y SINTAXIS

Existen **tres tipos de operadores:**

- Aritméticos,
- Relacionales y
- Lógicos



PALABRAS RESERVADAS Y SINTAXIS

Los **operadores aritméticos** se usan para realizar cálculos de aritmética de números reales y aritmética de punteros. Son los siguientes:

OPERADOR	NOMBRE	EJEMPLO	DESCRIPCIÓN
+	Suma	5+6	Suma de dos números.
-	Sustracción	7-9	Resta de dos números.
*	Multiplicación	6*3	Multiplicación de dos números.
/	División	4/8	División de dos números.
%	Módulo: el resto después de la división	7%2	Devuelve el resto de dividir ambos números, en este ejemplo el resultado es 1.
++	incremento	a++	Suma 1 al contenido de una variable.
--	Decremento	a--	Resta 1 al contenido de una variable.
-	Invierte el signo de una operación	a-	Invierte el signo de una operación.

PALABRAS RESERVADAS Y SINTAXIS

Los **operadores relacionales** son símbolos que se usan para comprobar dos valores. Si el resultado de la comparación es correcto, la expresión considerada es verdadera; en caso contrario, es falsa:

OPERADOR	NOMBRE	EJEMPLO	DESCRIPCIÓN
<	Menor que	$a < b$	a es menor que b .
>	Mayor que	$a > b$	a es mayor que b .
==	Igual a	$a == b$	a es igual a b .
!=	No igual a	$a != b$	a no es igual a b .
<=	Menor que o igual a	$a <= b$	a es menor que o igual a b .
>=	Mayor que o igual a	$a >= b$	a es mayor que o igual a b .

PALABRAS RESERVADAS Y SINTAXIS

Los **operadores lógicos** son empleados, generalmente, con valores lógicos. Estos operadores devuelven un valor booleano:

OPERADOR	NOMBRE	EJEMPLO	DESCRIPCIÓN
<code>&&</code>	AND	A&&B	Si ambos son verdaderos se obtiene verdadero (true).
<code> </code>	OR	A//B	Verdadero si alguno es verdadero.
<code>!</code>	NOT	!A	Negación de A.

PALABRAS RESERVADAS Y SINTAXIS

Si los operadores “++” y “– –” están de prefijos, la operación de incremento se efectúa antes de la operación de asignación.

Si los operadores “++” y “– –” están de sufijos, la asignación se efectúa en primer lugar, y el incremento o decremento a continuación.

EJEMPLO DEL OPERADOR DE INCREMENTACIÓN

```
int a=1, b;  
b=++a;
```

¿Cuál es el valor de a y de b?

```
int a=1, b;  
b=a++ // b vale 1 y a vale 2
```

PALABRAS RESERVADAS Y SINTAXIS

Las **sentencias** son unidades completas, ejecutables en sí mismas, e incorporan expresiones aritméticas, lógicas o generales como componentes. Existen **tres tipos**:

○ **Sentencias simples.** Son una expresión terminada con un carácter “;”. Un caso típico de este tipo de sentencia son las declaraciones o las sentencias aritméticas. Ejemplo: *float real;*

```
espacio = espacio_inicial + velocidad * tiempo;
```

○ **Sentencias nulas o vacías.** En algunas ocasiones, es necesario introducir en el programa una sentencia que ocupe un lugar, pero que no realice ninguna tarea. A este tipo de sentencia se le conoce como vacía. Consta de un simple carácter “;”.

PALABRAS RESERVADAS Y SINTAXIS

- **Sentencias compuestas o bloques.** A veces, es necesario poner varias sentencias en donde debería haber solo una. Una sentencia compuesta es un conjunto de declaraciones y sentencias agrupadas dentro de llaves “{...}”. También son conocidas como **bloques**. Un bloque puede incluir sentencias simples y compuestas. Ejemplo:

```
{  
int i=1, j=3, k;  
double masa;  
masa=3.0;  
k=y+k;  
}
```

PALABRAS RESERVADAS Y SINTAXIS

Para poder sumar dos variables hace falta que ambas sean del mismo tipo. Si una de estas es de tipo *int* y la otra de tipo *float*, la primera se convierte en *float* (es decir, la variable del tipo de menor rango se convierte al tipo de mayor rango), antes de realizar la operación. A esta conversión automática e implícita de tipo se le conoce como **promoción**. Los rangos de las variables de mayor a menor se ordenan del siguiente modo:

```
long double > double > float > unsigned long > long > unsigned  
int > int unsigned short > short > char
```

PALABRAS RESERVADAS Y SINTAXIS

El **casting** es una conversión de tipo. Esta conversión es forzada por el programador. Para ello, basta preceder la constante, variable o expresión que se desea convertir por el tipo al que se desea convertir, encerrado entre paréntesis. Ejemplo:

```
K = (int) 2.5 + (int) masa;
```

El lenguaje C++ dispone de otra conversión explícita de tipo con una notación similar a la de las funciones. Para ello, se utiliza el nombre del tipo al que se desea convertir, seguido del valor a convertir entre paréntesis. Por ejemplo:

```
y = double(25);
doublex = 5;
return int(x/y);
```

FORO 1

○ Entorno de trabajo.

Participa en el foro enviando imágenes que demuestren que ya tienes acceso a las siguientes herramientas en su versión de prueba:

- Eclipse
- NetBeans
- Visual Studio

Presiona el botón para participar en el foro.



Actualmente, C++ es uno de los mejores lenguajes en el uso eficiente de los recursos. Lo mejor de todo, es que esta cualidad no se ve disminuida en proyectos de una complejidad elevada.

En programación existen diversos conceptos fundamentales muy importantes, a saber: datos, compiladores, depuradores, entre otros. Una particularidad de C++ es la posibilidad de redefinir los operadores (sobrecarga de operadores), y poder crear nuevos tipos que se comporten como tipos fundamentales.

CONCLUSIÓN



¡FELICIDADES!

Acabas de concluir la primera unidad de tu curso *Lenguajes de Programación I*. Te invitamos a finalizar este esfuerzo realizando el examen parcial correspondiente. Para ello, debes regresar a la pantalla principal y dar clic en *Presentar examen*.

2

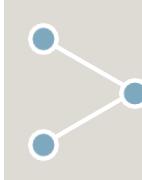
UNIDAD

ESTRUCTURAS DE CONTROL

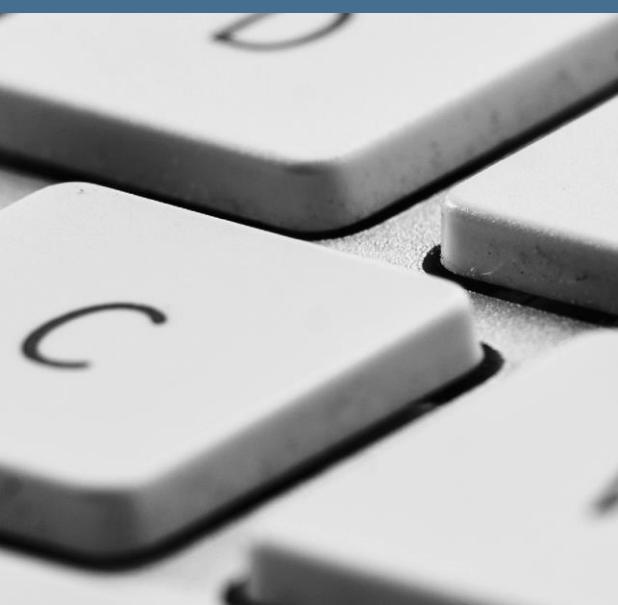


2.1

ESTRUCTURAS
REPETITIVAS



TEMARIO



2.2

ESTRUCTURAS
SELECTIVAS





INTRODUCCIÓN

En esta segunda unidad aprenderás más acerca de las estructuras de control. Estas estructuras son esenciales al momento de programar en C++. En particular, verás las estructuras de control repetitivas y las de decisión. Mismas que te permitirán dar un flujo a la transmisión y transformación de datos en tus programas.

COMPETENCIAS A DESARROLLAR



El alumno podrá implementar estructuras de control repetitivas.



El alumno podrá implementar estructuras de control selectivas.

ESTRUCTURAS REPETITIVAS

El proceso de resolución de un problema con una computadora conduce a la escritura de un programa y a su posterior ejecución. Aunque el proceso de diseño de programas es, esencialmente, creativo, se pueden considerar una serie de pasos o fases comunes. Estas, generalmente, son seguidas por todos los programadores.

Existen muchas **metodologías** para construir programas. En esta ocasión, aplicaremos una sencilla, que es adecuada para la construcción de programas. Sus pasos son los siguientes:



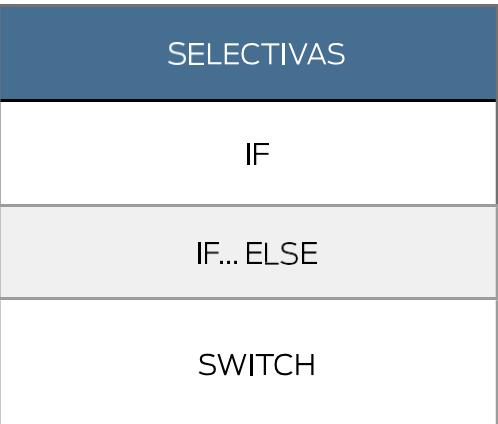


ESTRUCTURAS REPETITIVAS

- **Análisis.** Entender profundamente cuál es el problema que se busca resolver. Se debe incluir el contexto en el cual se usará.
- **Diseño.** Este es el cómo vamos a resolver este problema. Se deberá documentar la especificación por escrito. De esta manera, las decisiones se tomarán usando como dato de la realidad el contexto en el cual se aplicará la solución.
- **Implementación.** Traducir a un lenguaje de programación el diseño del punto anterior.

ESTRUCTURAS REPETITIVAS

Las estructuras de control son sentencias que bifurcan la ejecución del programa. En general, cuando se llega a este punto, es posible que deba ejecutar un grupo de instrucciones u otros; o bien, repetir un grupo de instrucciones determinado número de veces. Estas pueden ser:



ESTRUCTURAS REPETITIVAS

Este tipo de estructuras permite la repetición de un grupo de instrucciones mientras que una condición se cumpla. En C++ las **tres principales estructuras de control** son las siguientes:

- **While.** Las sentencias son ejecutadas repetidamente mientras la condición sea verdadera. Si la condición resulta falsa, las sentencias no se ejecutarán ninguna vez. El siguiente es un ejemplo de este tipo de estructuras:

```
while (condición)
{
    sentencias;
}
```

ESTRUCTURAS REPETITIVAS

A continuación, se presenta un ejemplo de un bucle o estructura repetitiva. Es un bucle o ciclo controlado por un contador. El bucle permite mostrar en pantalla los números enteros del 1 al 100:

```
int cont = 1; //Inicializar cont
while (cont ≤ 100) //Control del bucle
{
    cout << cont << endl; /*Imprime el contenido de cont*/
    ++cont; /*Incrementa cont, cuando llegue a 100 se saldrá del bucle*/
}
```

ESTRUCTURAS REPETITIVAS

Por otra parte, en el siguiente ejemplo se presenta un programa que permite leer varias notas. No se sabe cuantas notas se tendrán que leer. Simplemente este las acumula y cuenta dentro del bucle. Al terminar de procesar todas las notas, usa un valor centinela (-1) como último dato. Este último dato se utiliza para salirse del ciclo; por último, calcula e imprime el promedio:



ESTRUCTURAS REPETITIVAS

```
int nota, suma = 0, cont = 0;
const int CENTINELA = -1;
cout<<"Ingrese la nota (-1) para finalizar : ";
cin>>nota;
while (nota != CENTINELA)
{
    suma += nota;
    ++cont;
    cout<<"Ingrese la nota (-1) para finalizar : ";
    cin>>nota;
}
cout << "El promedio de nota es : "<< suma/cont << endl;t
```

ESTRUCTURAS REPETITIVAS



Siguiendo con los ejemplos, el siguiente ejercicio demuestra otra forma de utilizar la estructura *while*. En este, podemos ver cómo se ejecuta un bucle controlado a través de una pregunta al usuario. Este último, entonces, es el responsable de indicar si se desea la continuación del ciclo. Esto se realiza mediante la lectura a través de teclado de la decisión del usuario.

ESTRUCTURAS REPETITIVAS

```
int nota, resp, suma = 0, cont = 0;
cout<<"Existe alguna nota (1)Sí (2)No : ";
cin>>resp;
while (resp == 1)
{
    cout<<"Ingrese la nota : ";
    cin>>nota;
    suma += nota;
    ++cont;
    cout<<"Existe alguna otra nota (1)Sí (2)No? ";
    cin>>resp;
}
cout << "El promedio de notas es : " << suma/cont << endl;
```

ESTRUCTURAS REPETITIVAS

○ **Do while.** Esta sentencia va un paso más allá que la estructura *while*. Permite que las sentencias se ejecuten al menos una vez. Primero se ejecutan, y al final se evalúa la condición. Se repiten las sentencias hasta que la condición se haga falsa; o mejor dicho, se ejecuta el grupo de sentencias mientras la condición sea cierta; entonces, como mínimo, siempre se ejecutan las sentencias al menos una vez, ya que la condición de salida es la que se encuentra al final.

```
do  
<sentencias>  
while (condición);
```

ESTRUCTURAS REPETITIVAS

Ejemplo del uso de la estructura *do while*

```
char caracter;
do
{
    cout<<"Ingrese digito (0-9) : ";
    cin>>caracter;
}
while (caracter < '0' or caracter > '9');
cout<<"El digito es : "<<caracter<<endl;
```

ESTRUCTURAS REPETITIVAS

○ **For.** Resulta ideal para repetir una secuencia de instrucciones cuando se conoce la cantidad de veces que se requiere repetir la ejecución. Para ello, en primer lugar, se necesita una **inicialización**. Esta ofrece un valor a una variable que servirá para controlar el número de veces que debe repetirse el bucle. También, se necesita una **condición**. Esta determina cuándo debe parar de repetirse el bucle. Finalmente, un **incremento**. Este modifica el valor de la variable de control.

```
for (inicialización; condición; incremento)
{
    <sentencias>;
}
```

ESTRUCTURAS REPETITIVAS

En el siguiente ejemplo se suman diez números enteros secuenciales. Se inicia en el número “1” y se termina en el número “10”. Finalmente, se imprime el resultado obtenido:

```
int suma = 0;  
for (int n = 1; n <= 10; n++)  
    suma += n;  
cout << "La suma de los 10 primeros números es:" << suma << endl;
```

ESTRUCTURAS REPETITIVAS

No siempre el incremento tiene que ser de uno. El ciclo *for* tampoco se restringe a números. Por ejemplo, en el siguiente código es posible apreciar cómo se imprimen las letras desde 'c' hasta 'y' en incrementos de cinco:

```
#include <iostream>
using namespace std;
int main(){
    char letra;
    for (letra='c'; letra<='y'; letra+=5)
        cout << letra;
    return 0;
}
```

VIDEO

- Te invitamos a ver los siguientes videos:



ESTRUCTURAS SELECTIVAS

A diferencia de las estructuras repetitivas, enfocadas en ciclos o bucles, las estructuras selectivas se enfocan en **bifurcaciones**. Estas son sentencias para establecer una posible ruta, de acuerdo a una condición. Para ello, se lleva a cabo un determinado bloque de instrucciones y existen los siguientes tipos:

- **If.** Se basa en el resultado, ya sea verdadero o falso, en una expresión. Para ello, se requiere una condición y una sentencia. La primera es la que se evalúa como verdadera o falsa, y la segunda es la que se ejecuta si la condición es verdadera.

```
If (condición) sentencia;
```

ESTRUCTURAS SELECTIVAS

- **If... else.** Permite especificar que se realizarán acciones diferentes cuando la condición sea verdadera; así como cuando sea falsa. Si la condición es verdadera, entonces se ejecuta la “sentencia 1”; en caso contrario, se ejecuta la “sentencia 2”. A continuación, se muestran algunos ejemplos para diferenciar esta estructura con respecto al *if* simple.

```
if (condición)
    sentencia1;
else
    sentencia2
```

ESTRUCTURAS SELECTIVAS

```
#include <iostream>
using namespace std;
int main(){
    int annos = 2;
    float aumento = 0;
    float sueldo_base = 567;
    float sueldo_neto = 0;
    if (annos > 3){
        aumento = sueldo_base * 0.30;
        sueldo_neto = sueldo_base + aumento;
        cout << sueldo_neto;
    }else{
        aumento = sueldo_base * 0.15;
        sueldo_neto = sueldo_base + aumento;
        cout << sueldo_neto;
    }
}
```

ESTRUCTURAS SELECTIVAS

○ **Switch.** Es una instrucción de decisión múltiple. Con ella, se compara el valor de una expresión con una lista de constantes de tipo carácter o entero. En caso de que el valor de la expresión corresponda con alguna de las constantes, se ejecutan las acciones asociadas a esta. Ejemplo:

```
switch (expresión)
{
    case const1: instrucción(es);
    break;
    case const2: instrucción(es);
    break;
    case const3: instrucción(es);
    break; .....
    default: instrucción(es)
};
```

ESTRUCTURAS SELECTIVAS

En el siguiente ejemplo se evalúa lo que contiene el carácter *edo_civil*. Posteriormente, se guarda un mensaje y se imprime:

```
#include <iostream>
using namespace std;
int main(){
    string mensaje = "hola mundo";
    char edo_civil;edo_civil = 'S';
    switch (edo_civil){
        case 'S': mensaje = "SOLTERO"; break;
        case 'C': mensaje = "CASADO"; break;
        case 'D': mensaje = "DIVORCIADO"; break;
        default: mensaje = "Estado invalido"; }
    cout << mensaje;
}
```

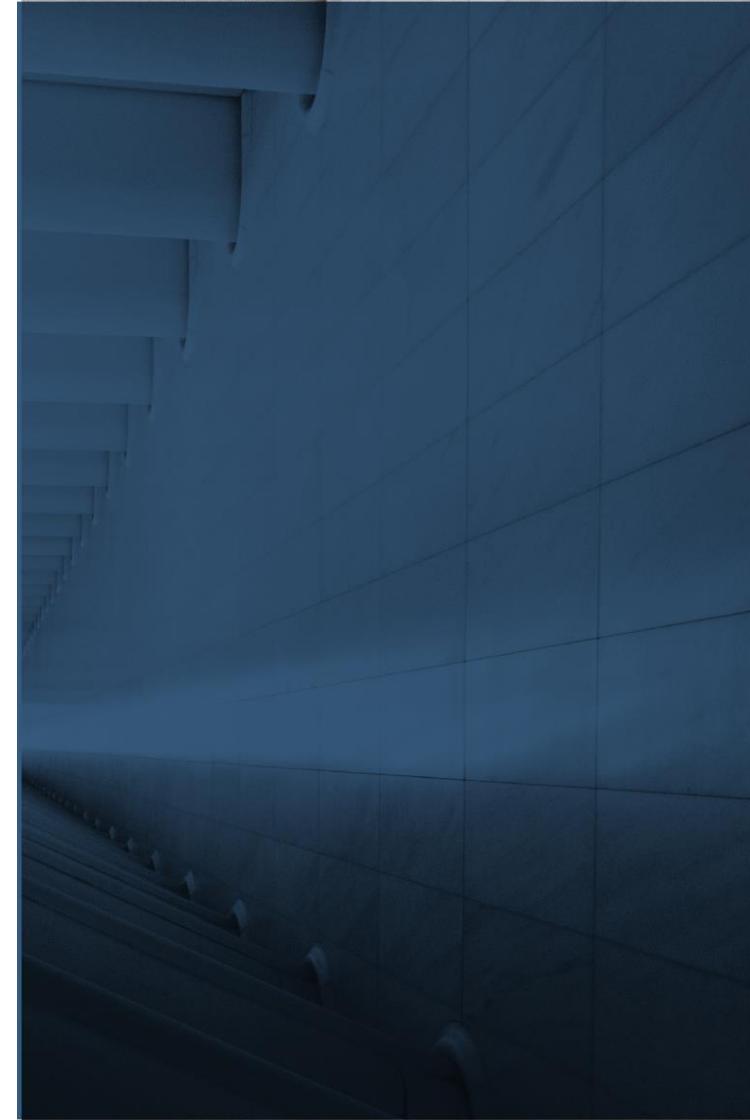
ACTIVIDAD 1

- Te invitamos a realizar la siguiente actividad:

Presiona el botón para descargar la actividad:



Presiona el botón para entregar la actividad:



El lenguaje C++ facilita la manera en la que las estructuras repetitivas y selectivas hacen referencia a la toma lógica de decisiones. Esto es básico para realizar innumerables tareas y resolver todo tipo de problemas a través de la programación. Sin este tipo de estructuras, la mayoría de los algoritmos no se podrían llevar a la práctica. En conclusión, este tipo de estructuras permiten la posibilidad de realizar tareas múltiples y tener una eficiencia superior.

CONCLUSIÓN



¡FELICIDADES!

Acabas de concluir la [cuarta unidad](#) de tu curso *Lenguajes de Programación I*. Te invitamos a finalizar este esfuerzo realizando el examen parcial correspondiente. Para ello, debes regresar a la pantalla principal y dar clic en *Presentar examen*.

3

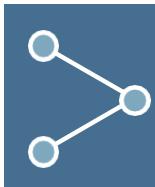
UNIDAD

ARREGLOS Y APUNTADORES

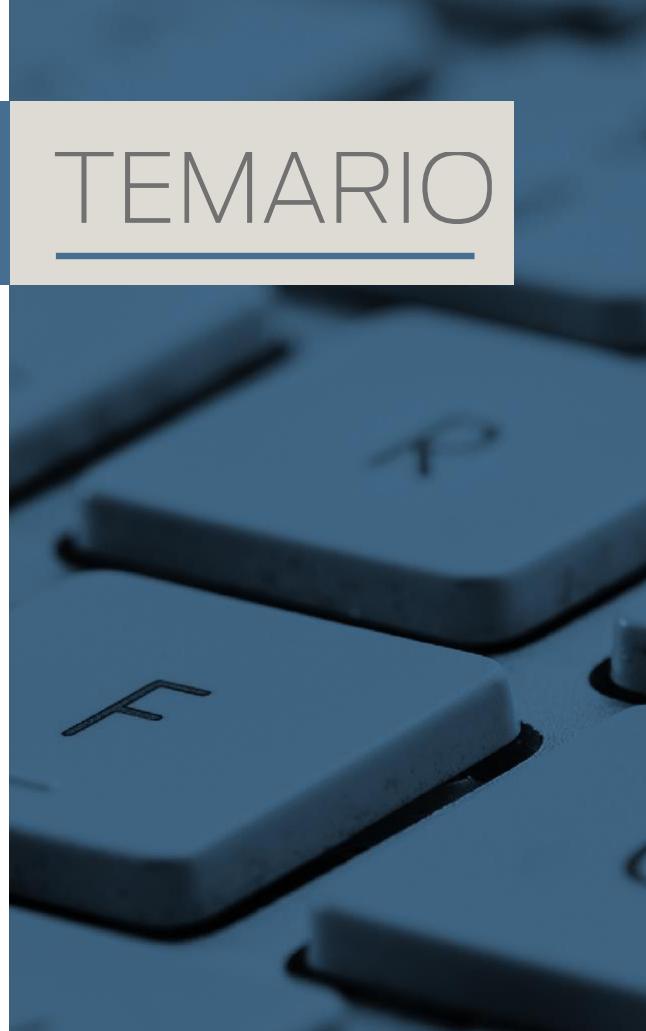


3.1

ARREGLOS



TEMARIO



3.2

APUNTADORES





INTRODUCCIÓN

En esta tercera unidad vamos a adentrarnos en los arreglos (*arrays*), también conocidos como vectores. Estas son, sin duda, una de las más útiles estructuras en programación. Además, en la segunda parte, conoceremos el uso de los apuntadores en C++. Estos son de gran ayuda cuando se necesitan valores y estructuras dinámicas.

COMPETENCIAS A DESARROLLAR



El alumno podrá implementar arreglos uni y multidimensionales.



El alumno podrá implementar apuntadores de memoria.



ARREGLOS

Un arreglo (array en inglés) es una **colección de variables relacionadas**, a las que se hace referencia por medio de un nombre común. Un arreglo hace referencia a conjuntos de datos que se almacenan en memoria de manera contigua con el mismo nombre; para diferenciar los elementos de un arreglo se utiliza un **índice**. En C++, un arreglo se conoce como un tipo de dato compuesto. Los arreglos pueden tener una o varias dimensiones:

Float arreglo [6];	Representación gráfica de un arreglo de una dimensión	1	arreglo [0]
		2	arreglo [1]
		3	arreglo [2]
		4	arreglo [3]
		5	arreglo [4]
		6	arreglo [5]

ARREGLOS

○ **Índice de un arreglo.** Todo arreglo está compuesto por cierto número de elementos. El índice es un número correlativo que indica la posición de un elemento del arreglo. Los índices en C++ van desde la posición “0” hasta la “-1”.

○ **Elemento de un arreglo.** Se trata de un valor particular dentro de la estructura del arreglo. Para acceder a un elemento del arreglo, es necesario indicar la posición o índice dentro del mismo. Ejemplo:

```
arreglo[0] //Primer elemento del arreglo  
arreglo[3] //Cuarto elemento del arreglo
```

ARREGLOS

Cuando se declara un arreglo, se reserva un solo bloque contiguo de memoria para almacenar todos sus elementos. Estos se almacenan en la memoria en el mismo orden que ocupan en el arreglo:

```
int main() {
    int i, a[10];
    for (i = 0; i < 10; i++) {
        cout << &a[i] << endl;
    }
    return 0;
}
```

ARREGLOS

Un **arreglo de una dimensión** es una lista de variables (todas de un mismo tipo), a la que se hace referencia por medio de un nombre común. Para declarar un arreglo de una sola dimensión se usa el formato general:

```
tipo_dato identificador[tamaño];
```

arreglo [3]:

1	arreglo [0]
2	arreglo [1]
3	arreglo [2]



ARREGLOS

Se accede a un elemento del arreglo indexándolo por medio de un número del elemento. Esto quiere decir que, si se desea acceder a su primer elemento, debe usarse el índice igual a 0. Para indexar un arreglo, se especifica el índice del elemento que interesa dentro de corchetes. Por ejemplo:

```
valor = arreglo[1];
```

ARREGLOS

Los arreglos empiezan en “0”; de manera que el índice “1” se refiere al segundo elemento. Para asignar el valor a uno de sus elementos, este se coloca en el lado izquierdo de una sentencia de asignación:

```
mi_arreglo[0] = 100;
```

C++ almacena arreglos de una sola dimensión en una localización de memoria contigua en el primer elemento, en la posición más baja. De esta manera, `mi_arreglo[0]` es adyacente a `mi_arreglo[1]`, que a su vez lo es a `mi_arreglo[2]`, y así sucesivamente. Es posible usar el valor de un elemento de un arreglo donde se usaría una variable sencilla o una constante.

ARREGLOS

A continuación, se muestra la declaración de un arreglo de una dimensión y tres elementos:

```
int arreglo[3];
```

En este caso

```
arreglo[0] → primer elemento  
arreglo[1] → segundo elemento  
arreglo[2] → tercer elemento
```

ARREGLOS

Los **arreglos multidimensionales** son también conocidos como **matrices**.

En este sentido, se llama matriz de $m \times n$ a un conjunto rectangular de elementos dispuestos en filas m , y en columnas n ; siendo estos números naturales.

Las matrices se denotan con letras mayúsculas: A, B, C...; y los elementos de las mismas con minúsculas: a, b, c ...





ARREGLOS

Un elemento genérico que ocupe, por ejemplo, la fila i y la columna j , se escribirá i,j . Si el elemento genérico aparece entre paréntesis, también representa a toda la matriz: $A(i,j)$.

La sintaxis es la siguiente:

```
tipoDatos nombreMatriz[filas][columnas];
```

ARREGLOS

Dentro de los arreglos multidimensionales, los que más se utilizan son los de dos dimensiones. Conocidos como **arreglos bidimensionales** o simplemente matrices. Una matriz de orden 3x4 se muestra a continuación. Siendo M una matriz de 3 filas y 4 columnas; la representación gráfica de sus posiciones sería:

M 3x4 Filas = 3, columnas = 4

	c0	c1	c2	c3
f0	$m [f0, c0]$	$m [f0, c1]$	$m [f0, c2]$	$m [f0, c3]$
f1	$m [f1, c0]$	$m [f1, c1]$	$m [f1, c2]$	$m [f1, c3]$
f2	$m [f2, c0]$	$m [f2, c1]$	$m [f2, c2]$	$m [f2, c3]$

ARREGLOS

Para un arreglo multidimensional, la declaración es la siguiente:

```
tipo_dato identificador [dimensión1] [dimensión2] ... [dimensiónN];
```

Donde N es un número natural positivo.

		Columnas		
		c0	c1	c2
Filas	f0	a	x	w
	f1	b	y	10

ARREGLOS



Los datos anteriores se expresan de la siguiente manera. Es importante señalar que en la declaración se forma una tabla de dos filas y tres columnas, en donde cada fila es un arreglo de una dimensión:

```
char m[2][3];
```

VIDEO

- Te invitamos a ver los siguientes videos:



APUNTADORES

Un apuntador se puede definir como una **variable que contiene una dirección de memoria** (donde posiblemente se almacene el valor de otra variable). Para crear apuntadores se utiliza el operador “*”, como se muestra a continuación:

```
int main() {
    int a = 10;
    int *p;
    p = &a;
    cout << "Valor de p: " << p << endl;
    cout << "Valor en o: " << *p << endl;
    cout << "Dirección de p: " << &p << endl;
}
```

APUNTADORES

La memoria puede verse como un conjunto de celdas numeradas y ordenadas; cada una de las cuales puede almacenar un byte de información. El número correspondiente a cada celda se conoce como su **dirección**.

Cuando se crea una variable, el compilador reserva el número suficiente de celdas de memoria (bytes) requerido para almacenar la variable, y se encarga de que los espacios reservados no se traslapen.





APUNTADORES

Una variable que puede almacenar una dirección de memoria se llama **puntero**. Es posible obtener la dirección de memoria donde se encuentra almacenada con una variable mediante el operador de referencia “&”.

Una variable de tipo puntero puede guardar direcciones de variables de un tipo determinado: punteros a *int*, *double*, *char*, etc. Esto se aprecia en el siguiente ejemplo:

APUNTADORES

```
int i=3, *p,*r; // p y r son punteros a entero
double d=3.3, *q; // q es un puntero a double
char* c='a', *t; //t es un puntero a carácter

p=&i;
r=p;
p = q; // es un error, porque son punteros de diferente tipo.
```

APUNTADORES

En C/C++, el nombre de un arreglo es también un apuntador al primer elemento del mismo. De hecho, el lenguaje C no puede distinguir entre un arreglo y un apuntador; ambos son completamente intercambiables. Por ejemplo:

```
int main() {
    int i, a[10], *p;
    for (i = 0; i < 10; i++) { a[i] = i; };
    p = a;
    for (i = 0; i < 10; i++) {
        cout << p[i] << endl;
    }
    return 0;
}
```

APUNTADORES



Un apuntador siempre está asociado con el tipo de dato que apunta. Por ejemplo: apuntador a *int*, apuntador a *float*, etc. Esto hace posible definir la operación $p+k$. Donde p es un apuntador y k es un entero. El resultado de esta operación es:

```
p + k * sizeof(tipo)
```

Donde *tipo* es el tipo de datos asociado a p .

APUNTADORES

Observa entonces que, si p es un arreglo:

$\&p[k]$ es equivalente a $p+k$; y $p[k]$ equivale a $*(p+k)$

Para cada función , el compilador C/C++ reserva una cierta cantidad de memoria para almacenar las variables locales.

En algunas ocasiones, será necesario utilizar un arreglo que requiera una mayor cantidad de memoria:

```
int main() {
    int arreglote[1024 * 1024];
    return 0;
}
```

APUNTADORES

La solución consiste en asignar memoria a un arreglo de manera dinámica, mediante el **operador new**:

```
int main() {
    int i, n = 1024 * 1024;
    int *a = new int[n];
    if (a != NULL) {
        for (i = 0; i < n; i++) {a[i] = i; }
        delete [] a;
    }
    return 0;
}
```

Es importante siempre liberar la memoria reservada mediante el operador **delete[]**, una vez que ya no se utiliza.

APUNTADORES

El lenguaje C++ define una constante especial llamada **NULL**. Esta no apunta a ningún lugar válido en la memoria. Tiene un valor numérico de **cero**.

Se recomienda inicializar los apuntadores con una dirección válida; o bien, con *NULL*. Para evitar sobrescribir información.

El operador *new* devuelve *NULL* si este no es capaz de reservar la cantidad de memoria solicitada. Esto **permite detectar la falta de memoria en un programa**.



ACTIVIDAD 2

- Te invitamos a realizar la siguiente actividad:

Presiona el botón para descargar la actividad:



Presiona el botón para entregar la actividad:



CONCLUSIÓN

Los arreglos son estructuras de datos que nos permiten almacenar/manipular conjuntos de datos agrupados de manera eficiente. En el lenguaje C++, los arreglos están íntimamente relacionados con los punteros, cuya función es la de reservar espacio en memoria para garantizar su almacenamiento y posterior consulta.



¡FELICIDADES!

Acabas de concluir la [tercera unidad](#) de tu curso *Lenguajes de Programación I*. Te invitamos a finalizar este esfuerzo realizando el examen parcial correspondiente. Para ello, debes regresar a la pantalla principal y dar clic en *Presentar examen*.

4

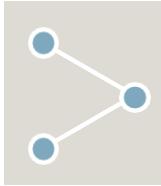
UNIDAD

CLASES Y OBJETOS

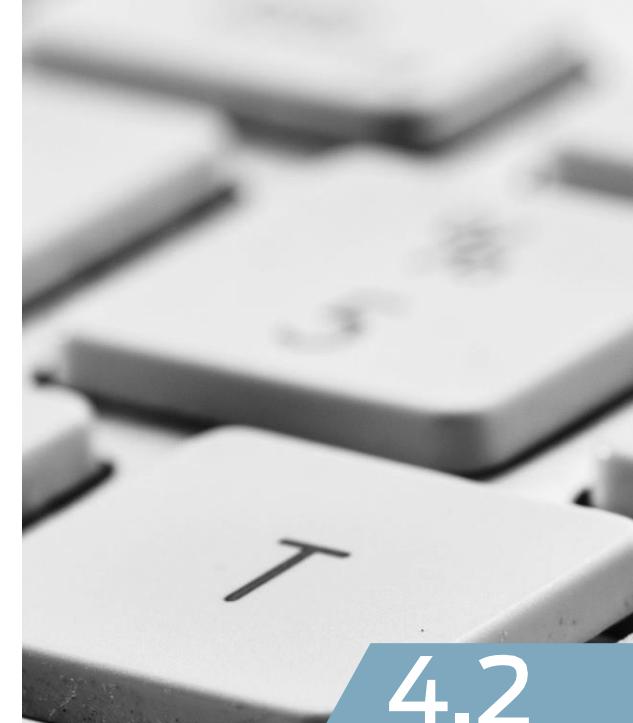
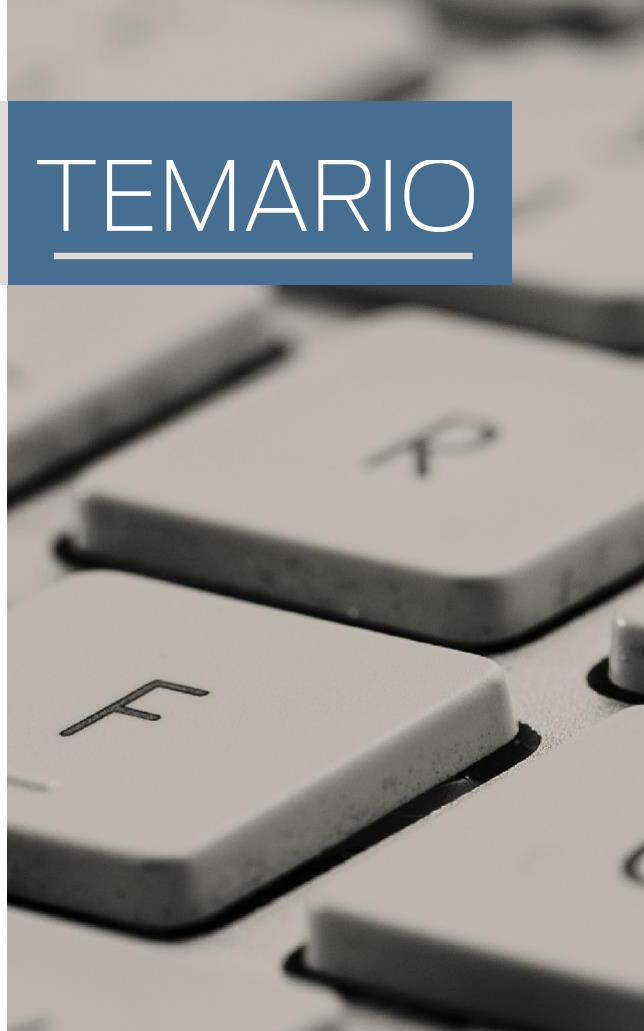
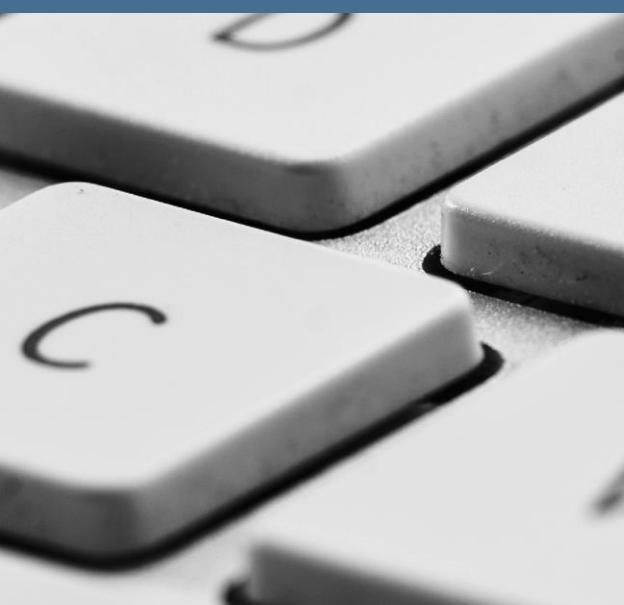


4.1

CLASES



TEMARIO



4.2

OBJETOS



INTRODUCCIÓN

En esta cuarta unidad vamos a conocer acerca de las clases en C++. Estas son un tipo de dato definido por el programador. Las clases contienen toda la información necesaria para construir un objeto, así como sus métodos. Por otra parte, conoceremos más acerca de los objetos, los cuales sirven para manipular los datos de entrada para la obtención de datos de salida específicos.

COMPETENCIAS A DESARROLLAR



El alumno comprenderá el manejo de clases en un programa, así como las herramientas necesarias.



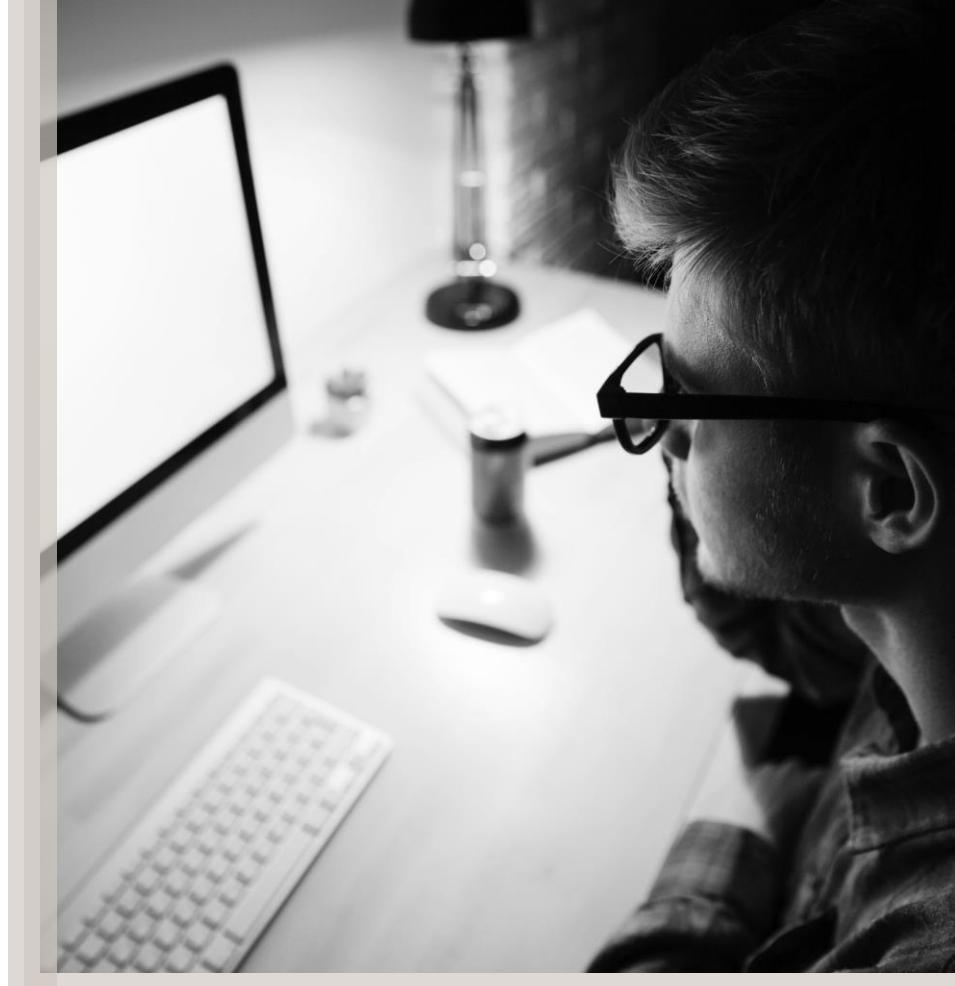
El alumno será capaz de aplicar la sintaxis de las estructuras de los objetos en la programación orientada a objetos.



CLASES.

Una clase puede definirse como un **patrón para construir objetos**. En C++, un objeto es simplemente un tipo de variable de una clase determinada.

Es importante distinguir entre clases y objetos. La clase es solo una declaración y no tiene asociado ningún objeto; de modo que no puede recibir mensajes ni procesarlos. Cosa que sí hacen los objetos.





Se puede acceder a los datos y métodos de una clase de la misma forma que se accede a un campo de una estructura; es decir, con “.” o con “->”, seguido del nombre del elemento a usar.

Para que C++ sepa distinguir entre una función y un método, siempre debemos poner un prefijo a este último, así como el nombre de la clase a la que pertenece seguido de “::” (operador de ámbito). Por ejemplo:

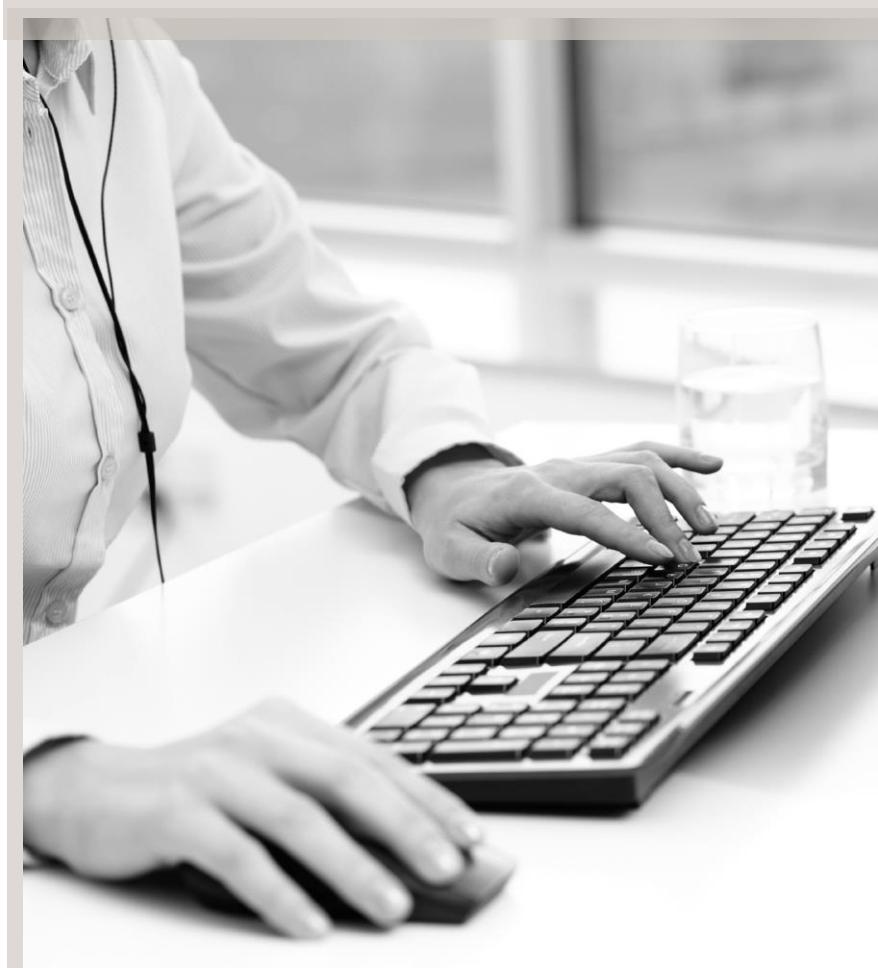
CLASES

```
class Celda {  
    char Caracter, Atributo;  
public: void FijaCelda(char C, char A);  
void ObtenCelda(char &C, char &A);  
};  
  
void Celda::FijaCelda(char C, char A)  
{  
    Caracter=C;Atributo=A;  
}  
void Celda::ObtenCelda(char &C, char&A)  
{  
    Caracter; A=Atributo;  
}
```

CLASES

Las variables y los métodos son de instancia. Para usarlos, se deben crear instancias de la clase que se necesita, o bien, tener una referencia a la misma.

Cada instancia de objetos tiene variables y métodos. En este sentido, para cada uno, el comportamiento será diferente. Las clases mismas también pueden tener variables y métodos. Estos se llaman **miembros de clases**. Se declaran con la palabra clave **static**.



A continuación, los miembros más importantes:

○ **Campo.** Son variables declaradas en el ámbito de clase. Un campo puede ser un tipo numérico integrado o una instancia de otra clase.

○ **Constantes.** Son campos o propiedades cuyo valor se establece en tiempo de compilación, y no se puede cambiar.

CLASES

○ **Propiedades.** Son métodos de una clase a los que se obtiene acceso como si fueran campos de esta. Puede proporcionar protección a un campo de clase, con el fin de evitar que se cambie sin el conocimiento del objeto.

○ **Métodos.** Definen las acciones que una clase puede realizar. Pueden aceptar parámetros que proporcionan datos de entrada, y devolver datos de salida a través de parámetros.





○ **Eventos.** Estos proporcionan a otros objetos las notificaciones sobre lo que ocurre, como clics en botones o la realización correcta de un método. Los eventos se definen y desencadenan mediante delegados.

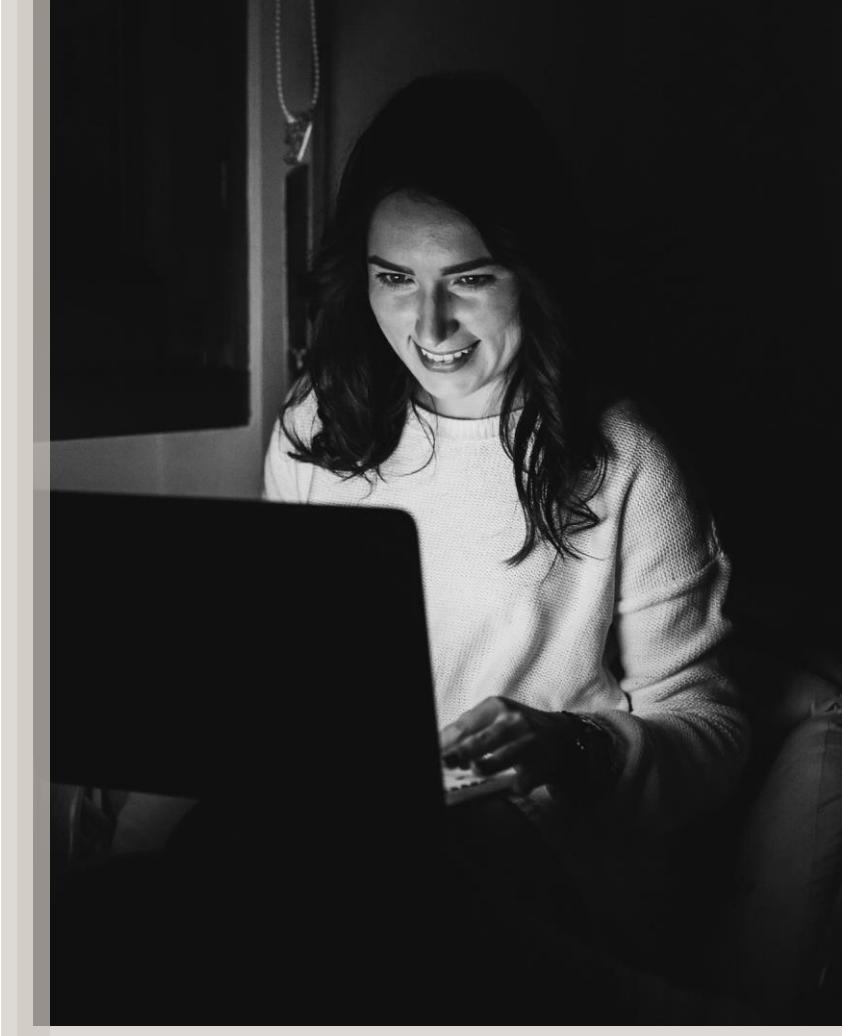
○ **Operadores.** Los operadores sobrecargados se consideran miembros de clase. Si se sobrecarga un operador, se define como miembro estático público en una clase.

CLASES

○ **Constructores.** Son métodos a los que se llama cuando el objeto se crea por primera vez. Se usan para inicializar los datos de un objeto.

○ **Destructores.** Se utilizan para asegurarse de que los recursos que se deben liberar se controlan apropiadamente.

○ **Tipos anidados.** Son tipos declarados dentro de otro tipo. Se usan para describir objetos utilizados únicamente por los tipos que contienen.



CLASES

Las diferencias entre los miembros de clase y los de instancias son las siguientes:

- Cada instancia de una clase comparte una sola copia de una variable de clase.
- Puede llamar a los métodos de clases en la clase misma, sin tener una instancia.
- Los métodos de instancias pueden acceder a las variables de clases, pero los métodos de clases no pueden acceder a variables de instancias.
- Los métodos de clases pueden acceder solo a las variables de clases.

CLASES

El **ámbito** es el contexto que tiene un nombre dentro de un programa. Este determina en qué partes del programa puede ser usada una entidad. Esto es útil para que se pueda volver a definir una variable con un mismo nombre, en diferentes partes del programa, sin conflictos entre ellos. Si una variable es declarada dentro de un bloque, esta será válida solo dentro de ese mismo bloque.



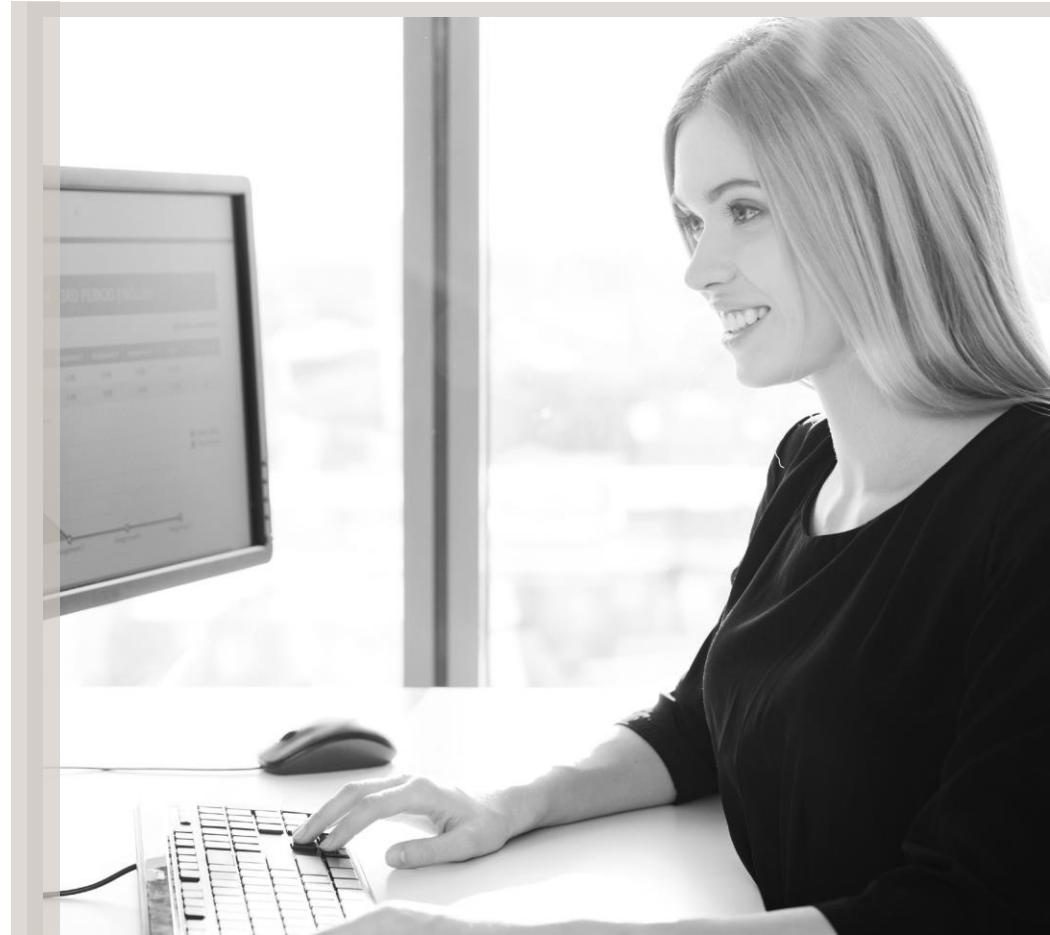
Una variable dentro de un bloque es una **variable local**, y solo tiene alcance dentro del bloque en el que se creó, así como en sus bloques hijos. En este sentido, no tiene alcance en sus bloques hermanos ni padres.

Una variable definida fuera de cualquier bloque es una **variable global**. Por lo tanto, cualquier bloque puede acceder a ella y modificarla.

CLASES

En el caso de la programación orientada a objetos (POO), una variable global dentro de una clase es llamada **variable de instancia**. De esta manera, cada objeto creado con esa clase tiene una variable de instancia.

Adicionalmente, existen variables globales que son comunes a todos los objetos creados con una clase. Estas son llamadas **variables de clase**.





Existen dos tipos de alcance en las variables:

- **Estático.** También llamado lexicográfico. Determina el tiempo de compilación.
- **Dinámico.** Estas se verificarán en el hilo de ejecución.

CLASES

En una definición de clase, un **especificador de acceso** se utiliza para controlar la visibilidad de los miembros de una clase fuera de su ámbito.

C++ introduce **tres nuevas palabras clave** para establecer las fronteras de una estructura:

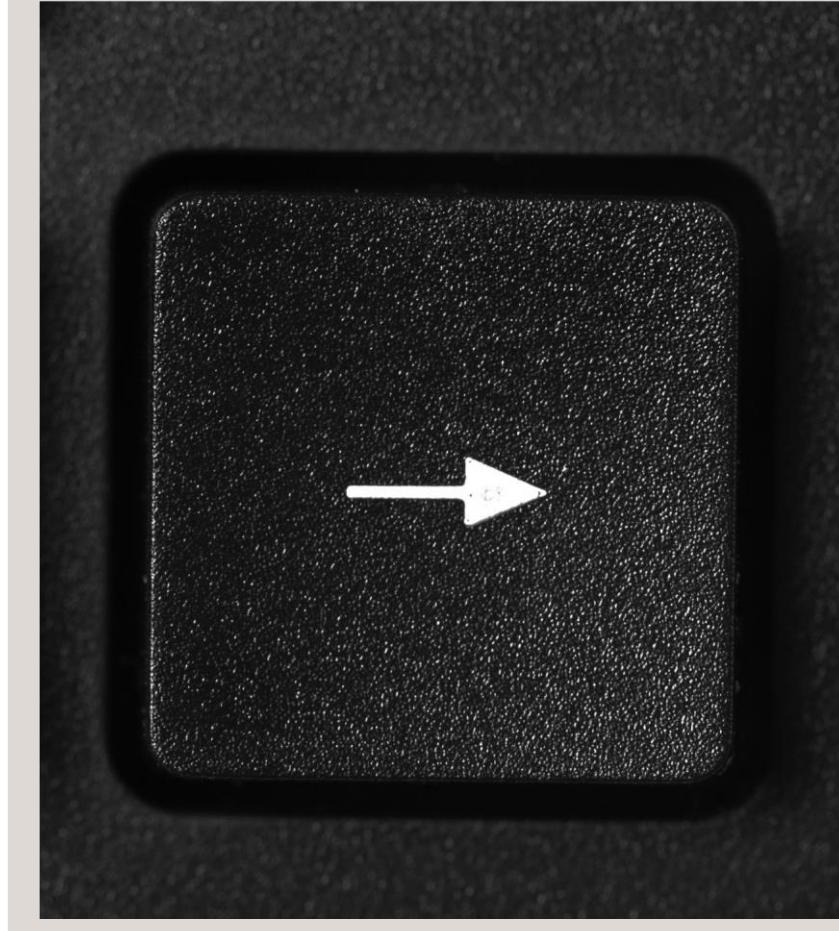
- public*
- private*
- protected*



CLASES

Los especificadores de acceso se usan solo en la declaración y en las estructuras, y cambian las fronteras para todas las declaraciones que los siguen. Cuando se utiliza un especificado de acceso, este debe ir seguido de “.”.

Todos los miembros precedidos por el especificador *public* son visibles fuera de la clase. Por ejemplo, un miembro público es visible desde el *main()*, como es el caso de *cin.get():cin.get()* // donde *cin* es el objeto y *get* es la función de acceso público.



CLASES

Todos los miembros precedidos por el especificados *private* quedan ocultos para funciones fuera de la clase. Estos pueden ser solo referenciados por funciones dentro de la misma clase.

Miembros precedidos por *protected* pueden ser accedidos por miembros de la misma clase, clases derivadas y clases amigas (*friend*).

Cuadro resumen (X representa que tiene acceso):

ESPECIFICADOR	MIEMBROS DE CLASE	FRIEND	CLASES DERIVADAS	OTROS
PRIVADO	X	X		
PROTECTED	X	X	X	
PUBLIC	X	X	X	X



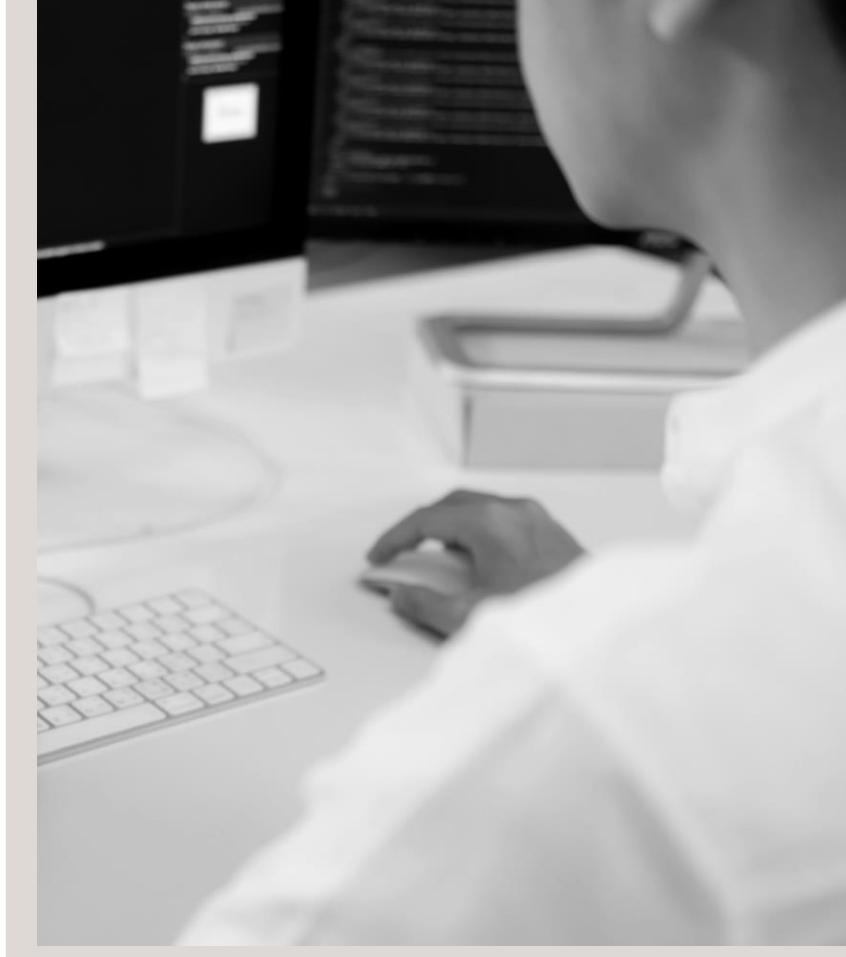
En la mayoría de los lenguajes de programación existen **librerías**. Estas se entienden como un conjunto de clases que poseen una serie de métodos y atributos. Lo realmente interesante de las librerías es que facilitan las operaciones.

Las librerías nos permiten **reutilizar código**; es decir, podemos hacer uso de los métodos, clases y atributos que las componen para evitar tener que implementar nosotros mismos esas funcionalidades.

CLASES

Los **paquetes** agrupan las clases en librerías (bibliotecas). En los paquetes, las clases son únicas, comparadas con las de otros paquetes. Estas permiten controlar el acceso. Esto quiere decir que los paquetes proporcionan una forma de ocultar clases, evitando que otros programas o paquetes accedan a clases que son de uso exclusivo de una aplicación determinada.

Los paquetes se declaran usando la palabra reservada *package*, seguida del nombre del paquete.



Por su parte, la sentencia ***import*** se utiliza para incluir en una lista de paquetes en los que se desea buscar una clase determinada. Su sintaxis es:

```
import nombre_paquete.Nombre_Clase;
```

Esta sentencia, o grupo de sentencias, debe aparecer antes que cualquier declaración de clase en el código fuente.

CLASES

Una excepción sucede cuando parte del código puede no ejecutarse por algún error inesperado. En este caso, si ocurre una excepción, se interrumpe la normal ejecución del código. Estas se pueden manejar realizando una acción adecuada para dejar el sistema en un estado estable.

En consecuencia, se separa el código para el caso en que ocurra una situación excepcional:

- **Try.** Identifica un bloque de código en el cual puede surgir una excepción.
- **Throw.** Causa que se origine una excepción.
- **Catch.** Identifica el bloque de código en el cual se maneja la excepción.



Cuando una excepción es lanzada (*throw*), la secuencia de ejecución continúa con el bloque *catch*.

Después de ejecutarse el código del bloque *catch*, se continua con la siguiente iteración del *for*.

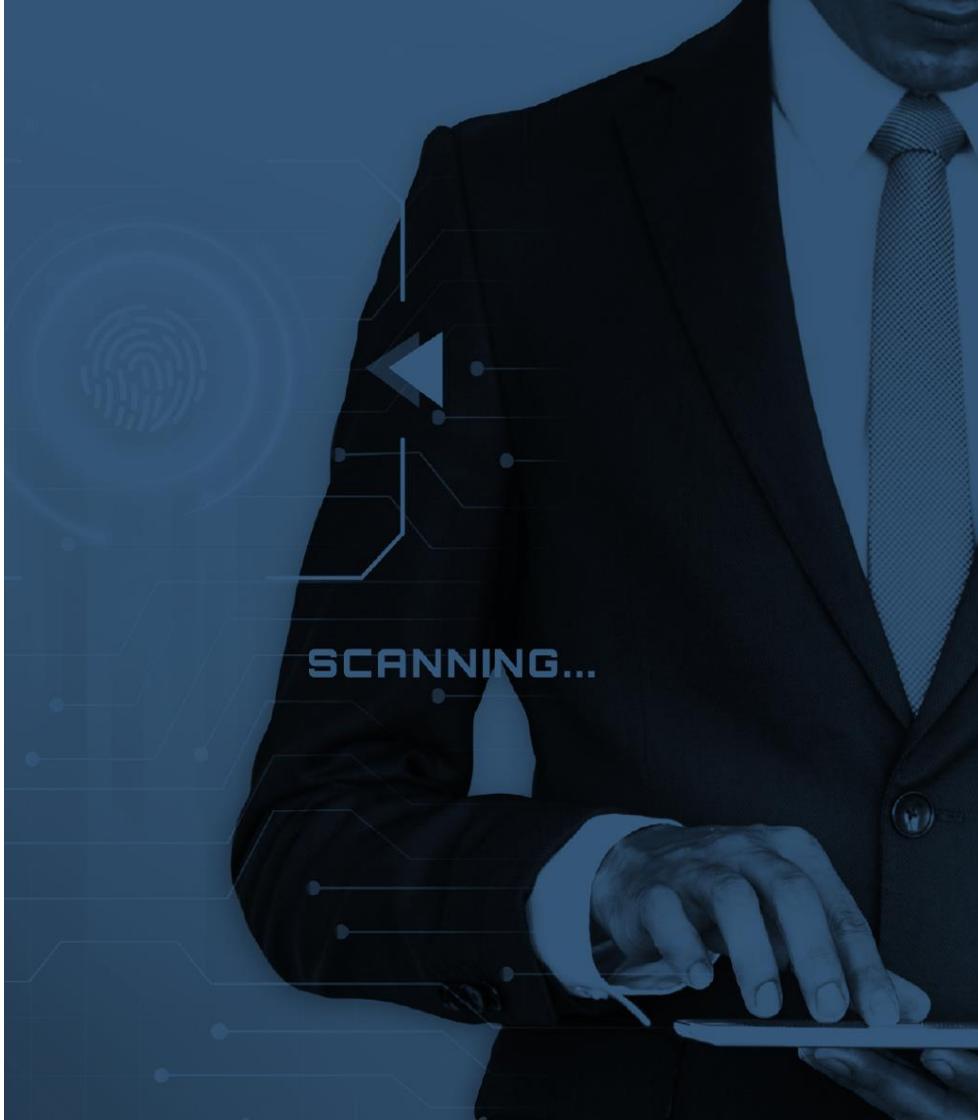
El compilador considera los bloques *try* y *catch* como una unidad única. A continuación, un ejemplo del **manejo de excepciones**:

CLASES

```
int main(void) {
    int counts[] = {34,54,0,27,0,10,0};
    int time = 60; //One hour in minutes
    for(int i = 0; i < sizeof counts /sizeof counts [0]; i++)
        try {
            cout << endl << "Hour" << i+1;
            if(counts[i] == 0)
                throw "Zero count - calculation not possible.";
            cout << "minutes per item:" <<
                static_cast<double>(time)/counts[i];
        } catch(const char aMessage[]) {
            cout << endl << aMessage << endl;
        }
    return0;
}
```

VIDEO

- Te invitamos a ver el siguiente video:



OBJETOS

Una clase es una plantilla que define los datos y métodos del objeto. Por su parte, **un objeto es una instancia de una clase**. Se pueden crear muchas instancias de una clase.

La creación de una instancia se conoce **instanciación**.

Una vez que se define una clase, su nombre se convierte en un nuevo tipo de dato. Este se utiliza tanto para declarar una variable de ese tipo como para crear un objeto del mismo.



OBJETOS

La sintaxis para declarar un objeto es:

```
NombreClase nombreObjeto;
```

Ejemplo:

```
Circulo miCirculo; // Declara la variable mi circulo
```

La variable *miCirculo* es una instancia de la clase *Circulo*. A la creación de un objeto de una clase se llama **creación de una instancia de una clase**. Un objeto es similar a una variable que tiene un tipo de clase. La creación de variables de un tipo de dato primitivo se realiza simplemente declarándolas. Esta operación crea la variable y le asigna espacio en la memoria.

OBJETOS

Ejemplo:

```
void main()
{
    Automovil Carro;
    Carro.set_Puertas( 4 );
    cout << "Introduce los datos del auto: ";
    Carro.Input();
    cout << "Datos Introducidos: ";
    Carro.Display();
    cout << "Es carro"
        << Carro.get_Puertas()
        << " doors.\n";
}
```

OBJETOS

Para cada objeto declarado de una clase se mantiene una copia de sus datos, pero todos comparten la misma copia de las funciones de esa clase.

Para solucionar esto, se utiliza el puntero especial ***this***. Este puntero tiene asociado cada objeto, y apunta a sí mismo. Este se usa para acceder a sus miembros. A continuación se muestra un ejemplo de uso del puntero *this*:



OBJETOS

```
#include <iostream>
using namespace std;
class clase
{
public: clase() {} void
EresTu(clase& c) {
if(&c == this)
    cout << "Si, soy yo."
<< endl;
else
    cout << "No, no soy
yo." << endl;
```

```
} };
int main() {
clase c1, c2;
c1.EresTu(c2);
c1.EresTu(c1);
return 0;
}
```

OBJETOS

Cada vez que se define una variable de tipo básico, el programa ejecuta un procedimiento que se encarga de asignar la memoria necesaria y, si se requiere, también realizará las inicializaciones pertinentes. De forma complementaria, cuando una variable queda fuera de ámbito, se llama a un procedimiento que libera el espacio que estaba ocupando dicha variable.





Al método que se llama cuando creamos una variable se le denomina **constructor**. Por su parte, al que se llama cuando se destruye una variable, se le denomina **destructor**.

Una clase puede tener uno o varios constructores, pero solo un destructor.

Es importante recordar que el constructor debe tener el mismo nombre de la clase a la que pertenece.

OBJETOS

Cuando se crea una clase, siempre se llama automáticamente a un constructor.

```
Polinomio pol1;  
Polinomio * pol2 = new (nothrow) Polinomio;
```

Se emplea para iniciar los objetos de una clase:

```
Polinomio pol3 (15); // Establece MaxGrado=15
```

Es particularmente útil reservar, si es necesario, memoria para ciertos campos del objeto. Su liberación se realiza, en ese caso, con el destructor. Cuando el objeto haya perdido memoria dinámicamente, debe implementarse el operador de asignación. Siempre debería implementarse el constructor básico (sin parámetros). De manera que el programador pueda controlar y personalizar la creación e iniciación. Como mencionamos anteriormente, solo existe un destructor para una clase. Cuando un objeto deja de existir, siempre se llama automáticamente al destructor.

En la programación orientada a objetos, los datos y el código que actúa sobre estos se convierten en una entidad única denominada clase. Esta es una evolución del concepto de estructura, ya que contiene la declaración de los datos. Sin embargo, a la clase se le añade la declaración de las funciones que manipulan dichos datos. Estas funciones se denominan métodos o función miembro. Además, en la clase se establecen permisos de acceso a sus miembros. Por defecto, en una clase los datos y las funciones se declaran como privados. De esta manera, se niega a las entidades exteriores el acceso a los miembros privados de un objeto.

Los objetos son miembros de una clase. Esta última es definida por el usuario. Está conformada por los métodos y los datos que definen las características comunes a todos los objetos de una clase.

CONCLUSIÓN



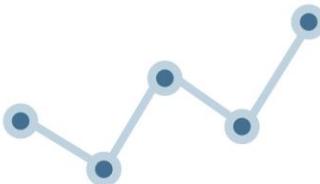
¡FELICIDADES!

Acabas de concluir la [cuarta unidad](#) de tu curso *Lenguajes de Programación I*. Te invitamos a finalizar este esfuerzo realizando el examen parcial correspondiente. Para ello, debes regresar a la pantalla principal y dar clic en *Presentar examen*.





PROYECTO FINAL



PROYECTO FINAL

- Te invitamos a realizar el siguiente proyecto final:

Presiona el botón para descargar el proyecto final:



Presiona el botón para entregar el proyecto final:

