



# C++ SEMANA 1

# Orígenes

C++ es un lenguaje de programación creado por el científico informático danés Bjarne Stroustrup a principios de la década de 1980. Su principal objetivo (en un principio) fue dotar al lenguaje C con capacidades de orientación a objetos. Cabe destacar que C++ no es lenguaje puramente orientado a objetos como por ejemplo Java. C++ puede ser empleado para desarrollar software utilizando para ello un modelo de programación puramente orientado a objetos o bien puramente estructurado o una combinación de ambos. Debido a su alto rendimiento, se utiliza para desarrollar desde drivers para dispositivos hasta sistemas operativos, video juegos y sistemas embebidos, aplicaciones web o cualquier aplicación donde el rendimiento es importante. C++ es un lenguaje **compilado** y por tal motivo para desarrollar aplicaciones para distintos sistemas operativos es necesario contar con el compilador que se ajuste a la plataforma para la cual vamos a desarrollar nuestro software.

# Orígenes de C++



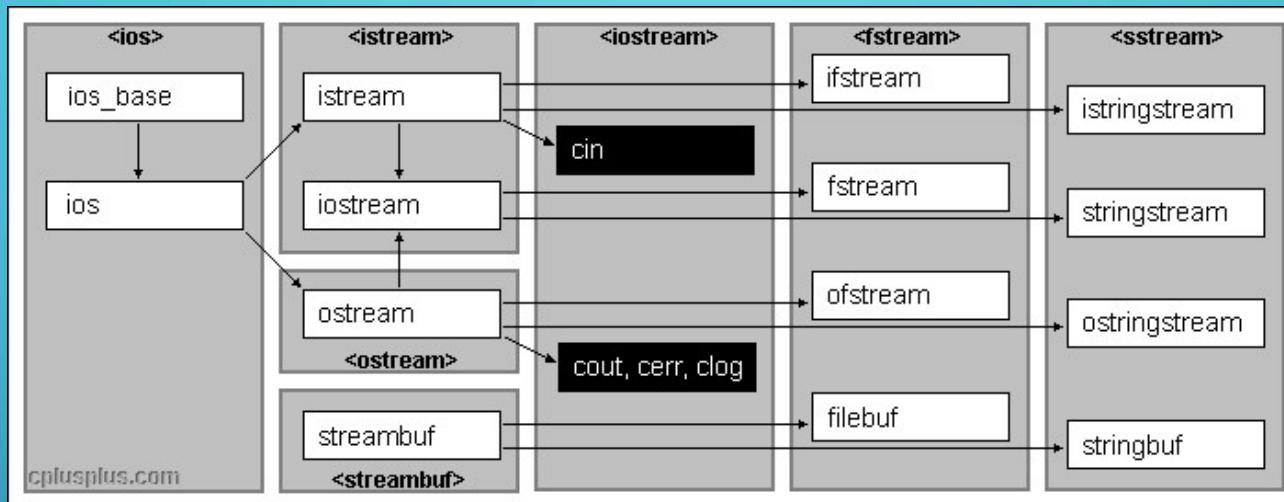
# Librería Standard

C++ incorpora una gran cantidad de código preescrito (colección de clases y funciones) que proporciona funcionalidades que son requeridas en muchos programas: Este conjunto de funcionalidades conforma la biblioteca Standard (*Standard Library o STL*).

- Manejo de Entrada/Salida, formateo de flujo de salida
- Contenedores: son las estructuras de datos genéricos.
- Algoritmos: ofrece algunos algoritmos comunes que pueden tener los contenedores, como un algoritmo de ordenación.
- Iteradores: permiten ir recorriendo estas estructuras de datos.

# Librería Standard

## Entrada / salida



Por ejemplo **iostream** define los siguientes objetos:

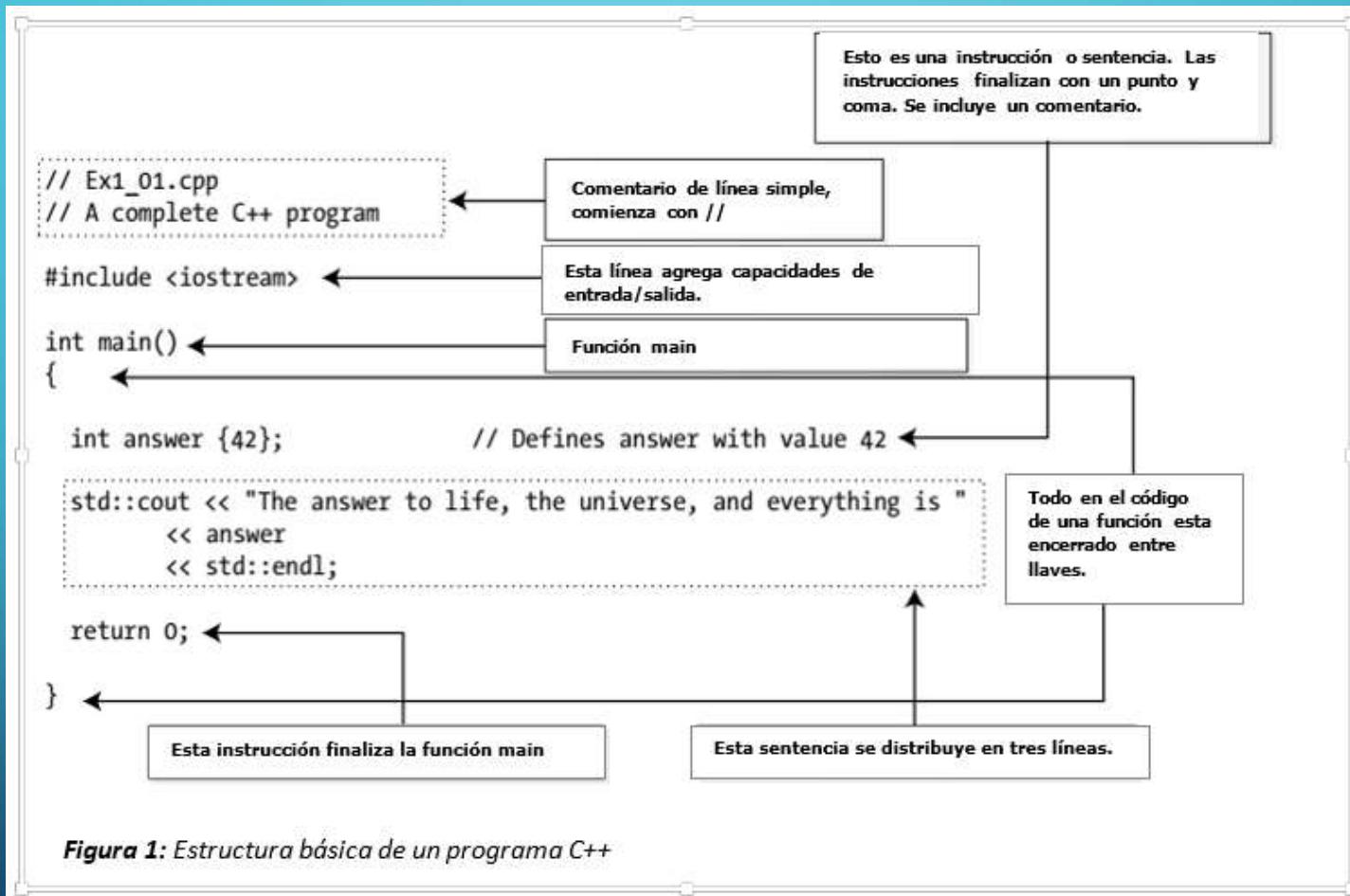
**cin** : Flujo de entrada

**cout** : Flujo de salida

**cerr** : Flujo de error no almacenado.

**clog** : Flujo de error almacenado.

# Estructura básica de un programa C++



# Archivos fuentes y de encabezado

En C++ la extensión de archivo **.cpp** se utiliza usualmente para referirse a archivos fuente, aunque también podemos encontrar **.cc**, **.cxx**, **.c++** como extensiones de archivos fuente.

El código C++ se almacena en dos tipos de archivos. Junto a los archivos fuente, también existen los llamados archivos de encabezado (headers). Los archivos de encabezado o cabecera contienen, entre otras cosas, prototipos de funciones y definiciones para clases y plantillas (templates), que son utilizados por el código ejecutable en un archivo **.cpp**. Los nombres de los archivos de cabecera suelen tener la extensión **.h**, aunque también se utilizan otras extensiones como **.hpp**.

# Comentarios

```
// Comentario de linea simple
/*
    Comentario de lineas múltiples
*/
```

Comentario para documentar código a través de la herramienta Doxygen por ejemplo.

```
*****
 * @brief Report both the error name of the source file and the source line number.
 * @param filename: pointer to the source file name.
 * @param uline: error line source number.
 * @retval None
*****/
```

# Directivas de preprocessamiento

```
#include <iostream>
```

Las directivas de preprocessamiento hacen que el código fuente se modifique de alguna manera antes de que se compile en forma ejecutable. La directiva de preprocessamiento `#include <file>`, agrega el contenido del archivo `file` al archivo fuente en el que se incluye la directiva. En este caso agrega el archivo de cabecera de la biblioteca estándar con el nombre `iostream` a este archivo fuente.

`iostream` contiene definiciones que se necesitan para realizar entradas desde el teclado y salida de texto a la pantalla utilizando las rutinas de la biblioteca estándar.

A través de `iostream` accedemos a los objetos, por ejemplo, `cout`, `cin`.

# Directivas de preprocessamiento

Ejemplo de uso de objetos cout y cin:

```
std::cout << "Varianza estimada: " << std::endl;  
std::cout << "Ingrese limite de calculo: ";  
std::cin >> limite_calc;
```

# Instrucciones o sentencias

Cada sentencia o instrucción en C++ debe terminar con un punto y coma. Este carácter le indica al compilador que la instrucción en cuestión terminó y que puede procesarla y pasar a la próxima. Por ejemplo:

```
int cantidad_items {};
```

Esta en una instrucción o sentencia de declaración e inicialización a cero de una variable llamada cantidad\_items

# Funciones o métodos

Las funciones o métodos en C++ tienen la siguiente estructura:

```
tipo_retorno nombre_función(par1, par2,..parn)
{
    bloque_instrucciones
    return(..);
}
```

El punto de entrada o inicio de cualquier programa C++ es la función `main`. Este método tiene una firma característica que es:

```
int main()
{
    bloque_instrucciones;
    return 0..n
}
```

# Nombres de espacio

Un gran proyecto puede involucrar a muchos programadores trabajando concurrentemente. Esto puede ocasionar un problema con los nombres de variables y funciones, ya que el mismo nombre puede ser utilizado para distintas cosas por diferentes programadores. Los espacios de nombres solucionan este problema agrupando bajo un prefijo declarativo a funciones, clases, tipos, etc. La agrupación se lleva a cabo utilizando llaves { }. En definitiva se trata de solucionar el conflicto de nombres. Para hacer referencia a las entidades agrupadas en un mismo nombre de espacio utilizamos el **operador de resolución de ámbito**, que es un nombre glamoroso para referirse a :: .La librería estándar (STL) esta agrupada bajo en namespace std. Observe el ejemplo siguiente:

# Nombres de espacio

```
#include <iostream>

namespace example_ns {

    int dummy_var {7};

    void dummy_fun()
    {
        std::cout << "Dummy function called!!" << dummy_var << std::endl;
    }

    int main()
    {
        example_ns::dummy_fun();

        return 0;
    }
}
```

# Nombres de espacio

Un gran proyecto puede involucrar a muchos programadores trabajando concurrentemente. Esto puede ocasionar un problema con los nombres de variables y funciones, ya que el mismo nombre puede ser utilizado para distintas cosas por diferentes programadores. Los espacios de nombres solucionan este problema agrupando bajo un prefijo declarativo a funciones, clases, tipos, etc. La agrupación se lleva a cabo utilizando llaves { }. Las llaves en C++ actúan como delimitadores de bloque. Un bloque esta formado por un conjunto de instrucciones

# Nombres de espacio

Existen una enorme cantidad de clases y funciones escritas por distintos programadores que están agrupadas bajo en namespace std. Es por esta razón que por ejemplo utilizamos el objeto cout, debemos incluir el header `iostream`, pero si utilizamos el manipulador de cadenas `setprecision()` debemos incluir el header `iomanip`. De esta forma la STL puede ser ampliada con nuevas funcionalidades simplemente utilizando el espacio de nombres std.

**CUIDADO:** la función `main()` no debe definirse dentro de un espacio de nombres. Las cosas que no están definidas en un espacio de nombres existen en el espacio de nombres global, que no tiene nombre.

## Nombres y palabras claves

Variables, constantes, funciones, clases y objetos necesitarán nombre para ser referidos. Existen una serie de reglas que deben ser respetadas y recomendaciones que deberían ser tenidas en cuenta a la hora de nombrar estas entidades. Veamos las reglas de definición de nombre para C++:

- Un nombre puede ser cualquier secuencia de letras mayúsculas o minúsculas de la A a la Z o de la a a la z, los dígitos 0 a 9, y el carácter de subrayado, \_.
- Un nombre debe comenzar con una letra o un guión bajo.

Los nombre son case sensitive, es decir distinguen entre mayúsculas y minúsculas. Por ejemplo no es lo mismo el nombre **suma** que el nombre **Suma**.

## Nombres y palabras claves

No se pueden utilizar como nombres:

Palabras reservadas: son aquellas que están reservadas para el uso del lenguaje y que tienen un significado específico en C++, por ejemplo: `class`, `double`, `throw`, `catch`, `while`, etc.

Nombres que comiencen con 2 guiones bajos consecutivos, por ejemplo:  
`__suma`, `__mediana`, `__contador`, `__counter`, etc.

Nombre que comiencen con un guión bajo seguido de una letra mayúscula, por ejemplo: `_ValorX`, `_Balance`, `_Average`, `_Max`, etc.

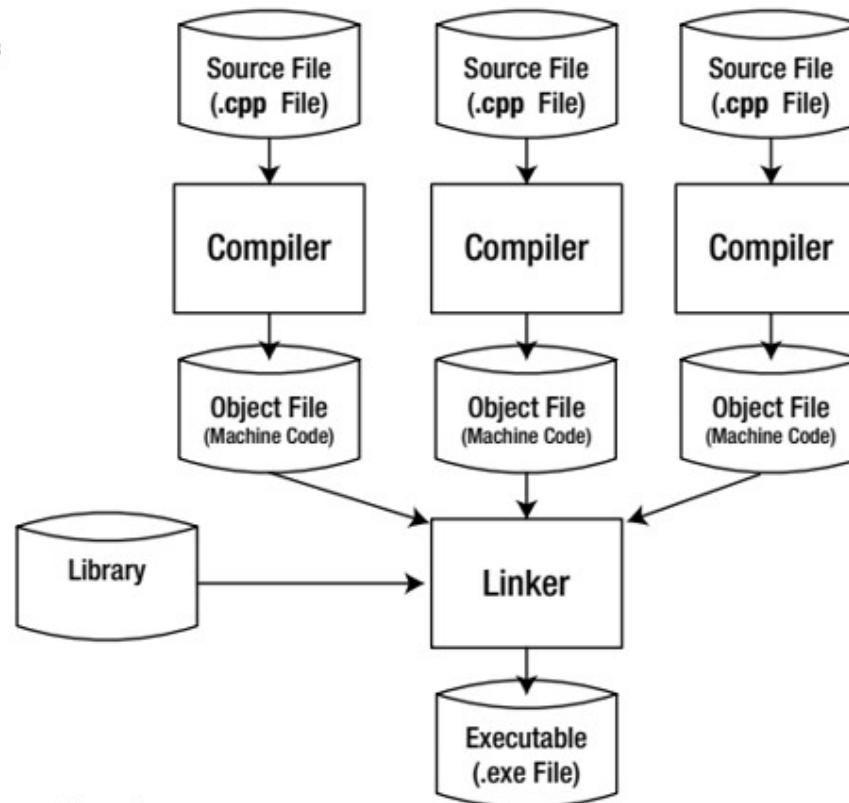
Dentro del espacio de nombres global: todos los nombres que comienzan con un guión bajo

# Construcción de un ejecutable en C++

El contenido de los archivos de cabecera será incluido antes de la compilación.

Cada archivo .cpp resultará en un archivo objeto.

El enlazador combinará todos los archivos objeto más las rutinas de biblioteca necesarias para producir el archivo ejecutable.



*Figura 2: Proceso de compilación y enlazado*



## Tipos de datos fundamentales

Los valores numéricos se dividen en dos grandes categorías: enteros y valores de punto flotante. Hay varios tipos fundamentales de C++ en cada categoría, cada uno de los cuales puede almacenar un rango específico de valores. C++ es un lenguaje **fuertemente tipado**. Esto significa que cada variable o constante que utilicemos en un programa en C++ debe ser declarada (debemos especificar su tipo). La siguiente tabla muestra el conjunto completo de tipos fundamentales que almacenan enteros con signo, es decir, tanto positivos como negativos. La memoria asignada para cada tipo y, por lo tanto, el rango de valores que puede almacenar, puede variar entre diferentes compiladores. Aquí muestra los tamaños y rangos típicos usados por la mayoría de compiladores para las plataformas y arquitecturas informáticas más comunes.



# Tipos de datos fundamentales

## Enteros con signo

*Cuadro 3: Tipos de Entero con signo*

Nombre de tipo	Tamaño Típico (Bytes)	Rango de Valores
bool	1	true o false (1 o 0)
signed char	1	-128 a 127
short	2	-256 a 255
short int		
signed short		
signed short int		
int	4	-2,147,483,648 a +2,147,483,647
signed		
signed int		
long	4 o 8	Mismo que int o long long
long int		
signed long		
signed long int		
long long	8	-9,223,372,036,854,775,808 a
long long int		+9,223,372,036,854,775,807
signed long long		
signed long long int		



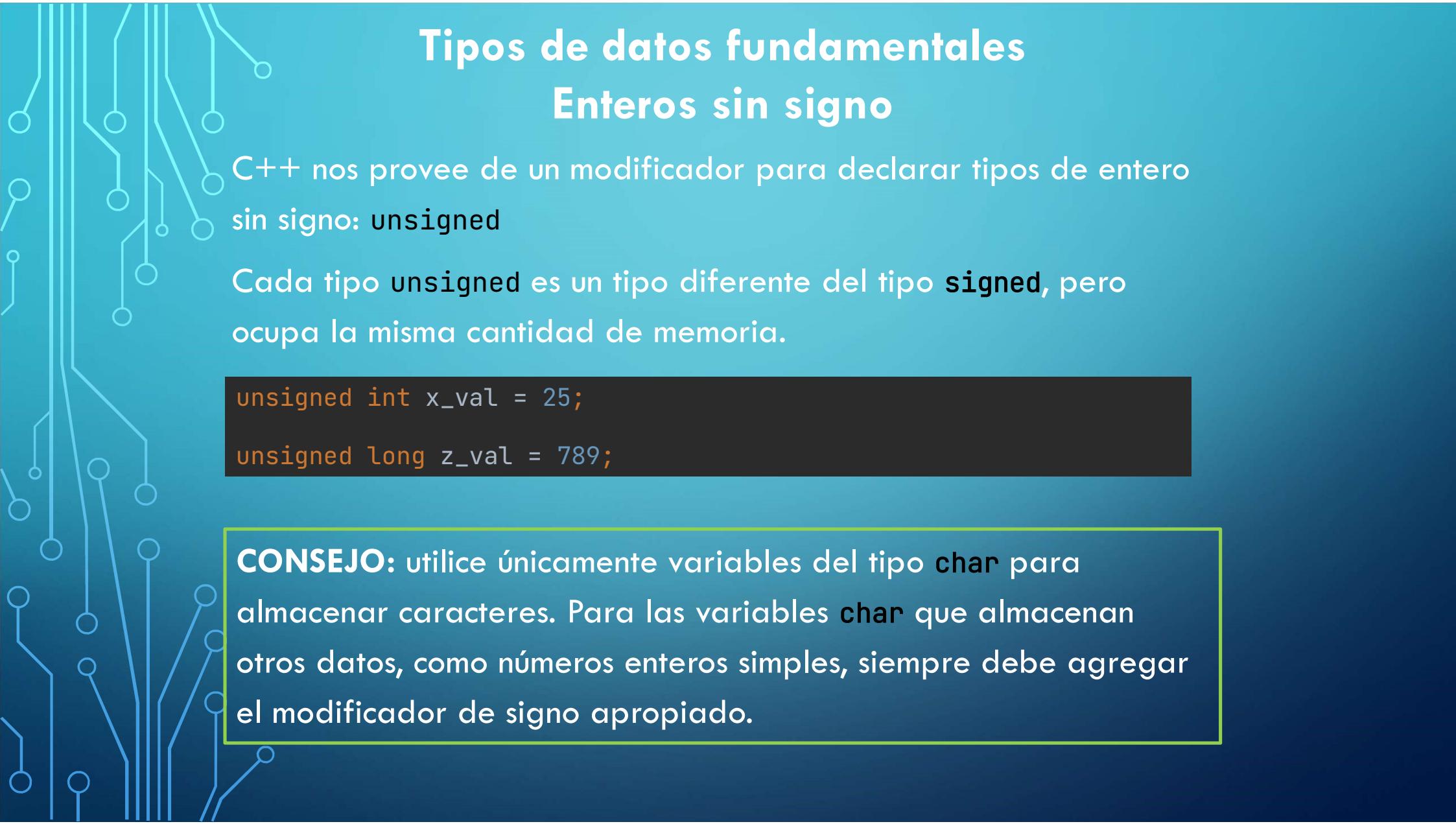
## Tipos de datos fundamentales

### Enteros con signo

El compilador realiza comprobaciones exhaustivas para asegurarse de que utiliza el tipo de datos correcto en cualquier contexto dado. También garantizará que cuando combine diferentes tipos en una operación como, por ejemplo, sumar dos valores, sean del mismo tipo o se puedan hacer compatibles convirtiendo un valor en el tipo del otro (*casting*). El compilador detecta e informa los intentos de combinar datos de diferentes tipos que son incompatibles.

El tipo **signed char** siempre tiene ancho de 1 byte, independientemente del compilador. El ancho de los demás tipos depende del compilador.

```
int suma {};// equivale a signed int suma
long sumaPonderada {};// equivale a signed long
char contador {};// depende de como este configurado el compilador
// si es signed o no
```



## Tipos de datos fundamentales

### Enteros sin signo

C++ nos provee de un modificador para declarar tipos de entero sin signo: `unsigned`

Cada tipo `unsigned` es un tipo diferente del tipo `signed`, pero ocupa la misma cantidad de memoria.

```
unsigned int x_val = 25;  
  
unsigned long z_val = 789;
```

**CONSEJO:** utilice únicamente variables del tipo `char` para almacenar caracteres. Para las variables `char` que almacenan otros datos, como números enteros simples, siempre debe agregar el modificador de signo apropiado.

## Definición e inicialización de variables

Cuando el programa es compilado, el compilador reserva memoria de acuerdo al tipo asignado a la variable y luego establece el valor de esa ubicación de memoria al valor inicial que deseamos.

Existen 3 formas de inicializar una variable. Las veremos a través de un ejemplo:

```
int contador_automóviles (5);      // Notación funcional  
int contadorAutomoviles = 5;       // Notación de asignación.  
int contador_automóviles { 5 };    // Inicialización uniforme
```

## Definición e inicialización de variables

```
int contador_automóviles (5);      // Notación funcional  
int contadorAutomoviles = 5;       // Notación de asignación.  
int contador_automóviles { 5 };     // Inicialización uniforme
```

Hasta C++11 la inicialización se realizaba a través de cualquiera de las 2 primeras formas (funcional o través del operador de asignación `=`). C++11 proporciona una nueva forma de inicializar llamada **inicialización uniforme**.

Una ventaja de esta forma de inicialización es que deduce el tipo de las constantes literales y el compilador nos avisa de un posible error.

## Definición e inicialización de variables

Ejecute las siguientes líneas y observe los resultados:

1. ¿Compila el código?
2. Si no compila ¿Qué errores se presentan?

```
#include <iostream>

int main() {
    int contador_piezas = 3.3;
    int valor_cuenta {3.3};

    std::cout << contador_piezas << " -- " << valor_cuenta << std::endl;
    return 0;
}
```



## Definición e inicialización de variables

Como habrá observado, obtuvo un error de compilación en la línea correspondiente a la inicialización de `valor_cuenta` (si está compilando con el estándar C++11 o superior), esto se debe a que el compilador analiza la constante que queremos asignar a `valor_cuenta`, como esta constante es de tipo float (3.3) y `valor_cuenta` es de tipo int, determina que existe una inconsistencia entre el valor a asignar y la variable a inicializar.

**NOTA:** La inicialización uniforme previene las conversiones de estrechamiento inadvertidas



## Definición e inicialización de variables

### Inicialización a cero

Para inicializar con un valor 0 (cero) una variable de tipo numérico fundamental, las siguientes expresiones son equivalentes:

```
unsigned int contador {0}; // contador comienza en cero  
unsigned int contador {}; // contador comienza en cero
```

Se pueden definir e inicializar más de una variable de un mismo tipo en una misma línea.

```
int contador_clicks {2}, contador_me_gusta {10}, contador_visitas {1};
```

Si bien esto estaría correcto, es recomendable declarar una variable por línea para que podamos colocar comentarios a un lado de ellas para describir su propósito.

## Definición e inicialización de constantes

Al anteponer el nombre de una variable con el modificador **const**, le estamos diciendo al compilador que este valor no podrá ser modificado en el resto del programa. Cualquier intento de modificación disparará un error de compilación.

```
const unsigned int UMBRAL {50} // declara un entero constante con valor 50
```

*Se recomienda utilizar mayúsculas para nombrar constantes, esto mejora la legibilidad ya que al leer un identificador con mayúsculas sabremos que se trata de una constante.*

```
const unsigned int ZOOM {2};  
const unsigned long VISIBLE_STARS { 25'852'963UL }
```

## Literales

Los valores constantes de cualquier tipo, tales como 42, 2.71828, 'Z' o "Mensaje de texto", son conocidos como *literales*. Los valores mostrados corresponden a *literal de tipo entero*, *literal de tipo flotante*, *literal de tipo char* y *literal de tipo string*.

*Ejemplo de literales enteros:*

-258L      +258L      258      22333      98U      1234LL      12345ULL

Los sufijos corresponden a: **U** : unsigned, **L** : long, **LL**: long long.

**NOTA:** Aunque en su mayoría es opcional, hay situaciones en las que necesita agregar los sufijos literales correctos, como cuando inicializa una variable con tipo automático o cuando llama a funciones sobrecargadas con argumentos literales.



## Literales

*Ejemplo de literales enteros:*

-258L

+258L

258

22333

98U

1234LL

12345ULL

Observe que por ejemplo el literal 258 es equivalente a +258. Para mejorar la legibilidad se recomienda anteponer el símbolo + cuando se trate de un valor positivo, nuevamente esto no es una exigencia, es una recomendación. *Los sufijos pueden ser escritos en mayúsculas o minúsculas, pero es recomendable que se escriban con mayúsculas para evitar la confusión con el dígito 1.*

## Literales

A partir de C++14 se introduce una mejora con respecto a la legibilidad de los números. Estos se pueden marcar con el carácter apóstrofe ' (agrupación de dígitos) para separar grupos de dígitos.

*Ejemplos:*

```
22'333           -12'458LL          125'789ULL
```

```
unsigned long montoFinal {99'788UL};  
unsigned short precio {10U};  
long long distancia {15'258'698LL}; // agrupación de dígitos con el  
// carácter ' (C++14)
```

```
unsigned long montoFinal {99'788UL};  
unsigned short precio {10U};  
long long distancia {15'258'698LL}; // agrupación de dígitos con el  
// carácter ' (C++14)
```



## Literales

El separador de dígitos se utiliza, como dijimos, para mejorar la legibilidad, no tiene ninguna injerencia en el valor final de la variable o constante.

Cabe acotar, que podemos utilizar o no la agrupación de dígitos. La agrupación mejora la legibilidad del código. Las siguientes definiciones son equivalentes. Observe cual es más sencilla y rápida de leer.

```
long long distancia {15258698LL};  
long long distancia {15'258'698LL};
```



## Literales

El separador de dígitos se utiliza, como dijimos, para mejorar la legibilidad, no tiene ninguna injerencia en el valor final de la variable o constante.

Cabe acotar, que podemos utilizar o no la agrupación de dígitos. La agrupación mejora la legibilidad del código. Las siguientes definiciones son equivalentes. Observe cual es más sencilla y rápida de leer.

```
long long distancia {15258698LL};  
long long distancia {15'258'698LL};
```

## Literales

*Ejemplo:* Las siguientes declaración producen el mismo efecto.

```
long long speed_of_ligth {299'792'458LL};
```

equivale a escribir:

```
long long speed_of_ligth {2'9979'2458LL};
```

y también equivale a:

```
long long speed_of_ligth {29'97924'58LL};
```

En algunos países asiáticos utilizan grupos de dos dígitos excepto el grupo de tres dígitos más a la derecha. Por ejemplo expresan 15 millones como:

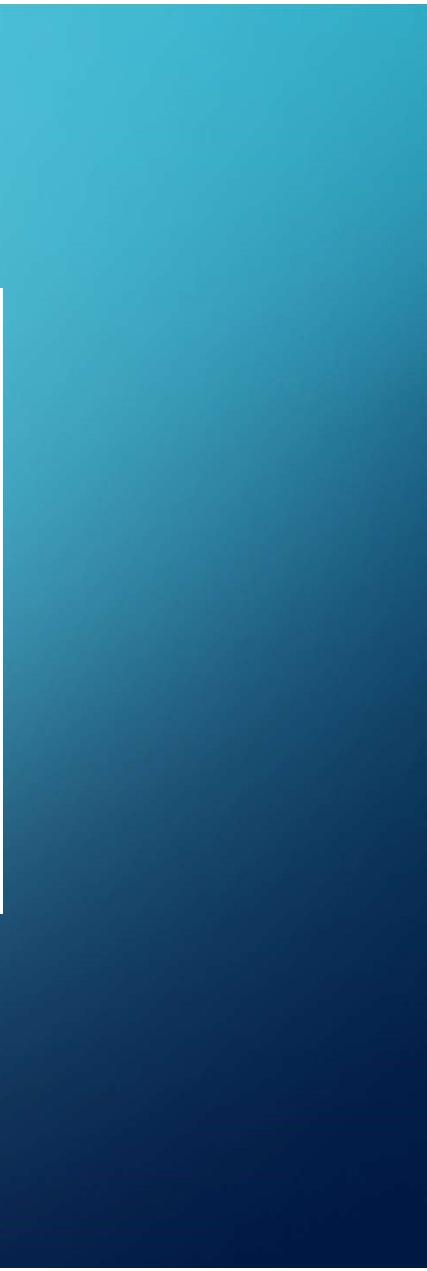
1'50'00'000LL



## Operaciones matemáticas Expresiones y operadores

**Cuadro 6:** Operaciones aritméticas básicas

Operador	Operación
+	Adición
-	Substracción
*	Multiplicación
/	División
%	Módulo (Resto de la división entera)

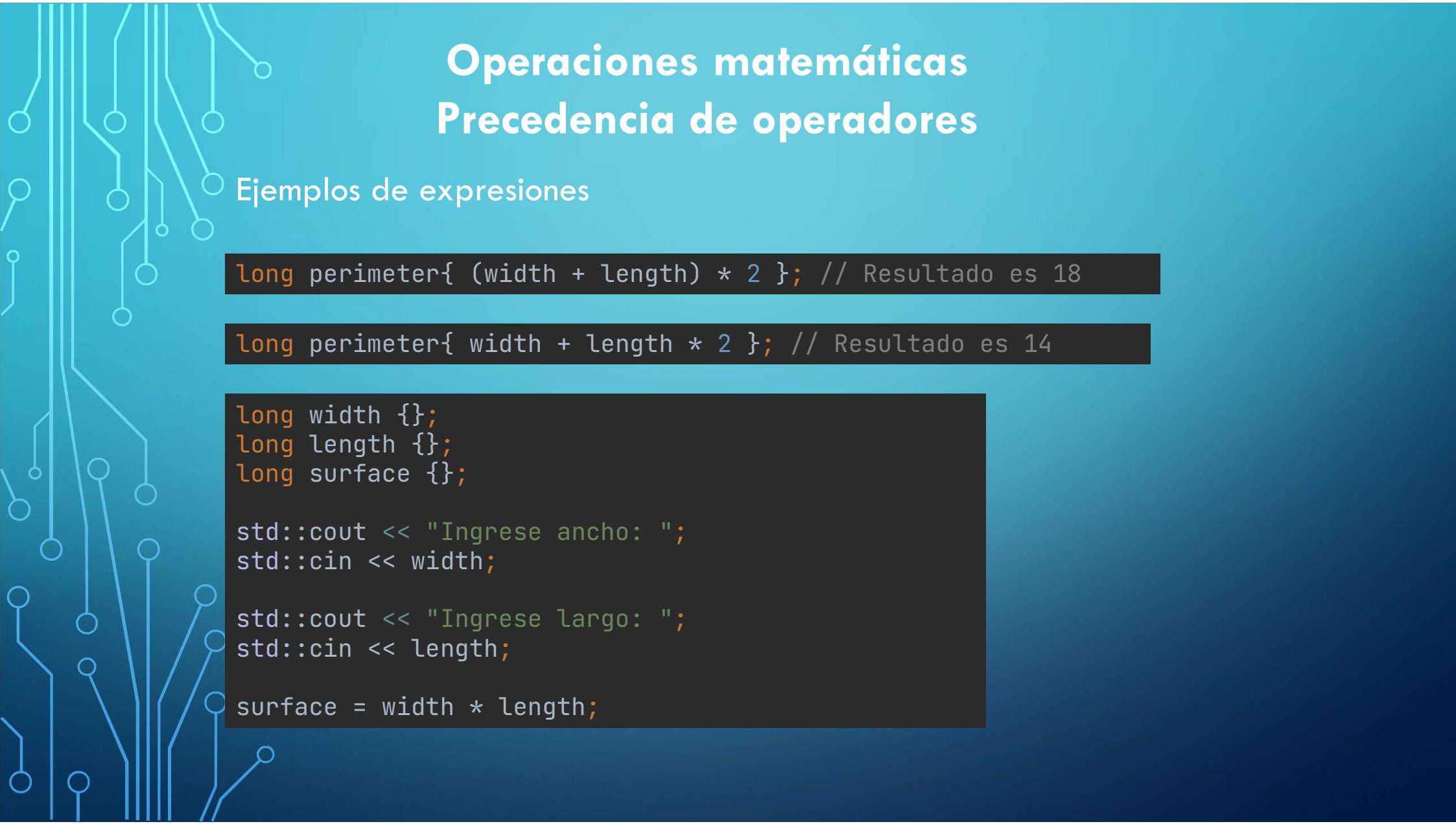


*La operación módulo ( % ) se aplica solo a enteros y nos brinda el resto de la división entre 2 operandos (operador binario).*

# Operaciones matemáticas

## Precedencia de operadores

Operador (es)	Operación (es)	Orden de evaluación (precedencia)
( )	Paréntesis	Evaluado primero. Para paréntesis anidados, como en la expresión $a * (b + c / (d + e))$ , la expresión en el par más interno se evalúa primero. [Precaución: si tiene una expresión como $(a + b) * (c - d)$ en el que dos conjuntos de paréntesis no están anidados, pero aparecen "en el mismo nivel", el estándar C++ no especifica el orden en el que se evaluarán estas subexpresiones entre paréntesis.]
*	Multiplicación	Evaluado segundo. Si hay varios, se evalúan de izquierda a derecha.
/	División	
%	Resto	
+	Suma Resta	Evaluado último. Si hay varios, se evalúan de izquierda a derecha.
-		



# Operaciones matemáticas

## Precedencia de operadores

### Ejemplos de expresiones

```
long perimeter{ (width + length) * 2 }; // Resultado es 18
```

```
long perimeter{ width + length * 2 }; // Resultado es 14
```

```
long width {};
long length {};
long surface {};

std::cout << "Ingrese ancho: ";
std::cin << width;

std::cout << "Ingrese largo: ";
std::cin << length;

surface = width * length;
```

# Operaciones matemáticas

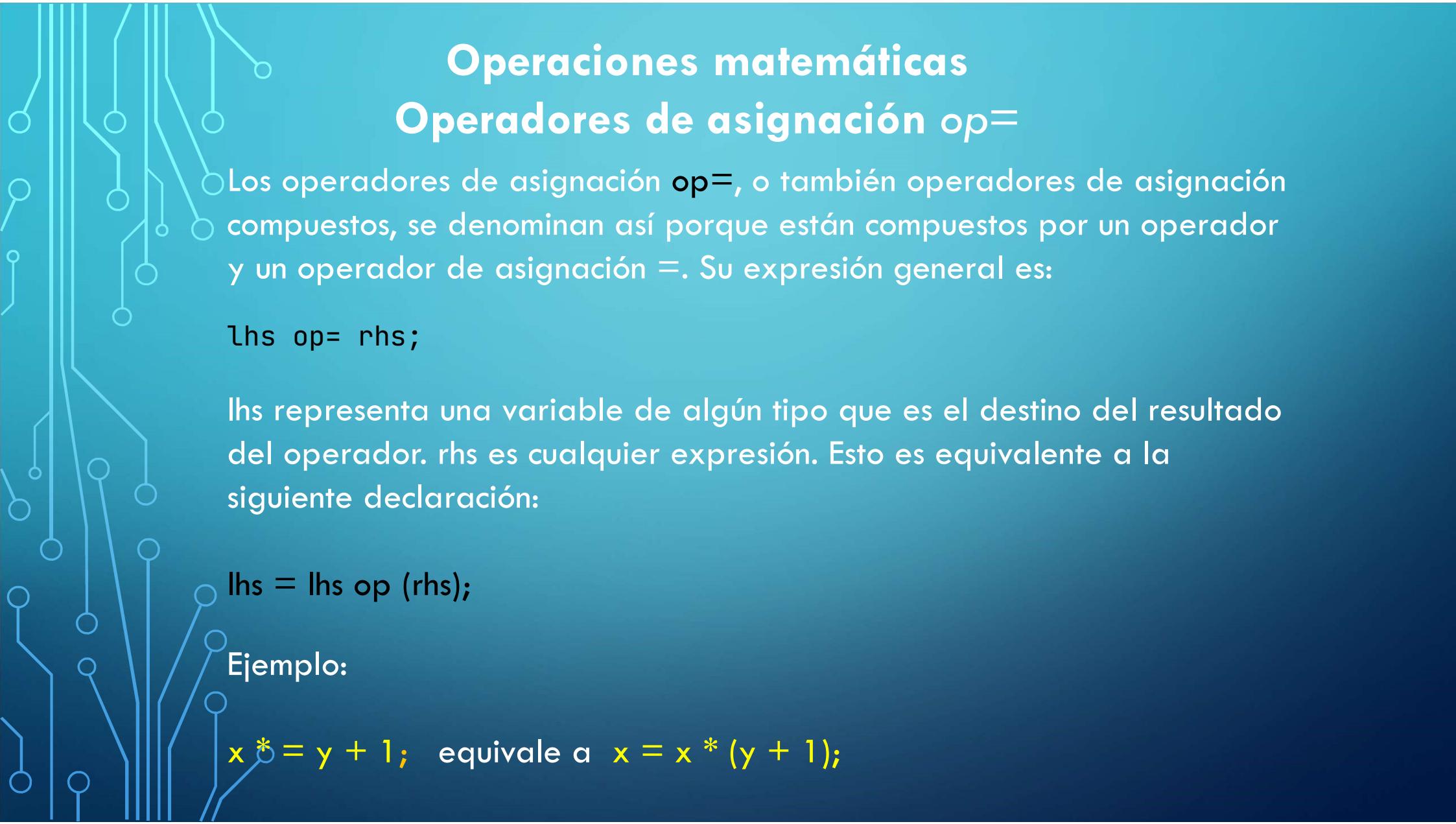
## Asignación

Puede asignar un valor a más de una variable en una sola declaración.  
Aquí hay un ejemplo:

```
int c {5}; // Variable entera c, inicializada a 5.  
int d {4}; // Variable entera c, inicializada a 4.  
  
a = b = c*c - d*d;
```

*Ejercicio:* Realice un programa sencillo que convierta una distancia  
ingresada en yardas, pies, y pulgadas a pulgadas. Utilice las siguientes  
constantes:

```
const unsigned int PIES_POR_YARDA {3};  
const unsigned int PULGADAS_POR_PIE {12};
```



## Operaciones matemáticas

### Operadores de asignación op=

Los operadores de asignación **op=**, o también operadores de asignación compuestos, se denominan así porque están compuestos por un operador y un operador de asignación **=**. Su expresión general es:

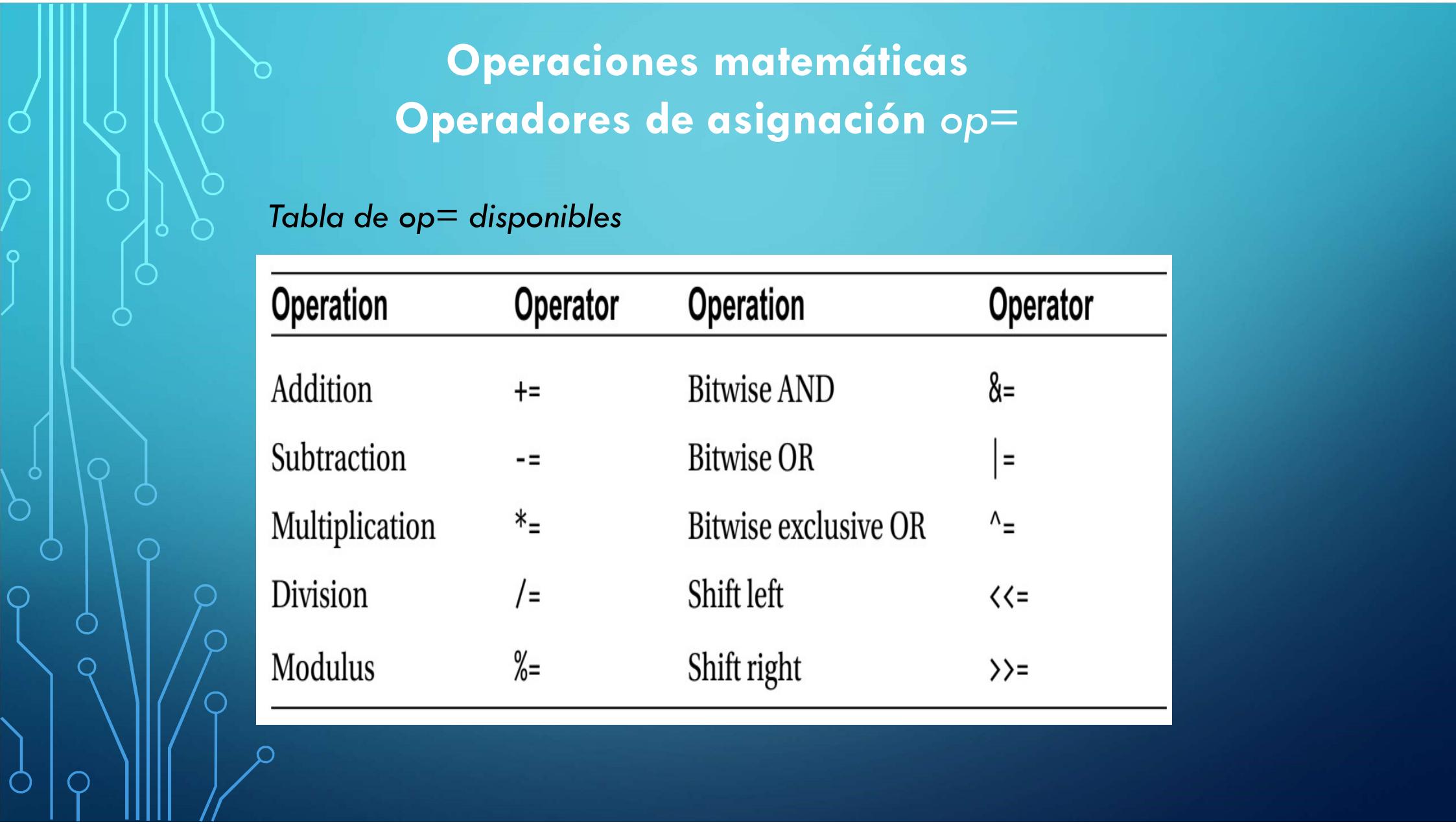
**lhs op= rhs;**

lhs representa una variable de algún tipo que es el destino del resultado del operador. rhs es cualquier expresión. Esto es equivalente a la siguiente declaración:

**lhs = lhs op (rhs);**

Ejemplo:

**x \* = y + 1;** equivale a **x = x \* (y + 1);**



# Operaciones matemáticas

## Operadores de asignación op=

Tabla de op= disponibles

Operation	Operator	Operation	Operator
Addition	$+=$	Bitwise AND	$\&=$
Subtraction	$-=$	Bitwise OR	$ =$
Multiplication	$*=$	Bitwise exclusive OR	$^=$
Division	$/=$	Shift left	$<<=$
Modulus	$\%=$	Shift right	$>>=$

## Incremento y decremento de números enteros

Existen 2 operadores que pueden ser utilizados para cumplir la misma función que los operadores `+=` y `-=`, nos referimos a los operadores de incremento y decremento unarios `++` y `--`.

```
contar = contar + 1;  
contar += 1;  
contar++;  
++contar;
```

Las cuatro expresiones anteriores tienen el mismo efecto, las cuatro incrementan en uno la variable contar. Lo mismo vale para las siguientes tres expresiones de decremento. Usar los operadores de incremento y decremento es claramente el más conciso.

```
contar = contar - 1;  
contar -= 1;  
contar--;  
--contar;
```

# Operaciones de Post Incremento y Post Decremento

En el ejemplo anterior podemos ver 2 formas de aplicar los operadores de incremento y decremento. Para el operador de incremento tenemos:

```
variable++  
++variable
```

La primera forma se llama **forma de sufijo** y la segunda **forma de prefijo**. Si bien en definitiva las dos formas incrementan en uno el valor de la variable, difieren en el instante de tiempo en que se realiza la operación. Veamos un ejemplo:

```
unsigned int total{}, cuenta{2};  
total = cuenta++ + 6;           // total = 8
```

1

```
unsigned int total{}, cuenta{2};  
total = ++cuenta + 6;          // total = 9
```

2

# Operaciones de Post Incremento y Post Decremento

- 1 La forma de sufijo de `++` incrementa la variable a la que se aplica (cuenta) después de que su valor se utiliza en la operación.
- 2 La forma de prefijo de `++` incrementa la variable a la que se aplica (cuenta) antes de que su valor se utilice en la operación.

El mismo razonamiento se aplica al operador de decremento.

```
unsigned int total{}, cuenta{2};  
total = cuenta-- + 6;           // total = 8
```

3

```
unsigned int total{}, cuenta{2};  
total = --cuenta + 6;           // total = 7
```

4

# Operaciones de Post Incremento y Post Decremento

## A TENER EN CUENTA

```
total = ++contar * 3 + contar++ * 5;
```

*El resultado de esta declaración no está definido porque la declaración modifica el valor de cuenta más de una vez usando operadores de incremento. Aunque esta expresión no está definida según el estándar C++, esto no significa que los compiladores no la compilarán. Simplemente significa que no hay ninguna garantía de consistencia en los resultados.*

*Puede recibir un warning de parte del compilador al evaluar esta expresión. Por ejemplo en gcc:*

*Multiple unsequenced modifications to 'contar'*

# Variables de punto flotante

Tipos de datos de punto flotante según su precisión.

*Cuadro 4: Tipos de datos de punto flotante*

Nombre de tipo	Descripción)
float	Valores flotantes de simple precisión
double	Valores flotantes de doble precisión
long double	Valores flotantes de doble precisión extendido



## Variables de punto flotante

- La precisión y el rango de valores no están prescritos por el estándar C++, por lo que lo que obtiene con cada tipo depende de su compilador.
- También dependerá de qué tipo de procesador use su computadora y la representación de punto flotante que utilice.
- En la actualidad todos los compiladores y arquitecturas informáticas utilizan números de punto flotante y aritmética según lo especificado por el estándar IEEE754.

# Variables de punto flotante

Rangos típicos de valores que puede representar con los tipos de punto flotante en un procesador Intel.

*Cuadro 5: Rangos de los tipos de punto flotante*

Tipo	Precisión(Dígitos Decimales)	Rango (+ o -)
float	7	$\pm 1,18 \times 10^{-38}$ a $\pm 1,8 \times 10^{38}$
double	15-16	$\pm 2,22 \times 10^{-308}$ a $\pm 3,4 \times 10^{308}$
long double	18-19	$\pm 3,65 \times 10^{-4932}$ a $\pm 1,18 \times 10^{4932}$

Ejemplos:

```
const float PI {3.1415926f};  
const double PULGADAS_A_MM {25.4};  
double volume {};
```

## Literales de punto flotante

Para escribir literales de punto flotante debe utilizar los siguientes sufijos:

**F o f:** Para números tipo float.

**L:** para long double.

Para escribir literales de tipo double no hace falta ningún sufijo. O sea si obvia el sufijo, el literal es de tipo double.

Ejemplos: 605502764

```
float area_total {256.478F};  
  
double size_unit {12.879456};  
  
long double limit_temp {46558899.856L};
```

# Variables de punto flotante

**Mejores prácticas:** Siempre asegúrese de que el tipo de sus literales coincida con el tipo de las variables a las que se asignan o se utilizan para inicializar. De lo contrario, se producirá una conversión innecesaria, posiblemente con una pérdida de precisión.

**Ejemplo:** Ejecute las siguientes líneas en su compilador y observe los resultados. Debería obtener distintas precisiones de acuerdo al ancho de datos. Recuerde que esto depende del compilador y plataforma. Tip: debe incluir header iomanip (#include <iomanip>)

```
float fval {1.1000000000000002F};  
double fval2 {1.1000000000000002L};  
long double fval3 {1.1000000000000002L};  
  
std::cout << std::setprecision(25);  
std::cout << "fval: " << fval << std::endl;  
std::cout << "fval2: " << fval2 << std::endl;  
std::cout << "fval3: " << fval3 << std::endl;
```

## Variables de punto flotante

*El operador de módulo, %, no se puede usar con operandos de punto flotante, pero todos los demás operadores aritméticos binarios que ha visto, +, -, \* y /, sí se pueden usar. También puede aplicar los operadores de incremento y decremento de prefijo y posfijo, ++ y --, a una variable de punto flotante con esencialmente el mismo efecto que para un número entero; la variable se incrementará o decrementará en 1.0.*

*Ejemplo:* Calcule el volumen de una pizza.

```
const double PI {3.141592653589793};  
double a {15}; // Espesor de la pizza promedio: 15mm  
double z {300}; // Radio de una pizza promedio: 30cm=300mm  
  
double volumenPizza {}; // Volumen de la pizza.  
volumenPizza = PI*z*z*a;
```



## Resultados de punto flotante no válidos

- En el estándar C++, el resultado de la división por cero no está definido.
- Las operaciones de punto flotante en la mayoría de las computadoras se implementan de acuerdo con el estándar IEEE 754. Este define valores especiales **+infinito** o **-infinito**, según el signo para la división de un numero distinto de cero por cero. Otro valor especial de punto flotante definido por este estándar se llama **not-a-number**, generalmente abreviado como **NaN**. Esto representa un resultado que no está definido matemáticamente, como cuando divide cero por cero o infinito por infinito.
- Cualquier operación en la que uno o ambos operandos sean **NaN** da como resultado **NaN**. Una vez que una operación da como resultado  **$\pm\infty$** , esto contaminará todas las operaciones subsiguientes en las que también participe.

## Resultados de punto flotante no válidos

Operation	Result	Operation	Result
$\pm\text{value} / 0$	$\pm\text{infinity}$	$0 / 0$	NaN
$\pm\text{infinity} \pm \text{value}$	$\pm\text{infinity}$	$\pm\text{infinity} / \pm\text{infinity}$	NaN
$\pm\text{infinity} * \text{value}$	$\pm\text{infinity}$	$\text{infinity} - \text{infinity}$	NaN
$\pm\text{infinity} / \text{value}$	$\pm\text{infinity}$	$\text{infinity} * 0$	NaN

**NOTA:** la forma más fácil de obtener un valor de punto flotante que represente **infinito o nan** es usar las funciones del encabezado de límites de la biblioteca estándar, que se analizan más adelante. De esa manera, realmente no tiene que recordar las reglas de cómo obtenerlos a través de divisiones por cero. Para verificar si un número dado es infinito o nan, debe usar las funciones `std::isinf()` y `std::isnan()` proporcionadas por el encabezado `cmath`.

## Resultados de punto flotante no válidos

**Ejemplo:** Ejecute el siguiente ejemplo y observe los resultados. `std::boolalpha` es un manipulador que se aplica al flujo de salida para poder visualizar un valor booleano como `true` o `false` en vez como 1 o 0. Para anular `boolalpha`: `std::noboolalpha`

```
#include <iostream>
#include <iomanip>
#include <cmath>

int main(){

    double a{ 1.5 }, b{}, c{};
    double resultado { a / b };

    std::cout << a << "/" << b << " = " << resultado << std::endl;
    std::cout << resultado << " + " << a << " = " << resultado + a << std::endl;

    resultado = b / c;

    std::cout << b << "/" << c << " = " << resultado << std::endl;
    std::cout << std::boolalpha << resultado << std::endl;

    return 0;
}
```

## Funciones matemáticas

El archivo de encabezado de la biblioteca estándar **cmath** define una gran selección de funciones trigonométricas y numéricas que puede usar en sus programas.

Todos los nombres de funciones definidos están en el espacio de nombres estándar (**std**). A menos que se indique lo contrario, todas las funciones de **cmath** aceptan argumentos que pueden ser de cualquier tipo de punto flotante o entero. El resultado siempre será del mismo tipo que los argumentos de punto flotante dados y de tipo double para argumentos enteros.

**Ejemplo:**

```
double angulo {1.5}; // En radianes
double valor_coseno {std::cos(angulo)};
```

# Funciones matemáticas

Aquí se presentan solo algunas de las muchas existentes.

Function	Description
<code>abs(arg)</code>	Computes the absolute value of <code>arg</code> . Unlike most <code>cmath</code> functions, <code>abs()</code> returns an integer type if <code>arg</code> is integer.
<code>ceil(arg)</code>	Computes a floating-point value that is the smallest integer greater than or equal to <code>arg</code> , so <code>std::ceil(2.5)</code> produces <code>3.0</code> and <code>std::ceil(-2.5)</code> produces <code>-2.0</code> .
<code>floor(arg)</code>	Computes a floating-point value that is the largest integer less than or equal to <code>arg</code> , so <code>std::floor(2.5)</code> results in <code>2.0</code> and <code>std::floor(-2.5)</code> results in <code>-3.0</code> .
<code>exp(arg)</code>	Computes the value of $e^{arg}$ .
<code>log(arg)</code>	Computes the natural logarithm (to base $e$ ) of <code>arg</code> .
<code>log10(arg)</code>	Computes the logarithm to base 10 of <code>arg</code> .
<code>pow(arg1, arg2)</code>	Computes the value of <code>arg1</code> raised to the power <code>arg2</code> , or $arg1^{arg2}$ . <code>arg1</code> and <code>arg2</code> can be integer or floating-point types. The result of <code>std::pow(2, 3)</code> is <code>8.0</code> , <code>std::pow(1.5f, 3)</code> equals <code>3.375f</code> , and <code>std::pow(4, 0.5)</code> is equal to <code>2</code> .
<code>sqrt(arg)</code>	Computes the square root of <code>arg</code> .
<code>round(arg)</code>	Rounds <code>arg</code> to the nearest integer. The result is a floating-point number though, even for integer inputs. The <code>cmath</code> header also defines <code>lround()</code> and <code>llround()</code> that evaluate to the nearest integer of type <code>long</code> and <code>long long</code> , respectively. Halfway cases are rounded away from zero. In other words, <code>std::lround(0.5)</code> gives <code>1L</code> , whereas <code>std::round(-1.5f)</code> gives <code>-2.0f</code> .

# Precedencia de operadores y asociatividad

Precedence	Operators	Associativity
1	::	Left
2	() [] -> . postfix ++ and --	Left
3	! ~ unary + and - prefix ++ and -- address-of & indirection * C-style cast (type) sizeof new new[] delete delete[]	Right
4	.* ->*	Left
5	* / %	Left
6	+ -	Left
7	<< >>	Left
8	< <= > >=	Left
9	== !=	Left
10	&	Left
11	^	Left
12		Left
13	&&	Left
14		Left
15	?:(conditional operator) = *= /= %= += -= &= ^=  = <<= >>= throw	Right
16	, (comma)	Left

## Entrada y salida de datos

Los streams de entrada y salida estándar en C++ se denominan `cout` y `cin`, respectivamente, y por defecto corresponden a la pantalla y el teclado de su computadora. El objeto `cout` permite la salida por pantalla. Ejemplo:

```
std::cout << "La variable answer tiene un valor de: " << answer << std::endl;
```

Todo lo que aparece a la derecha de cada `<<` se transfiere a `cout`. Al insertar `endl` en `std::cout`, se escribe una nueva línea en la secuencia y se vacía el búfer de salida. Vaciar (flush) el buffer de salida significa que el mensaje será mostrado inmediatamente en la salida (en este caso, la pantalla). Podemos escribir el código anterior como:

```
std::cout << "La variable answer tiene un valor de: " // Demo stream de salida  
          << answer  
          << std::endl;
```

# Entrada y salida de datos

## Observaciones sobre std::endl:

Como ya dijimos std::endl inserta un salto de línea y vacía el buffer de salida. Esto permite que el mensaje se muestre inmediatamente en pantalla (en caso de el flujo de salida sea hacia la pantalla). Cada vez que se vacía el búfer, se debe realizar una solicitud al sistema operativo y estas requests son comparativamente caras. Otra alternativa es simplemente insertar un salto de línea y dejar que el sistema operativo decida cuando vaciar el buffer. Es decir por ejemplo:

```
std::cout << "La variable answer tiene un valor de: " << answer << "\n";
```

Cuando la cantidad de operaciones de salidas aumenta considerablemente, el uso de endl reduce el rendimiento. Esto lo podemos ver por ejemplo, cuando escribimos una gran cantidad de datos hacia un archivo

std::endl equivale a '\n' << std::flush;

## Entrada y salida de datos

El objeto `cin` permite obtener valores desde el teclado. Cabe acotar que no pueden leerse cadenas de caracteres utilizando `cin`. Para poder leer cadenas utilizamos el método `getline()` que veremos en un momento. **Ejemplo:**

```
double input {};  
  
std::cout << "Ingrese un valor: " << std::endl;  
std::cin >> input;
```

En el caso de que se ingrese un dato de un tipo diferente al esperado se tratará de convertir dicho dato al tipo adecuado.

### LECTURA DE STRINGS

Para leer strings debemos utilizar el método `std::getline`

## Entrada y salida de datos

```
std::string title, subtitle;  
  
std::cout << "Ingrese titulo documento: " << std::endl;  
std::cin >> title;  
  
// Las siguiente linea limpia el buffer cin  
// para eliminar ENTERS remanentes en el buffer de entrada.  
std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');  
  
std::cout << "Ingrese subtítulo documento: " << std::endl;  
std::getline(std::cin, subtitle);  
  
std::cout << "Titulo doc: " << title << ", subtítulo: "  
      << subtitle << std::endl;
```

Existe otro método con el mismo nombre pero destinado a leer cadenas de tipo C. Lo veremos al explorar el tema arrays. Cuando profundicemos en la clase string veremos en detalle el método `std::getline`, sus variantes y otros métodos útiles.



# Entrada y salida de datos

*Cuadro 1: Secuencias de Escape que representan Caracteres de Control*

Secuencia de Escape	Carácter de Control
\n	Salto de línea
\t	Tab horizontal
\v	Tab vertical
\b	Retroceso
\r	Retorno de carro
\f	Avance de página
\a	Alerta/Campana

## Entrada y salida de datos

Dado que en C++ algunos caracteres tienen un significado especial, existe un mecanismo que nos permite utilizarlos en nuestros programas. Para poder utilizar dichos caracteres, debemos hacer uso de las **secuencias de escape**. Un ejemplo típico es el uso de las comillas dentro de un mensaje a imprimir, dado que las comillas sirven de delimitadores de **string**, debemos utilizar la representación con la secuencia de escape correspondiente.

*Cuadro 2: Secuencias de Escape que representan Caracteres Problemáticos*

Secuencia de Escape	Caracter Problematico
\\	Barra invertida
\'	Comilla simple
\"	Comilla doble

## Formateando flujos (streams) de salida



Puede cambiar el formato de los datos cuando se escriben en un flujo de salida mediante manipuladores de flujo, que se declaran en los encabezados de la biblioteca estándar de **iomanip** e **ios**. Un manipulador de flujo, se aplica a un flujo de salida con el operador de inserción, <<. Solo presentaremos los manipuladores más útiles. Debe consultar una referencia de la Biblioteca estándar si desea conocer los demás. Todos los manipuladores declarados por **ios** están disponibles automáticamente si incluye el encabezado **iostream**. A diferencia de los del encabezado **iomanip**, estos manipuladores de flujo no requieren un argumento.

# Formateando flujos (streams) de salida

<code>std::fixed</code>	Output floating-point data in fixed-point notation.
<code>std::scientific</code>	Output all subsequent floating-point data in scientific notation, which always includes an exponent and one digit before the decimal point.
<code>std::defaultfloat</code>	Revert to the default floating-point data presentation.
<code>std::dec</code>	All subsequent integer output is decimal.
<code>std::hex</code>	All subsequent integer output is hexadecimal.
<code>std::oct</code>	All subsequent integer output is octal.
<code>std::showbase</code>	Outputs the base prefix for hexadecimal and octal integer values. Inserting <code>std::noshowbase</code> in a stream will switch this off.
<code>std::left</code>	Output is left-justified in the field.
<code>std::right</code>	Output is right-justified in the field. This is the default.

# Formateando flujos (streams) de salida

El encabezado **iomanip** también proporciona manipuladores paramétricos útiles, algunos de los cuales se enumeran a continuación. Para usarlos, primero debe incluir el encabezado **iomanip** en su archivo fuente.

<code>std::setprecision(n)</code>	Sets the floating-point precision or the number of decimal places to <code>n</code> digits. If the default floating-point output presentation is in effect, <code>n</code> specifies the number of digits in the output value. If fixed or scientific format has been set, <code>n</code> is the number of digits following the decimal point. The default precision is 6.
<code>std::setw(n)</code>	Sets the output field width to <code>n</code> characters, but only for the next output data item. Subsequent output reverts to the default where the field width is set to the number of output character needed to accommodate the data.
<code>std::setfill(ch)</code>	When the field width has more characters than the output value, excess characters in the field will be the default fill character, which is a space. This sets the fill character to be <code>ch</code> for all subsequent output.

# Formateando flujos (streams) de salida

Ejemplo: Realice un programa que imprima una tabla similar a la mostrada.

Ayuda: Puede utilizar la clase string para generar una línea de la siguiente

manera: `std::string linea(31, '-');` Esto genera una línea de 31 caracteres. Utilice el manipulador de flujo `std::setw(n)`.

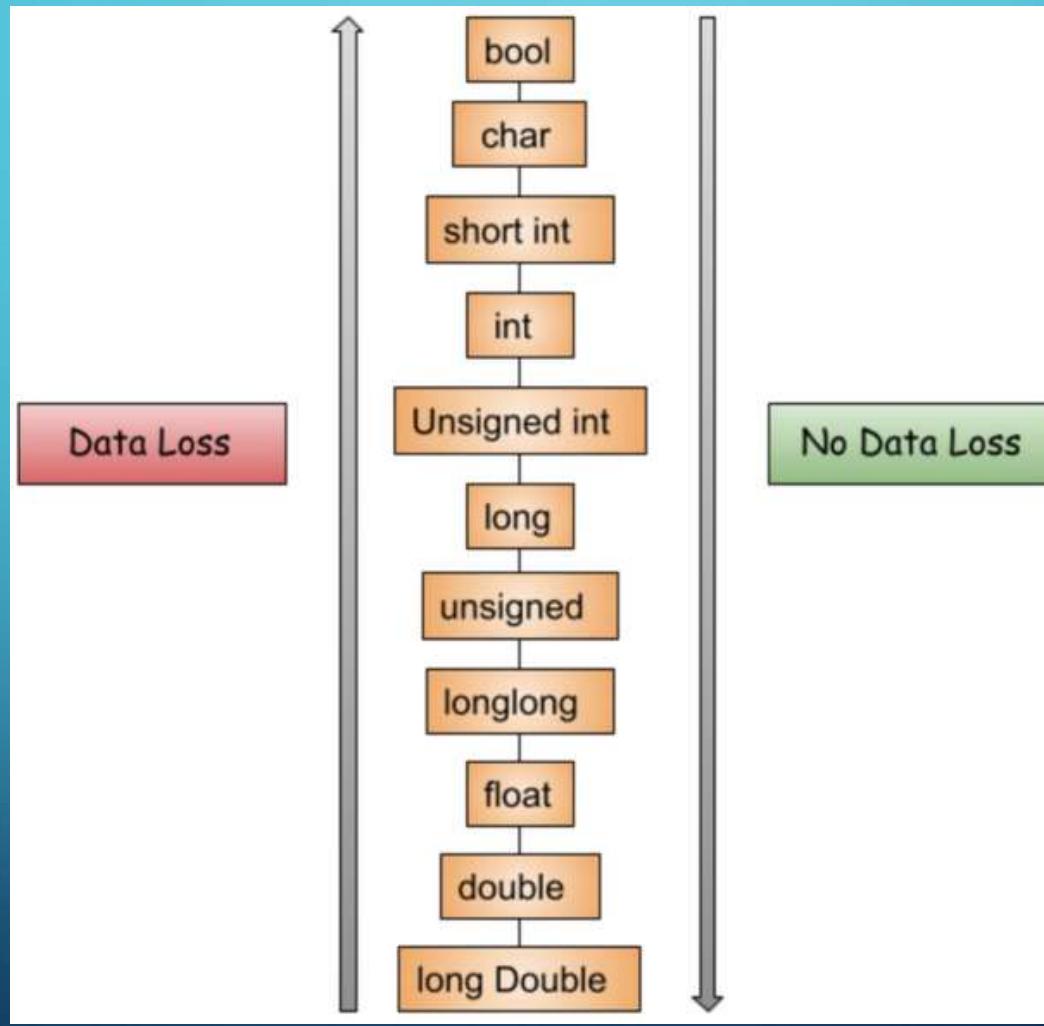
Movimientos Anuales		
	Entrada	Salida
Enero		
Febrero		
Marzo		

## Conversiones implícitas

Técnicamente, todos los operandos aritméticos binarios requieren que ambos operandos sean del mismo tipo. Sin embargo, cuando este no sea el caso, el compilador se encargará de convertir uno de los valores del operando al mismo tipo que el otro. Estas son **conversiones implícitas**. La forma en que esto funciona es que la variable de un tipo con el rango más limitado se convierte al tipo del otro. La variable `factor_long` en la primera declaración es de tipo `double`. El tipo `double` tiene un rango mayor que el tipo `unsigned int`, por lo que el compilador insertará una conversión para el valor de `cant_unidades` a tipo `double` para que sea del mismo tipo `factor_long` antes de que se ejecute la operación de multiplicación.

```
const double factor_long {25.789};  
unsigned int cant_unidades {50};  
  
double longitud_total = factor_long * cant_unidades;
```

# Conversiones implícitas



## Conversiones implícitas

El operando a convertir será el de menor rango. Así, en una operación con operandos de tipo `long long` y tipo `unsigned int`, este último se convertirá a tipo `long long`. Un operando de tipo `char`, `signed char`, `unsigned char`, `short` o `unsigned short` siempre se convierte al menos al tipo `int`. Las conversiones implícitas pueden producir resultados inesperados. *Ejemplo:*

```
unsigned int x {20u};  
int y {30};  
  
std::cout << x - y << std::endl;
```

Puede esperar que la salida sea `-10`, pero no lo es. ¡Lo más probable es que la salida sea `4294967286`! Esto se debe a que el valor de `y` se convierte en entero sin signo (`unsigned int`) para que coincida con el tipo de `x`, por lo que el resultado de la resta es un valor entero sin signo. Y `-10` no puede ser representado por un tipo sin signo.

## Conversiones explícitas

Cuando tenga operandos de diferente tipo **no confíe en la conversión de tipo implícita** para producir el resultado que desea a menos que esté seguro de que lo hará. Si no está seguro, lo que necesita es una conversión de tipo explícita, también llamada conversión explícita.

Este tipo de conversión se utiliza para forzar la conversión por estrechamiento. La expresión general de una conversión explícita es:

```
static_cast<tipo_al_que_se_desea_convertir>(expresion)
```

La palabra clave **static\_cast** refleja el hecho de que la conversión se comprueba de forma estática, es decir, cuando se compila el código.

## Conversiones explícitas

*Ejemplo 1:* El problema del ejemplo anterior podría solucionarse de la siguiente manera:

```
unsigned int x {20u};  
int y {30};  
  
std::cout << (static_cast<int>(x) - y) << std::endl;
```

*Ejemplo 2:* Observe las siguientes líneas:

```
double valor1 {10.9};  
double valor2 {15.9};  
int suma_entera {static_cast<int>(valor1) + static_cast<int>(valor2)};
```

¿Qué ocurre si reescribimos la expresión anterior de la siguiente manera?

```
int suma_entera {static_cast<int>(valor1 + valor2)};
```

## Conversiones explícitas

Antes de la introducción de `static_cast<>` en C++ alrededor de 1998, una conversión explícita del resultado de una expresión se escribía así:

`(tipo_al_que_se_desea_convertir)expression`

Si participa en proyectos con código heredado, quizás pueda encontrar este tipo de conversión explícita. Este tipo de cast se llama **cast de estilo C**, dado que proviene de dicho lenguaje. *Ejemplo:*

```
double valor1 {10.9};  
double valor2 {15.9};  
int suma_entera {(int)(valor1) + (int)(valor2)};
```

Hay varios tipos de conversiones en C++ que ahora se diferencian, pero la sintaxis de conversión al estilo antiguo las cubre todas. Debido a esto, el código que usa las conversiones de estilo antiguo es más propenso a errores. *Recomendamos no utilizar este estilo de casting en sus programas.*

# Valores límite de los distintos tipos de datos

El encabezado de la biblioteca estándar `limits` hace que la información acerca de los valores límite esté disponible para todos los tipos de datos fundamentales para que pueda acceder a ella para su compilador.

`std::numeric_limits<tipo>::max()`  
`std::numeric_limits<tipo>::min()`

## Ejemplos:

```
std::cout << "Min tipo int:" << std::numeric_limits<unsigned int>::min()<< std::endl;
std::cout << "Max tipo int:" << std::numeric_limits<unsigned int>::max()<< std::endl;

std::cout << "Min tipo unsigned int:" << std::numeric_limits<unsigned int>::min()<< std::endl;
std::cout << "Max tipo unsigned int:" << std::numeric_limits<unsigned int>::max()<< std::endl;

unsigned long limite_Ulong {std::numeric_limits<unsigned long>::max()};
```

# Inferencia de tipos

A partir de C++11 se incorpora la palabra clave **auto**, que permite inferir el tipo de datos en base a su valor.

```
std::numeric_limits<tipo>::max()  
std::numeric_limits<tipo>::min()
```

*Ejemplos:*

```
auto m {10};           // m es de tipo int  
auto n {200UL};        // n es de tipo unsigned long  
auto pi {3.14159};      // pi es de tipo double  
  
auto suma = 0UL;  
auto flag = true;
```

# Inferencia de tipos

A tener en cuenta:

```
auto m = {10};
```

El tipo deducido para m no será int, sino std::initializer\_list<int>

Para darle algo de contexto, este es el mismo tipo que obtendría si usara una lista de elementos entre llaves:

```
auto list = {1, 2, 3}; // la lista tiene tipo std::initializer_list<int>
```

Tales listas se usan normalmente para especificar los valores iniciales de contenedores como std::vector<>. Las reglas de deducción de tipos han cambiado en C++17. Si está utilizando un compilador más antiguo, el tipo que el compilador deduce en lugar de automático puede no ser lo que esperaría en muchos más casos.

# Inferencia de tipos

```
/* C++11 y C++14 */
auto i {10};           // i es de tipo std::initializer_list<int> !!!
auto pi = {3.14159}; // pi es de tipo std::initializer_list<double>
auto list1{1, 2, 3}; // list1 es de tipo std::initializer_list<int>
auto list2 = {4, 5, 6}; // list2 es de tipo std::initializer_list<int>

/* C++17 y posteriores */
auto i {10};           // i es de tipo int
auto pi = {3.14159}; // pi es de tipo std::initializer_list<double>
auto list1 {1, 2, 3}; // error: no compila!!
auto list2 = {4, 5, 6}; // list2 es de tipo std::initializer_list<int>
```

Si su compilador es compatible correctamente con C++ 17, puede usar la inicialización entre llaves para inicializar cualquier variable con un solo valor, siempre que no la combine con una asignación. Sin embargo, si su compilador aún no está completamente actualizado, simplemente nunca debe usar inicializadores uniformes con **auto**. En su lugar, establezca explícitamente el tipo o use la asignación o la notación funcional.



## Ejercitación

**Ejercicio:**

Suponga que desea construir un estanque circular en el que tendrá peces. Habiendo investigado el asunto, sabe que debe permitir 0.2 metros cuadrados de área de superficie del estanque por cada 0.15 metros de largo del pez. Necesita averiguar el diámetro del estanque que mantendrá felices a los peces.



## Ejercitación

1 - Escriba un programa para calcular cuántas cajas cuadradas se pueden contener en una sola capa en un estante rectangular. Las dimensiones del estante en pies y la dimensión de un lado de la caja en pulgadas se leen del teclado. Use variables de tipo **double** para el largo y la profundidad del estante y tipo **int** para el largo del lado de una caja. Defina e inicialice una constante entera para convertir de pies a pulgadas (1 pie equivale a 12 pulgadas). Calcule la cantidad de cajas (tipo **long**) que el estante puede contener en una sola capa e imprima el resultado.

```
const int PULGADAS_POR_PIE {12};
```

## El operador sizeof

Operador que permite obtener el número de bytes que es ocupado por un tipo. Sintaxis:

`sizeof(tipo_dato)`

*Ejemplos:*

`sizeof(int) // da como resultado 4, ya que un int ocupa 4 bytes.`

```
int altura {74};

std::cout << "La variable altura ocupa: " << sizeof(altura)
             << " bytes." << std::endl;

std::cout << "El tipo \"long long\" ocupa: " << sizeof(long long)
             << " bytes." << std::endl;

std::cout << "El resultado de la expresion * altura * altura /2 ocupa: "
             << sizeof(altura * altura/2) << " bytes." << std::endl;
```

## El operador sizeof

Puede aplicar `sizeof` a cualquier tipo fundamental, tipo de clase o tipo de puntero (lo veremos más adelante). El resultado que produce `sizeof` es del tipo `size_t`, que es un tipo de entero sin signo que se define en el encabezado `cstdint` de la biblioteca estándar. El tipo `size_t` está definido por la implementación (`size_t` es *un entero sin signo cuyo tamaño sea suficiente para contener el tamaño de cualquier objeto en bytes*).

Puede utilizar `std::is_same_v` para saber si `size_t` es de un tipo específico.

*Ejemplo:*

```
std::cout << std::boolalpha << std::is_same_v<size_t, unsigned long long>
          << std::endl;
```

## Enumeraciones

Un tipo de datos enumerado es una manera de asociar nombres a números, y por consiguiente de ofrecer más significado a alguien que lea el código, es un tipo definido por el usuario y consiste en un conjunto de nombres constantes llamados enumeradores. Un **tipo de datos enumerado** es útil cuando se quiere poder seguir la pista de alguna característica.

**Ejemplo:** Vamos a crear un ejemplo utilizando una de las ideas que acabamos de mencionar: un tipo de variables que pueden asumir valores correspondientes a los días de la semana

```
enum class Dia {Lunes, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo};
```

Esto define un tipo de dato enumerado llamado Dia, y las variables de este tipo solo pueden tener valores del conjunto que aparece entre llaves, de Lunes a Domingo. Si intenta establecer una variable de tipo Dia en un valor que no es uno de estos valores, el código no se compilará. Los nombres simbólicos entre llaves se denominan enumeradores.

## Enumeraciones

Un tipo de datos enumerado es una manera de asociar nombres a números, y por consiguiente de ofrecer más significado a alguien que lea el código, es un tipo definido por el usuario y consiste en un conjunto de nombres constantes llamados enumeradores. Un **tipo de datos enumerado** es útil cuando se quiere poder seguir la pista de alguna característica.

**Ejemplo:** Vamos a crear un ejemplo utilizando una de las ideas que acabamos de mencionar: un tipo de variables que pueden asumir valores correspondientes a los días de la semana

```
enum class Dia {Lunes, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo};
```

Esto define un tipo de dato enumerado llamado Dia, y las variables de este tipo solo pueden tener valores del conjunto que aparece entre llaves, de Lunes a Domingo. Si intenta establecer una variable de tipo Dia en un valor que no es uno de estos valores, el código no se compilará. Los nombres simbólicos entre llaves se denominan enumeradores.

## Enumeraciones

Cada enumerador se definirá automáticamente para tener un valor entero fijo de tipo int de forma predeterminada. El primer nombre de la lista, Lunes, tendrá el valor 0, Martes será 1, y así sucesivamente, hasta Domingo con valor 6. Puede definir hoy como una variable del tipo de enumeración Dia con la siguiente instrucción:

```
Dia hoy {Dia::Martes};
```

La siguiente línea imprimirá 1.

```
std::cout << "Hoy es " << static_cast<int>(hoy) << std::endl;
```

Por defecto, el valor de cada **enumerador** es uno mayor que el anterior y, de forma predeterminada, los valores comienzan en 0. Sin embargo, puede hacer que los valores implícitos asignados a los enumeradores comiencen en un valor entero diferente.

```
enum class Dia {Lunes=1, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo};
```

## Enumeraciones

Puede asignar cualquier valor entero que desee a los enumeradores, y la asignación de estos valores tampoco se limita a los primeros enumeradores

```
enum class Dia {Lunes = 3, Martes, Miercoles, Jueves, Viernes,  
    Sabado = 1, Domingo};
```

¿Qué valor tiene el enumerador Domingo?

Los enumeradores ni siquiera necesitan tener valores únicos. Podría definir el lunes y el lunes con el valor 1, por ejemplo, así:

```
enum class Dia {Lunes = 1, Lun = 1, Martes, Miercoles, Jueves, Viernes,  
    Sabado, Domingo };
```

Ahora puede usar Lun o Lunes como el primer día de la semana. Una variable **ayer**, que haya definido como tipo Dia podría establecerse con esta declaración:

```
Dia ayer = Dia::Lun;
```

# Enumeraciones

También puede definir enumeradores en base a enumeradores previamente definidos:

```
enum class Dia { Lunes, Mon = Lunes,
    Martes = Lunes + 2, Mart = Martes,
    Miercoles = Martes + 2, Mier = Miercoles,
    Jueves = Miercoles + 2, Juev = Jueves,
    Viernes = Jueves + 2, Vier = Viernes,
    Sabado = Viernes + 2, Sab = Sabado,
    Domingo = Sabado + 2, Sun = Domingo
};
```

Los valores de los enumeradores deben ser **constantes de tiempo de compilación**, es decir, expresiones constantes que el compilador pueda evaluar. Dichas expresiones incluyen literales, enumeradores que se han definido previamente y variables que ha especificado como const. **No puede usar variables que no sean constantes, incluso si las ha inicializado usando un literal.**

## Enumeraciones

Los enumeradores pueden ser un tipo entero que elija, en lugar del tipo predeterminado `int`. También puede asignar valores explícitos a todos los enumeradores. Por ejemplo, podría definir esta enumeración:

```
enum class Puntuacion: char {Coma = ',', Exclamacion = '!', Pregunta = '?'};
```

La especificación de tipo para los enumeradores va después del nombre del tipo de enumeración y está separada por dos puntos. Puede especificar cualquier tipo de datos enteros para los enumeradores.

### Ejercicio:

Imprima el siguiente mensaje utilizando de alguna manera la enumeración del ejemplo anterior:

Hoy es martes?. Si que lo es, y es un buen dia!

# Enumeraciones

En versiones anteriores de C++, las enumeraciones tenían otro estilo (estilo C) para declarar enumeraciones. Por ejemplo la enumeración **Dia**, podía definirse así:

```
enum Dia {Lunes, Martes, Miercoles, Jueves, Vienes, Sabado, Domingo};
```

El nuevo estilo es menos propenso a errores. Aunque el viejo estilo puede seguir siendo usado, recomendamos utilizar el nuevo estilo **enum class**. Los enumeradores de estilo antiguo se convierten a valores de tipo entero o incluso de punto flotante sin conversión, lo que puede conducir fácilmente a errores.

# Enumeraciones

Transcriba estas líneas, quite casting realizado para `saludo2::tal` y observe que ocurre.

```
enum saludo{hola, que, tal};  
enum class saludo2{hola, que, tal};  
  
std::cout << saludo::tal << std::endl;  
std::cout << static_cast<int>(saludo2::tal) << std::endl;
```

Recuerde que cuando vimos conversión explícita dijimos que no es recomendable dejar que el compilador realice conversiones implícitas dado que es una fuente de bugs, es por ello que `enum class` exige la conversión explícita a través de `static_cast` cuando utilizamos los enumeradores.

## Alias de tipos de datos

La palabra clave **using** le permite especificar un alias de tipo, que es su propio nombre de tipo de datos que sirve como alternativa a un nombre de tipo existente. Usando **using**, puede definir el tipo alias **BigOnes** como equivalente al tipo estándar **unsigned long long** con la siguiente declaración:

```
using BigOnes = unsigned long long;
```

**BigOnes** no define un nuevo tipo, es un nombre alternativo para el tipo **unsigned long long**.

**using** se introdujo en C++11, previamente (y todavía puede utilizarse), el alias se definía a través de la palabra clave **typedef**. Utilizando **typedef**, el alias anterior puede definirse como:

```
typedef unsigned long long BigOnes;
```

## Alias de tipos de datos

La razón de la incorporación de `using` (existen otras razones), fue de proveer claridad al código. Observe la declaración del alias con `typedef`, se hace escribiendo primero el tipo para el cual queremos definir el alias, mientras que usando `using`, primero se escribe el alias lo cual es más natural. Cualquiera de las 2 definiciones es válida. Si está modificando código heredado y este utiliza `typedef`, siga utilizando `typedef`. Por el contrario, si va a crear código nuevo en C++11 o posteriores, recomendamos utilizar `using` para definir alias.

```
BigOnes myVal {}; // Define e inicializa como tipo unsigned long long
```

## Alias de tipos de datos

Un uso importante de los alias es simplificar el código que involucra nombres de tipos complejos. Por ejemplo, un programa puede incluir un nombre de tipo como:

```
using AgendaTelefonica = std::map<std::shared_ptr<Contacto>, std::string>
```

El uso de `AgendaTelefonica` en el código en lugar de la especificación de tipo completo puede hacer que el código sea más legible. Otro uso para un alias de tipo es proporcionar flexibilidad en los tipos de datos utilizados por un programa que puede necesitar ejecutarse en una variedad de computadoras. Definir un alias de tipo y usarlo en todo el código permite modificar el tipo real simplemente cambiando la definición del alias.

De todas maneras, no debe abusarse del uso del alias. Usarlo demasiado en un programa puede ir en contra de la legibilidad del código.



## Tiempo de vida de una variable

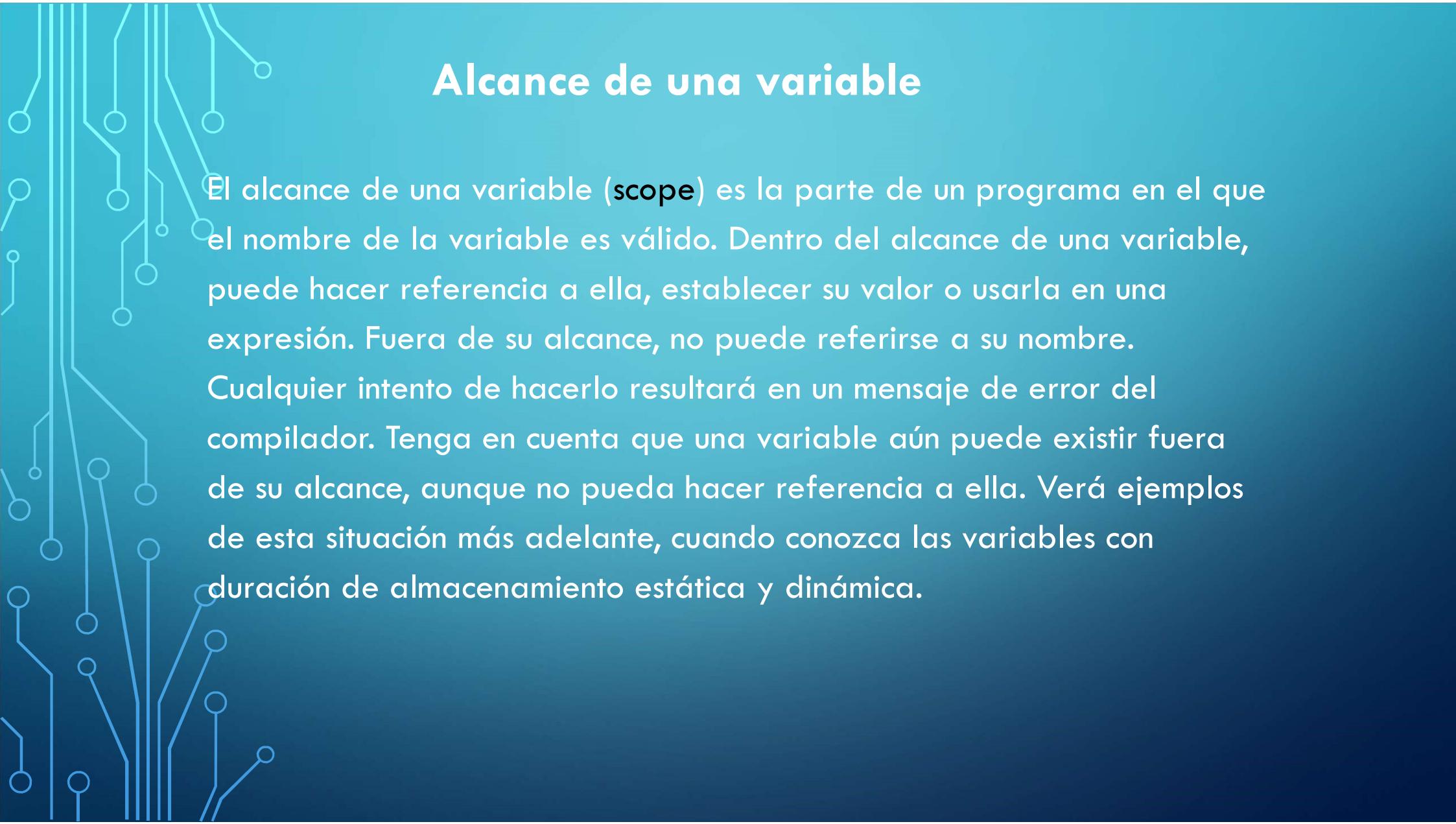
Hay cuatro tipos diferentes de duración de almacenamiento:

- Las variables definidas dentro de un bloque que no están definidas para ser estáticas tienen una duración de **almacenamiento automática**. Existen desde el punto en el que se definen hasta el final del bloque, que es la llave de cierre, **्**. Se denominan variables automáticas o variables locales. Se dice que las variables automáticas tienen alcance local o alcance de bloque.
- Las variables definidas con la palabra clave **static** tienen una **duración de almacenamiento estática**. Se llaman variables estáticas. Las variables estáticas existen desde el punto en que se definen y continúan existiendo hasta que finaliza el programa.



## Tiempo de vida de una variable

- Las variables para las que asigna memoria en tiempo de ejecución tienen una **duración de almacenamiento dinámica**. Existen desde el momento en que las creas hasta que liberas su memoria para destruirlos.
- Las variables declaradas con la palabra clave **thread\_local** tienen una **duración de almacenamiento de subprocesos**. No las trataremos en este curso.



## Alcance de una variable

El alcance de una variable (**scope**) es la parte de un programa en el que el nombre de la variable es válido. Dentro del alcance de una variable, puede hacer referencia a ella, establecer su valor o usarla en una expresión. Fuera de su alcance, no puede referirse a su nombre. Cualquier intento de hacerlo resultará en un mensaje de error del compilador. Tenga en cuenta que una variable aún puede existir fuera de su alcance, aunque no pueda hacer referencia a ella. Verá ejemplos de esta situación más adelante, cuando conozca las variables con duración de almacenamiento estática y dinámica.



## Alcance de una variable

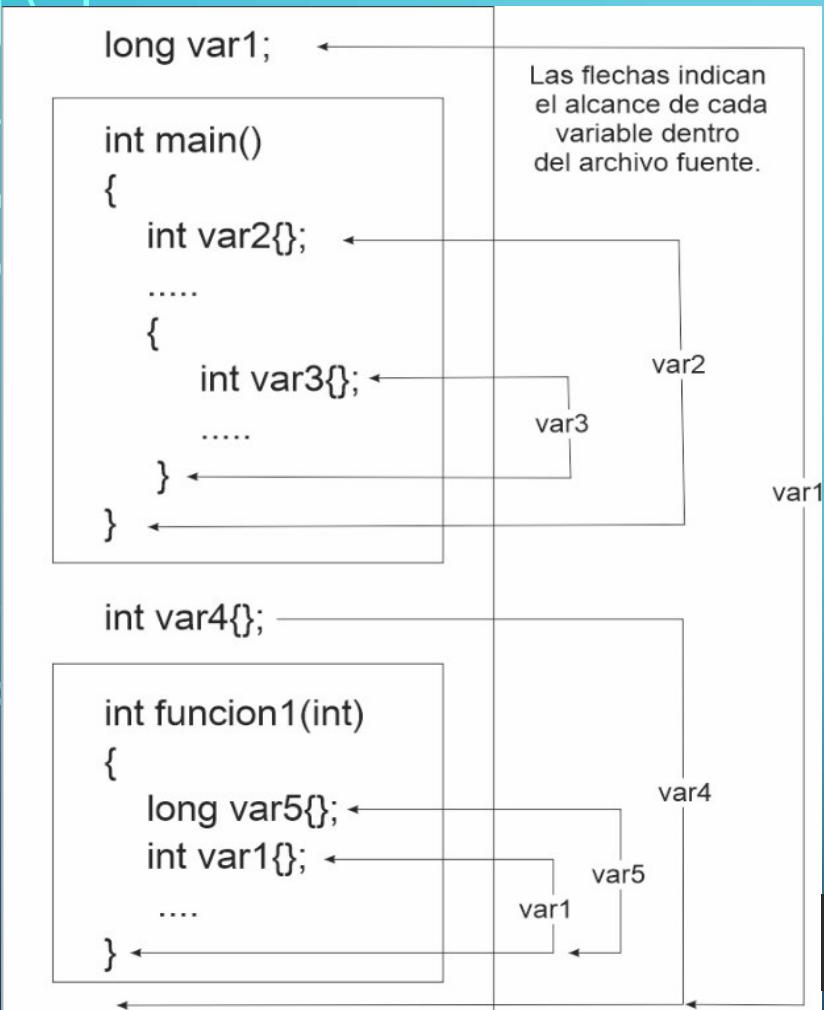
Por lo general, debe colocar una definición lo más cerca posible de donde se usa la variable por primera vez. Esto hace que su código sea más fácil de entender para otro programador. Existen excepciones: las llamadas **variables globales**. Puede definir variables fuera de todas las funciones de un programa. Las variables definidas fuera de todos los bloques y clases también se denominan globales y tienen un alcance global (que también se denomina **alcance de espacio de nombres global**). Esto significa que son accesibles en todas las funciones en el archivo de origen siguiendo el punto en el que se definen. Si los define al principio de un archivo de origen, estarán accesibles en todo el archivo. Las variables globales tienen una duración de almacenamiento estática por defecto, por lo que existen desde el inicio del programa hasta que finaliza la ejecución del programa.



## Alcance de una variable

La inicialización de las variables globales tiene lugar antes de que comience la ejecución de `main()`, por lo que siempre están listas para usarse dentro de cualquier código que esté dentro del alcance de la variable. Si no inicializa una variable global, se inicializará en cero de forma predeterminada. Esto es diferente a las variables automáticas, que contienen valores basura cuando no se inicializan.

# Tiempo de vida de una variable

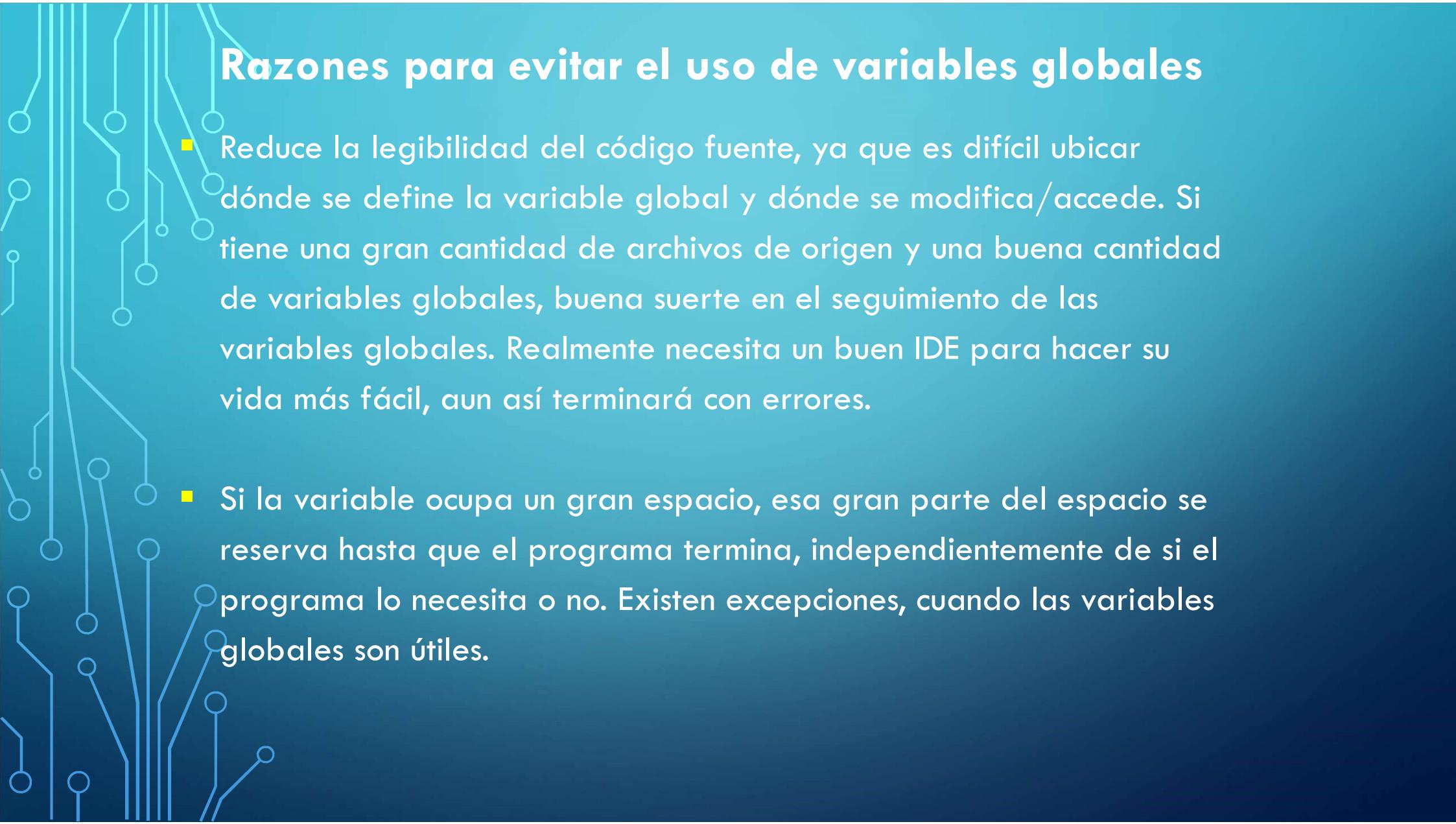


La variable `var1` de tipo `long` es global. Observe que dentro de `funcion1()` tenemos definida una variable con el mismo nombre pero del tipo `int`. La variable local llamada `var1` en `funcion1()` ocultará la variable global del mismo nombre. Si usa el nombre `var1` en la función, está accediendo a la variable automática local de ese nombre. Para acceder a la variable global `var1`, debe calificarlo con el **operador de resolución de alcance ::**. Así es como podría acceder los valores de las variables locales y globales que tienen el nombre `var1`:

```
std::cout << "Var1 global = " << ::var1 << std::endl;
std::cout << "Var1 local = " << var1 << std::endl;
```

## Razones para evitar el uso de variables globales

- La variable global tiene alcance global. Entonces, cualquier función en el programa puede modificarlo, y no tienes idea de quién lo modificó.
- Dificulta la depuración ya que no existe certeza acerca de quién modificó el valor.
- No es adecuado para programas de subprocessos múltiples (threads), ya que los subprocessos terminarían con una condición de carrera. Para evitar esto, es posible que necesite un mecanismo de sincronización que ralentizaría su programa.



## Razones para evitar el uso de variables globales

- Reduce la legibilidad del código fuente, ya que es difícil ubicar dónde se define la variable global y dónde se modifica/accede. Si tiene una gran cantidad de archivos de origen y una buena cantidad de variables globales, buena suerte en el seguimiento de las variables globales. Realmente necesita un buen IDE para hacer su vida más fácil, aun así terminará con errores.
- Si la variable ocupa un gran espacio, esa gran parte del espacio se reserva hasta que el programa termina, independientemente de si el programa lo necesita o no. Existen excepciones, cuando las variables globales son útiles.



## Razones para evitar el uso de variables globales

*Las pautas comunes de codificación y diseño dictan que las variables globales generalmente deben evitarse, y por una buena razón. Las constantes globales son una excepción a esta regla. Es decir, variables globales que se declaran con la palabra clave **const**. Se recomienda definir todas sus constantes solo una vez, y las variables globales son perfectamente adecuadas para eso.*

# Condicionales

*Operadores relacionales*

Operator	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to

# Condicionales

*Operadores lógicos*

Operator	Description
<code>&amp;&amp;</code>	Logical AND
<code>  </code>	Logical OR
<code>!</code>	Logical negation (NOT)

# Condicionales

Las variables tipo bool pueden contener valores true(1) o false(0).  
Representan valores numéricos.

```
bool isValid {true}; // Define e inicializa una variable lógica
```

Ejemplos de aplicación de operadores de comparación

```
i > j      i != j     j > -8     i <= j + 15
```

i y j son valores enteros. Dado que la operación de suma tiene mayor precedencia que la comparación, la última expresión evalúa primero  $j+15$  y luego  $i \leq$  resultado suma  $j+15$ . Puede almacenar el resultado de una comparación en una variable tipo bool:

```
isValid = i < j;
```

# Condicionales

Otro ejemplo:

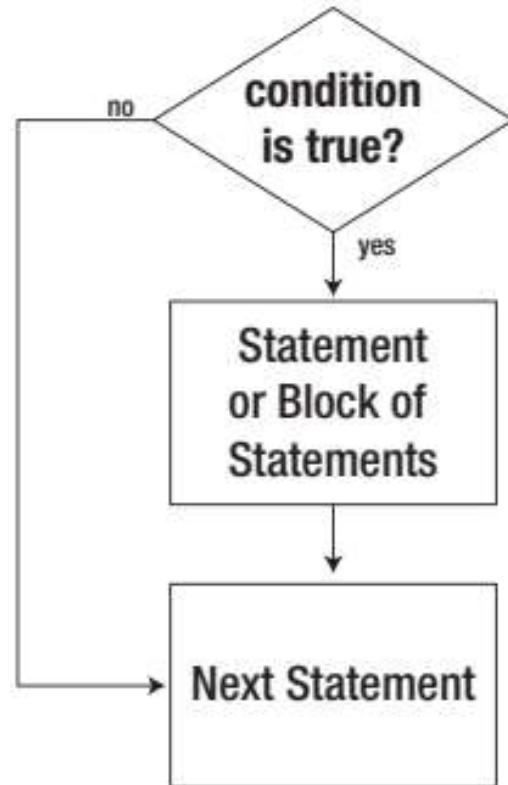
```
char first {'A'};    // Ascii 'A'  
char last {'Z'};    // Ascii 'Z'  
  
first < last 'E' <= first first != last
```

# Sentencia if

```
if (condition)
    statement;
Next statement;
```

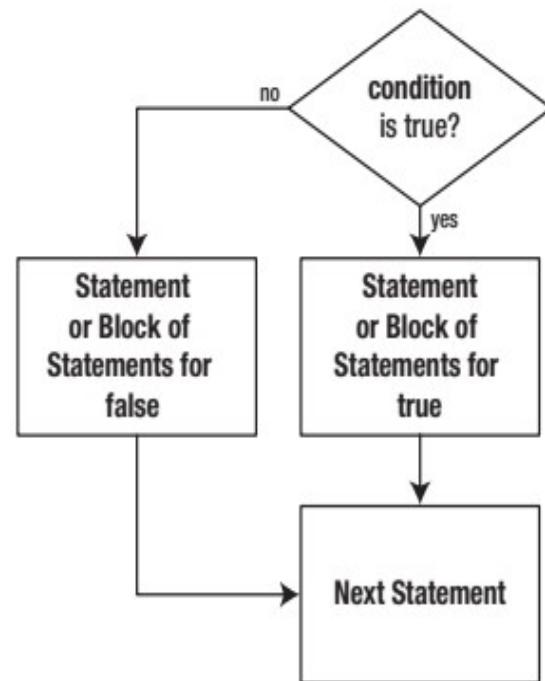
or

```
if (condition)
{
    statement;
    ...
}
Next statement;
```



# Sentencia if-ele

```
if (condition)
{
    // Statements when condition is true
}
else
{
    // Statements when condition is false
}
// Next statement
```



# Sentencia if

*Funciones para la clasificación de caracteres provistas por el header ctype*

Function	Operation
isupper(c)	Tests whether c is an uppercase letter, by default 'A' to 'Z'.
islower(c)	Tests whether c is a lowercase letter, by default 'a' to 'z'.
isalpha(c)	Tests whether c is an uppercase or lowercase letter (or any alphabetic character that is neither uppercase nor lowercase, should the locale's alphabet contain such characters).
isdigit(c)	Tests whether c is a digit, '0' to '9'.
isxdigit(c)	Tests whether c is a hexadecimal digit, either '0' to '9', 'a' to 'f', or 'A' to 'F'.
isalnum(c)	Tests whether c is an alphanumeric character; same as isalpha(c)    isdigit(c).
isspace(c)	Tests whether c is whitespace, by default a space (' '), newline ('\n'), carriage return ('\r'), form feed ('\f'), or horizontal ('\t') or vertical tab ('\v').
isblank(c)	Tests whether c is a space character used to separate words within a line of text. By default either a space (' ') or a horizontal tab ('\t').
ispunct(c)	Tests whether c is a punctuation character. By default, this will be either a space or one of the following: _ { } [ ] # ( ) < > % : ; . ? * + - / ^ &   ~ ! = , \ " '
isprint(c)	Tests whether c is a printable character, which includes uppercase or lowercase letters, digits, punctuation characters, and spaces.
iscntrl(c)	Tests whether c is a control character, which is the opposite of a printable character.
isgraph(c)	Tests whether c has a graphical representation, which is true for any printable character other than a space.

## Sentencia if

Cada una de las funciones anteriores devuelve un valor de tipo **int**. El valor será distinto de cero (**true**) si el carácter es del tipo que se está probando y 0 (**false**) si no lo es. Quizás se pregunte por qué estas funciones no devuelven un valor **bool**, lo que tendría mucho más sentido. La razón por la que no devuelven un valor **bool** es que se originan en la biblioteca estándar de C y son anteriores al tipo **bool** en C++. Observe que el nombre de las funciones comienza con **is**, lo cual por convención nos indica que devuelve un tipo **bool**.

# Sentencia if

*Funciones para la conversión de caracteres provistas por el header ctype*

Function	Operation
<code>tolower(c)</code>	If <code>c</code> is uppercase, the lowercase equivalent is returned; otherwise, <code>c</code> is returned.
<code>toupper(c)</code>	If <code>c</code> is lowercase, the uppercase equivalent is returned; otherwise, <code>c</code> is returned.

Todas las funciones estándar de conversión y clasificación de caracteres, excepto `isdigit()` e `isxdigit()`, funcionan de acuerdo con las reglas de la configuración regional actual. todos los ejemplos proporcionados en la tabla 4-2 son para el entorno local predeterminado, denominado "C", que es un conjunto de preferencias similares a las utilizadas por los estadounidenses de habla inglesa. la biblioteca estándar de C++ ofrece una amplia biblioteca para trabajar con otras configuraciones regionales y juegos de caracteres. Puede usarlos para desarrollar aplicaciones que funcionen correctamente independientemente del idioma del usuario y las convenciones regionales. Consulte una referencia de la biblioteca estándar para mas detalles

# Sentencia if

Ejemplo:

```
#include <iostream>

int main() {

    char letter{}; // para almacenar entrada.

    std::cout << "Ingrese una letra: ";
    std::cin >> letter;

    if (std::isupper(letter)) {
        std::cout << "Usted ingreso una letra mayúscula." << std::endl;
        return 0;
    }

    if (std::islower(letter)) {
        std::cout << "Usted ingreso una letra minúscula." << std::endl;
        return 0;
    }
}
```

## Evaluación de cortocircuito

- Si necesita probar múltiples condiciones que están unidas con operadores lógicos, entonces debe poner los menos costosos para calcular primero. Por supuesto, esta técnica realmente solo vale la pena si uno de los operandos es realmente costoso de calcular.
- El cortocircuito se utiliza más comúnmente para evitar la evaluación de los operandos de la derecha que, de otro modo, no se evaluarían. Esto se hace poniendo otras condiciones primero que provocan un cortocircuito evitando operandos que pueden fallar. Como veremos más adelante, una aplicación popular de esta técnica es verificar que un puntero no sea nulo antes de desreferenciarlo.
- Este tipo de evaluación no es aplicable al operador XOR.

## Evaluación de cortocircuito

En este ejemplo como  $x < 0$  es falso, no se evalúa  $x*x + 632*x == 1268$

```
int x = 2;
if (x < 0 && (x*x + 632*x == 1268))
{
    std::cout << "El resultado es correcto" << std::endl;
```

Aquí como  $x = 2$  es verdadero, no se evalúa  $x*x + 632*x == 1268$

```
int x = 2;
if (x == 2 || (x*x + 632*x == 1268))
{
    std::cout << "El resultado es correcto" << std::endl;
```

```
int denom {0}, num {145};

if (denom && num/denom)
{
    // Otras operaciones
```

## Operador ternario

El operador condicional a veces se denomina operador ternario porque involucra tres operandos, el único operador que lo hace. Es similar a la instrucción **if-else**, en que en lugar de seleccionar uno de los dos bloques de instrucciones para ejecutar según una condición, selecciona el valor de una de las dos expresiones. Por lo tanto, el operador condicional le permite elegir entre dos valores.

`c = a > b? a : b;` equivale a:

```
if(a > b){  
    c = a;  
}else{  
    c = b;  
}
```

Su forma general es: *condition ? expresion1 : expresion2*

# Operador ternario

## Ejemplos

```
prestamo = (2*ingresos < saldo/2)? (2*ingresos) : (saldo/2);  
  
divisor? (dividendo / divisor) : 0;  
  
std::cout << (ingresos < egresos? "Su negocio no es rentable":"Su negocio es rentable!!")  
    << std::endl;  
  
std::cout << (ingresos < egresos? "Su negocio no es rentable":  
    (ingresos == egresos?"Su negocio esta al limite":"Su negocio es rentable!!"))  
    << std::endl;
```

# Operador ternario

## Ejemplos

```
prestamo = (2*ingresos < saldo/2)? (2*ingresos) : (saldo/2);  
  
divisor? (dividendo / divisor) : 0;  
  
std::cout << (ingresos < egresos? "Su negocio no es rentable":"Su negocio es rentable!!")  
    << std::endl;  
  
std::cout << (ingresos < egresos? "Su negocio no es rentable":  
    (ingresos == egresos?"Su negocio esta al limite":"Su negocio es rentable!!"))  
    << std::endl;
```

## Ejercitación

Escriba un programa que lea 3 valores de tipo double distintos de cero y determinar si representan los lados de un triángulo.

# SWITCH - CASE

## Estructura general:

```
switch(opción) //donde opción es la variable a comparar
{
    case valor1: //Bloque de instrucciones 1;
        break;
    case valor2: //Bloque de instrucciones 2;
        break;
    case valor3: //Bloque de instrucciones 3;
        break;
        // Nótese que valor 1 2 y 3 son los valores que puede tomar la opción
        // la instrucción break es necesaria, para no ejecutar todos los casos.
    default: //Bloque de instrucciones por defecto;
        //default, es el bloque que se ejecuta en caso de que no se de ningún caso
}
```

## SWITCH - CASE

La sentencia `switch` le permite seleccionar entre múltiples opciones, según el valor de una variable o expresión entera. Las opciones se identifican mediante un conjunto de números enteros fijos o valores de enumeración, y la selección de una opción particular está determinada por el valor de un número entero o una constante de enumeración dados. Las opciones en una declaración de cambio se llaman **casos (case)**.

Las opciones posibles en una sentencia `switch` aparecen en un bloque, y cada opción se identifica por un valor de caso. Un valor de caso aparece en una **etiqueta de caso**, que tiene la siguiente forma:

```
case valor_caso: // valor caso se llama etiqueta de caso
```

## SWITCH - CASE

- Cada valor de caso debe ser una expresión constante, que es una expresión que el compilador puede evaluar en tiempo de compilación.
- Los valores de caso son en su mayoría literales o variables constantes que se inicializan con literales. Cualquier etiqueta de caso debe ser del mismo tipo que la expresión de condición dentro del switch() anterior o ser convertible a ese tipo.
- Una etiqueta llamada default determina el caso por defecto en caso de que no se cumpla con ninguno de los casos listados en el switch. Esta etiqueta es opcional. No se necesita indicar break para terminar el bloque default, dado que es el último que se evalúa.
- El bloque de instrucciones correspondientes a un caso está limitado por la palabra clave break que “rompe” el bloque switch finalizando su ejecución.

## SWITCH - CASE

- Los valores de caso (etiquetas) deben ser únicos, no pueden existir 2 con el mismo valor.
- Varios valores de casos pueden compartir la misma acción, como muestra el siguiente ejemplo. La ejecución continua hasta encontrar el primer break.

```
switch (std::tolower(letra))
{
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        std::cout << "Es una vocal." << std::endl;
        break;
    default:
        std::cout << "Es una consonante" << std::endl;
        break;
}
```

## SWITCH - CASE

Podemos reescribir el ejemplo anterior como:

```
switch (std::tolower(letra))
{
    case 'a': case 'e': case 'i': case 'o': case 'u':
        std::cout << "Es una vocal." << std::endl;
        break;
    default:
        std::cout << "Es una consonante" << std::endl;
        break;
}
```

# SWITCH - CASE

## Fallthrough

La sentencia `break` al final de cada grupo de sentencias `case` transfiere la ejecución a la sentencia posterior al cambio. Cuando quitamos un `break`, el código debajo de la etiqueta del caso que sigue directamente al caso sin `break` también se ejecuta. Este fenómeno se llama **fallthrough** porque, de alguna manera, "caemos" en el siguiente caso. La mayoría de las veces, un `break` faltante indica un descuido y, por lo tanto, un error. Algunos compiladores pueden emitir una advertencia por no encontrar un `break` que termine el caso. C++17 agrega una nueva función de lenguaje para señalar tanto al compilador como a la persona que lee su código que usted está utilizando intencionalmente **fallthrough**: puede agregar una declaración `[[fallthrough]]` en el mismo lugar donde agregaría una declaración de ruptura. Obviamente que esto es *opcional*, pero se lo considera *una buena práctica*.

## SWITCH - CASE

```
switch (std::tolower(letra))
{
    case 'a':
        [[fallthrough]];
    case 'e':
        [[fallthrough]];
    case 'i':
        [[fallthrough]];
    case 'o':
        [[fallthrough]];
    case 'u':
        std::cout << "Es una vocal." << std::endl;
        break;
    default:
        std::cout << "Es una consonante" << std::endl;
        break;
}
```

Para casos vacíos, como los del ejemplo, se permite pero no se requiere una declaración `[[fallthrough]]`. Los compiladores ya no emiten advertencias para esto.

## SWITCH - CASE

**NOTA:** Solo puede evaluar valores de tipos integrales (`int`, `long`, `unsigned short`, etc.), caracteres (`char`, etc.) y enumeraciones. Técnicamente, también se permite evaluar valores booleanos, pero en lugar de evaluar valores booleanos, debería utilizar `if/else`. Sin embargo, a diferencia de otros lenguajes de programación, C++ no le permite crear sentencias `switch()` con condiciones y etiquetas que contengan expresiones de cualquier otro tipo. Por ejemplo, no se permite que un `switch` evalúe `strings`.

## SWITCH - CASE

En una sentencia switch una variable definida dentro de un caso tiene alcance desde que se crea hasta que finaliza el switch, salvo que se encierre en un bloque { }. Existe un problema con esto, porque dicha variable podría accederse en un caso posterior y si el caso donde se define es pasado por alto estaríamos intentando acceder a una variable no definida. **Ejemplo:**

```
int test {3};

switch (test) {
    int i{1}; // ILEGAL - no puede ser alcanzada
case 1:
    int j{2}; // ILEGAL - puede ser alcanzada pero puede ser pasada por alto.
    std::cout << test + j << std::endl;
    break;
    int k{3}; // ILEGAL - no puede ser alcanzada
case 3: {
    int m{4}; // OK - puede ser alcanzada y no puede ser pasada por alto.
    std::cout << test + m << std::endl;
    break;
}
default:
    int n {5}; // OK - puede ser alcanzada y no puede ser pasada por alto.
    std::cout << test + n << std::endl;
    break;
}
```

## SWITCH - CASE

Solo dos de las definiciones de variables en esta sentencia switch son válidas: las de m y n. Para que una definición sea legal, primero debe ser posible que se alcance y, por lo tanto, se ejecute en el curso normal de la ejecución.

Claramente, este no es el caso para las variables i y k. En segundo lugar, no debe ser posible durante la ejecución ingresar al alcance de una variable sin pasar por alto su definición, como es el caso de la variable j. Si la ejecución salta al caso con la etiqueta 3 o al caso predeterminado, ingresa al ámbito en el que se definió la variable j, mientras omite su definición real. Eso es ilegal.

Sin embargo, la variable m solo está "dentro del alcance" desde su declaración hasta el final del bloque que la encierra, por lo que esta declaración no se puede omitir. Y la declaración de la variable n no se puede omitir porque no hay casos después del caso predeterminado. Tenga en cuenta que no es porque se trate del caso predeterminado que la declaración de n es legal; si hubiera casos adicionales después del predeterminado, la declaración de n habría sido igual de ilegal.

## Condicionales – Nuevas sintaxis C++17

C++17 introduce nueva sintaxis que permite inicializar variables dentro de los paréntesis donde evalúa la expresión condicional

```
if (initialization; condition) ...
```

```
switch (initialization; condition) { ... }
```

Ejemplo:

```
if(int valor{}; test > 2){  
    valor = test * test + 250;
```

# EJERCITACION

**Ejercicio:** Suponga que una entidad financiera ofrece cuatro tipos de inversiones posibles a saber:

1. Bonos tipo A con un interés del 5% mensual. Solo se permiten montos a invertir mayores a 150000
2. Bonos tipo B con un interés de 4.3% mensual. Solo se permiten montos a invertir hasta 150000 mayores a 100000.
3. Bonos tipo C con un interés de 3 % mensual. Solo se permiten montos a invertir hasta 100000 mayores a 80000.
4. Bonos tipo D con un interés de 2.7 % mensual. Para montos menores a 80000.

Realice un programa que le permita al usuario ingresar:

1. Monto a invertir
2. Tipo de bono en cual desea invertir
3. Plazo en meses.

Calcule la ganancia obtenida y preséntela en pantalla. Si el monto a invertir no cumple las condiciones del bono seleccionado, indique que no es posible realizar la operación e indique el saldo mínimo requerido. Trate de utilizar enumeraciones y switch-case.



# Arrays

Variable que representa una secuencia de ubicaciones de memoria. Definición:

```
tipo_dato nombre[cantidad_elementos];
```

```
double temperaturas[366]; // Define un array de 366 temperaturas
```

Esto define un array con el nombre `temperaturas` para almacenar 366 valores de tipo `double`. Los elementos del array no se inicializan en esta declaración, por lo que contienen valores basura.

# Arrays

*El tamaño del array siempre debe especificarse mediante una expresión entera constante.* Se puede usar cualquier expresión entera que el compilador pueda evaluar en el momento de la compilación, aunque en su mayoría será un literal entero o una variable entera constante que a su vez se inicializó usando un literal. *No puede definirse el tamaño de un array en tiempo de ejecución.*

## Acceso a elementos del array

Se puede acceder a los elementos en forma indexada (índices 0 a n-1, siendo n la dimensión del array). **Ejemplo:**

```
temperatures[1] = 35.7;
```

# Arrays

Podemos inicializar un array utilizando de la siguiente manera:

```
double temperatures[366] {25.7, 32.1, 15, 28.3} // desde C++11
```

```
double temperatures[366]= {25.7, 32.1, 15, 28.3} // antes de C++11
```

Los 362 elementos restantes serán cero.

*Ejemplo 2:*

```
unsigned int height[6] {26, 37, 47, 55, 62, 75}; // Define e inicializa  
// un array de 6 alturas
```

El array ocupa 24 bytes (6 int x 4 bytes) en un procesador Intel.

height [0]	height [1]	height [2]	height [3]	height [4]	height [5]
26	37	47	55	62	75

## Arrays

Para definir un array de valores que no se pueden modificar, simplemente agregue la palabra clave **const** a su tipo. Lo siguiente define un array de seis constantes **unsigned int**:

```
const unsigned int height[6] {26, 37, 47, 55, 62, 75};
```

## Operaciones

Se opera de la misma forma que una variable ordinaria. **Ejemplo:**

```
double average_temp = (temperature[0] + temperatura[1]) / 2
```

# Arrays

## Copia de arrays

A partir de C++03 se puede copiar arrays de la siguiente forma:

```
const size_t SIZE 6;
unsigned int height[SIZE] {26, 37, 47, 55, 62, 75};
unsigned int copy[SIZE];

std::copy(height, height + SIZE, copy); //std::copy. Método C++03 o superior
```

Otra forma de copiar es a través de bucles. Obviamente, si trabaja con un compilador con standard anterior a C++03 la única opción disponible será a través de bucles.

```
double average_temp = (temperature[0] + temperatura[1]) / 2
```

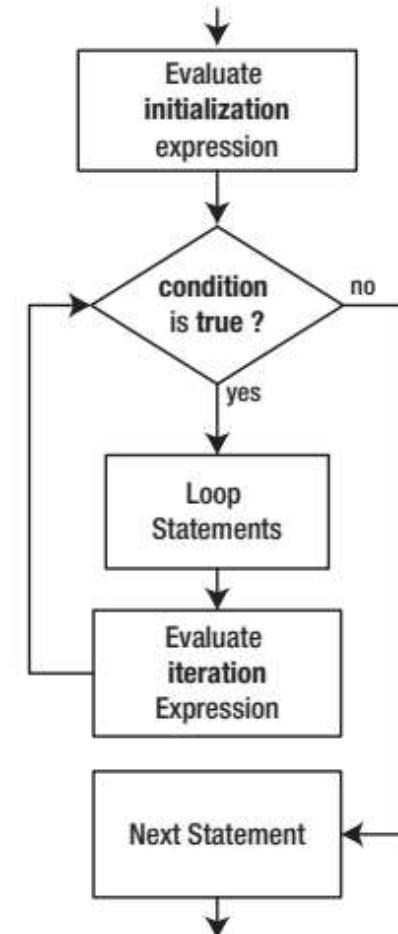
# Estructuras repetitivas

Estructuras repetitivas disponibles en C++

- ✓ Ciclo **for** ( ciclo del tipo "0-N" )
- ✓ Ciclo **while** ( ciclo del tipo "0-N" )
- ✓ Ciclo **do while** ( ciclo del tipo "1-N" )

## Estructura repetitiva – Ciclo for

```
for (initialization; condition; iteration)
{
    // Loop statements
}
// Next statement
```

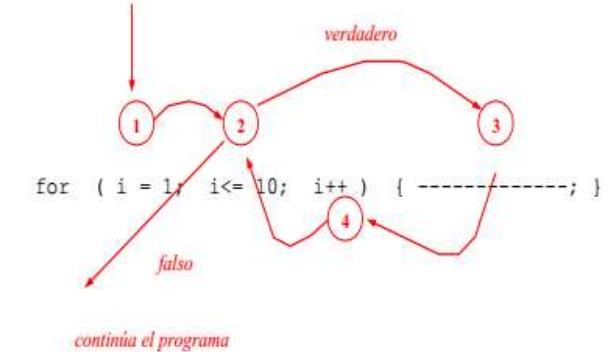
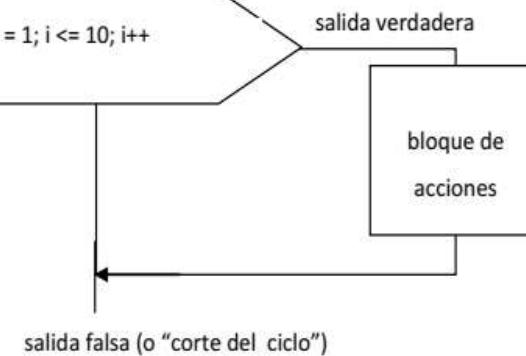


# Estructura repetitiva – Ciclo for

## Sintaxis:

```
for ( i=1; i<=10; i++ )  
{  
    bloque de acciones  
}
```

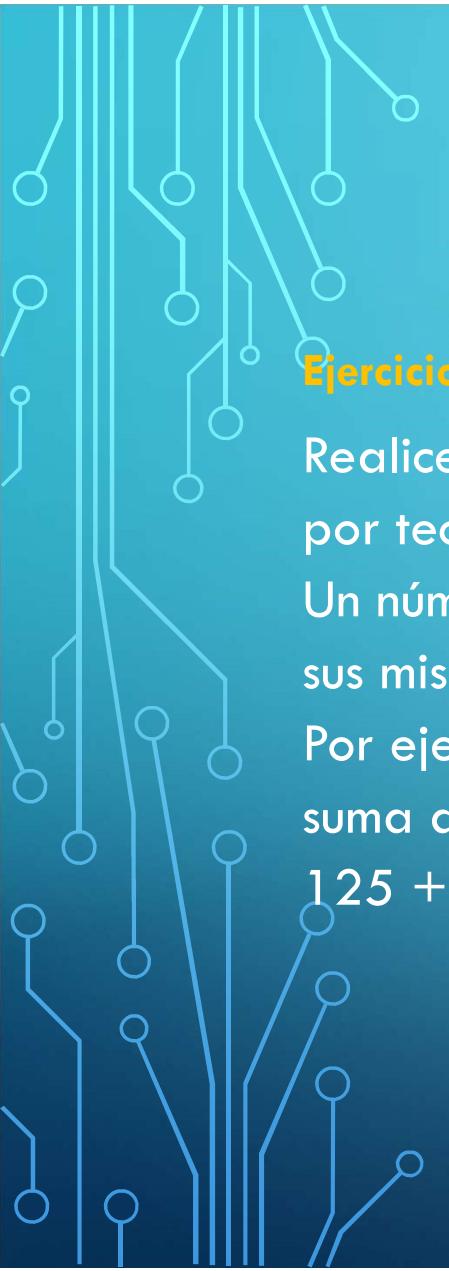
## Diagrama



# EJERCITACION

**Ejercicio:**

**Realice un programa que lea un numero desde teclado que sea de 4 dígitos, si es de menos dígitos lanzar un mensaje que rechace el número. Luego determine si todos los dígitos del dicho numero son números primos y en tal caso imprimir el mensaje “Todos los dígitos del número xxxx son primos”.**



## EJERCITACION

### Ejercicio:

Realice un programa que compruebe si un número(de hasta 6 cifras) ingresado por teclado es un número de Armstrong o no.

Un número de Armstrong , es todo aquel número que es la suma de cada uno de sus mismos dígitos elevado al número total de dígitos.

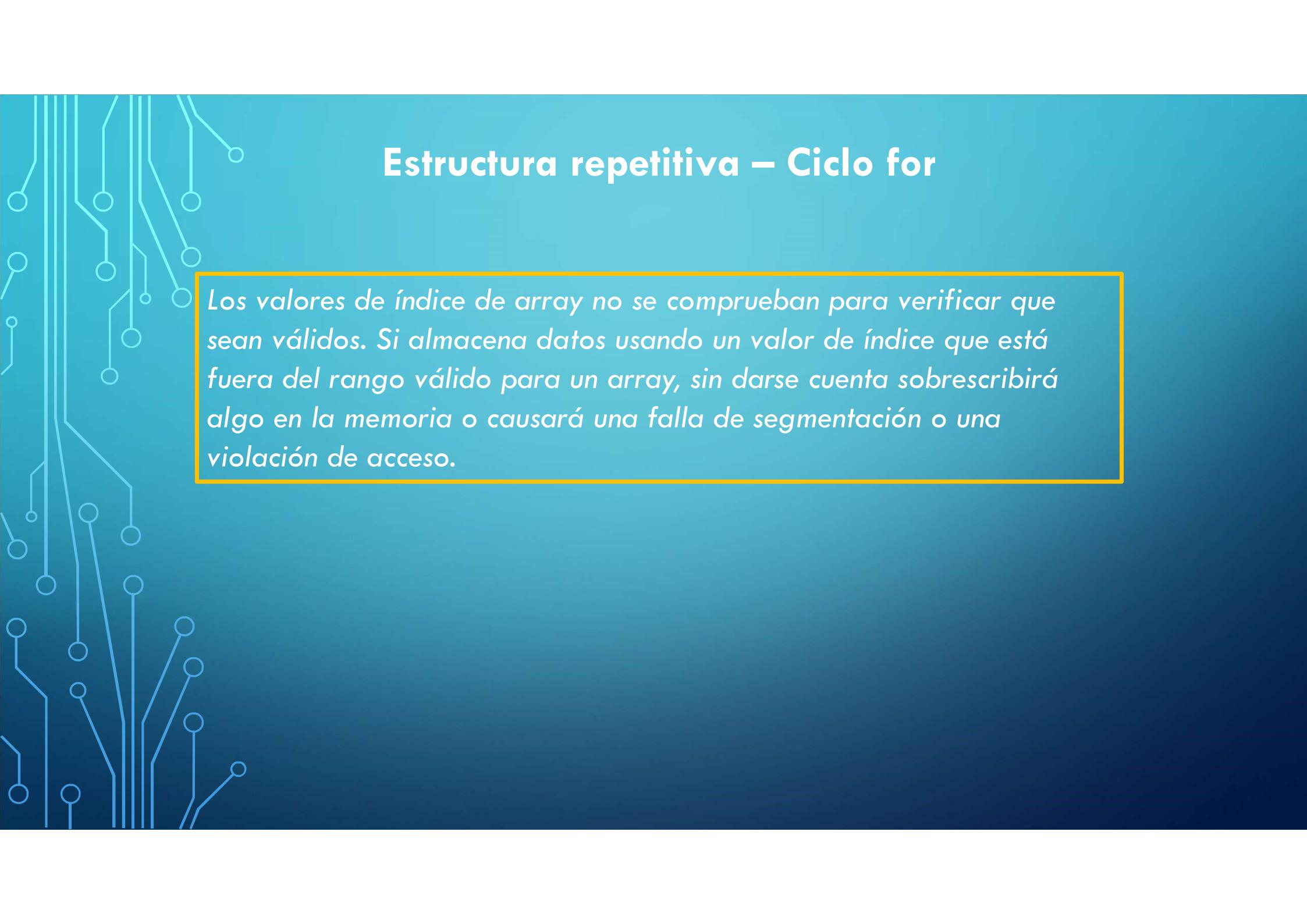
Por ejemplo el número 153 es de Armstrong ya que este posee 3 dígitos y la suma de cada uno de sus dígitos elevado a 3 es igual a  $1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$ .

## Estructura repetitiva – Ciclo for

En el uso más típico del bucle for, la primera expresión inicializa un contador, la segunda expresión comprueba si el contador ha alcanzado un límite determinado y la tercera expresión incrementa el contador. La variable de control es de tipo `size_t`. Por ejemplo, podría copiar los elementos de un array a otro de esta manera:

```
// Registro de lluvias caídas durante un año (12 meses).
double precipitaciones[12] {1.1, 2.8, 3.4, 3.7, 2.1, 2.3, 1.8, 0.0,
0.3, 0.9, 0.7, 0.5};
double copia[12] {};

for (size_t i {}; i < 12; ++i) // i varia desde 0 a 11
{
    // Copiar elemento i de precipitaciones en elemento i de copia
    copia[i] = precipitaciones[i];
}
// i ya no existe aquí...
```



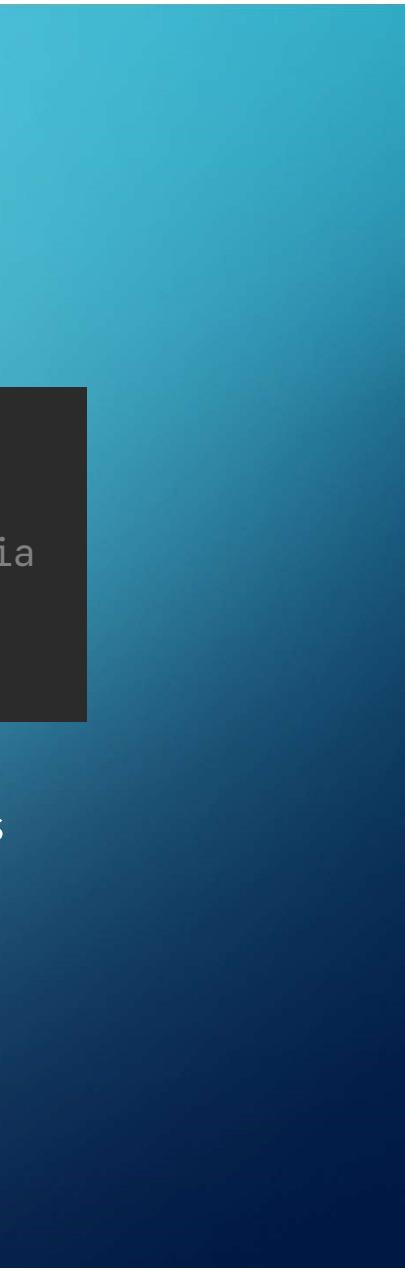
## Estructura repetitiva – Ciclo for

*Los valores de índice de array no se comprueban para verificar que sean válidos. Si almacena datos usando un valor de índice que está fuera del rango válido para un array, sin darse cuenta sobrescribirá algo en la memoria o causará una falla de segmentación o una violación de acceso.*



## Estructura repetitiva – Ciclo for

```
size_t i {};
for (i = 0; i < 12; ++i) // i varia desde 0 a 11.
{
    // Copiar elemento i de precipitaciones en elemento i de copia
    copia[i] = precipitaciones[i];
}
// i todavía existe aquí...
```



Si aplicamos los conceptos de alcance vistos con anterioridad nos daremos cuenta que el alcance de la variable **i** va mas alla del bloque del ciclo for.



## Estructura repetitiva – Ciclo for

```
size_t i {};
for ( ; i < 12; ++i) // i varia desde 0 a 11.
{
    // Copiar elemento i de precipitaciones en
    // elemento i de copia
    copia[i] = precipitaciones[i];
}
// i todavía existe aquí...
```



Dado que `i` fue definido e inicializado previamente, podemos obviar la inicialización en el encabezado del ciclo `for`.

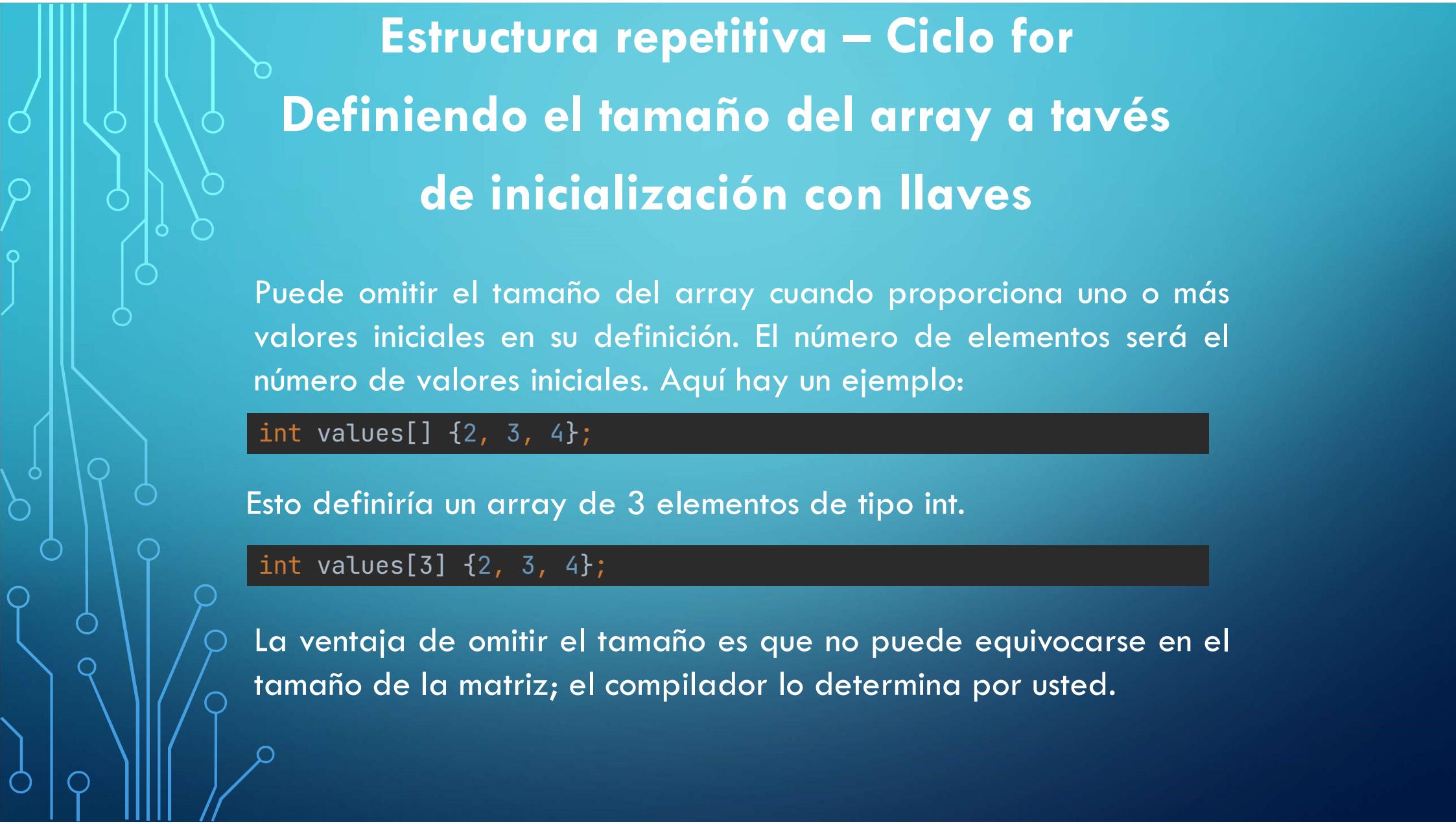
# Estructura repetitiva – Ciclo for

## Evitando los números mágicos

EJERCICIO: En base a los datos proporcionados a continuación, crear un programa que calcule el promedio de alturas de un grupo de 6 personas y cuantas personas superan el promedio de altura.

```
const unsigned size {6}; // Tamaño del array.  
unsigned height[size] {175, 180, 169, 171, 135, 175}; // Array de  
alturas en cm
```

```
// Tip. Forma abreviada de incrementar el contador.  
count += height[i] > average
```



## Estructura repetitiva – Ciclo for

### Definiendo el tamaño del array a través de inicialización con llaves

Puede omitir el tamaño del array cuando proporciona uno o más valores iniciales en su definición. El número de elementos será el número de valores iniciales. Aquí hay un ejemplo:

```
int values[] {2, 3, 4};
```

Esto definiría un array de 3 elementos de tipo int.

```
int values[3] {2, 3, 4};
```

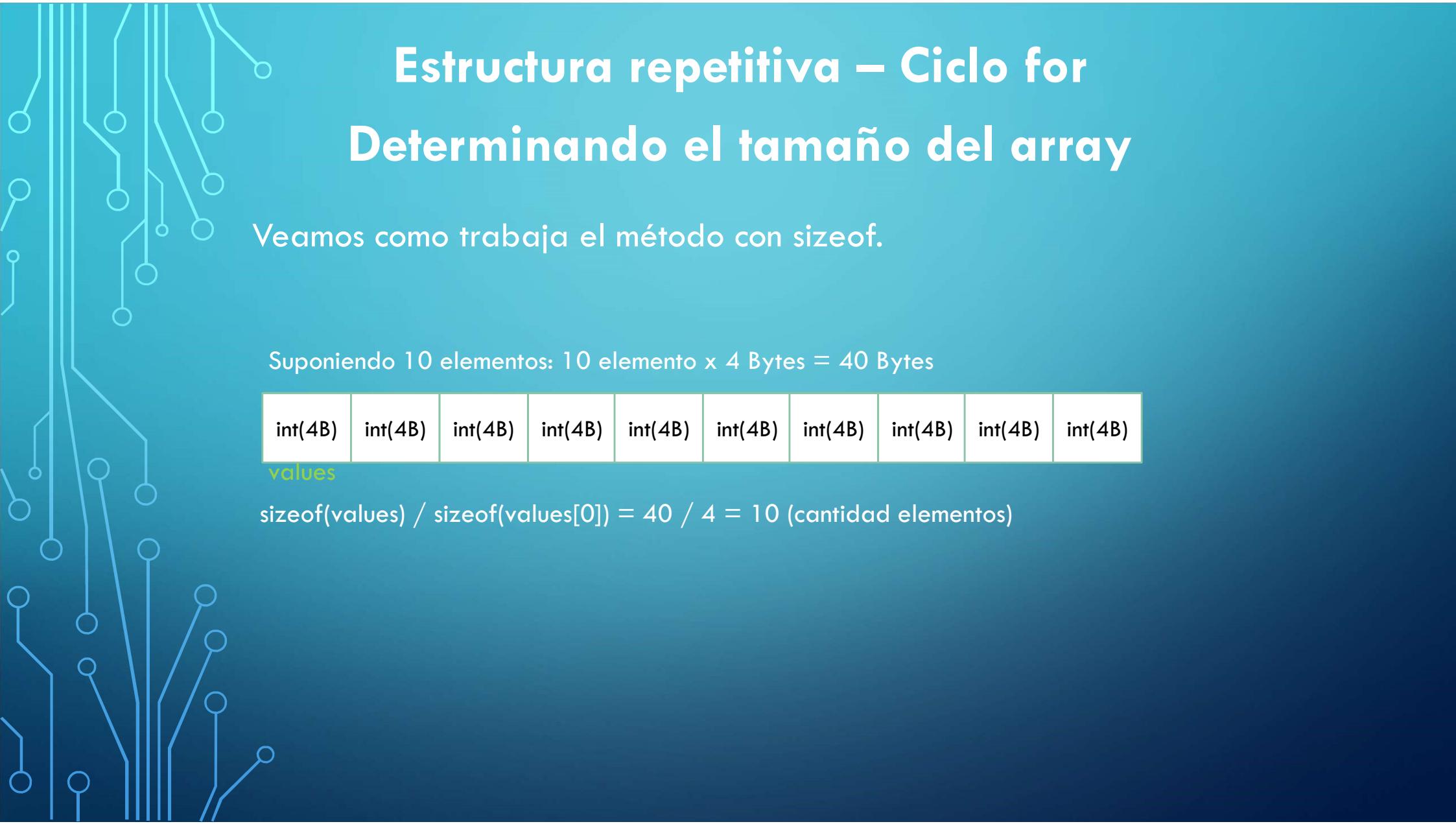
La ventaja de omitir el tamaño es que no puede equivocarse en el tamaño de la matriz; el compilador lo determina por usted.

# Estructura repetitiva – Ciclo for

## Determinando el tamaño del array

Antes de C++17 se utilizaba (se sigue utilizando) `sizeof`, a partir de C++17 disponemos de la función `std::size()`. Se debe incluir el header `array`.

```
// Como obtener el numero de elementos de un array.
#include <iostream>
#include <array> // para std::size()
int main()
{
    int values[] {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
    std::cout << "Hay" << sizeof(values) / sizeof(values[0])
          << " elements en el array." << std::endl;
    int sum {};
    for (size_t i {}; i < std::size(values); ++i)
    {
        sum += values[i];
    }
    std::cout << "La suma de los elementos del array es " << sum
    << std::endl;
}
```

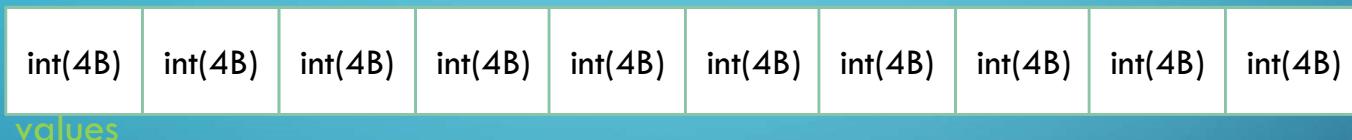


# Estructura repetitiva – Ciclo for

## Determinando el tamaño del array

Veamos como trabaja el método con sizeof.

Suponiendo 10 elementos: 10 elemento x 4 Bytes = 40 Bytes



$$\text{sizeof(values)} / \text{sizeof(values[0])} = 40 / 4 = 10 \text{ (cantidad elementos)}$$



# Estructura repetitiva – Ciclo for

## Determinando el tamaño del array

En el ejemplo anterior la suma la podemos realizar dentro de la tercera expresión de control del ciclo for:

```
int sum {};
for (size_t i {}; i < std::size(values); sum += values[i++]) {}
```

La expresión anterior acumula los elementos del array `values` y luego incrementa `i` (postincremento) que es la variable de control del ciclo. Si usa la forma de preincremento, obtiene la respuesta incorrecta para la suma de los elementos; también usará un valor de índice no válido que accede a la memoria más allá del final del array.

**Sugerencia:** Si bien la forma indicada es válida, se muestra para ver que es posible realizar esta acción. Sin embargo se sugiere colocar las instrucciones dentro del bloque de código para mejorar la legibilidad.

# Estructura repetitiva – Ciclo for

## Control de punto flotante

Transcriba este fragmento a su IDE y observe los resultados.

```
// Control de punto flotante en un bucle for
#include <iostream>
#include <iomanip>

int main()
{
    const double pi { 3.14159265358979323846 };
    const size_t perline {3}; // Salidas por linea
    size_t linecount {}; // Contador de salidas por linea

    for (double radius {0.2}; radius <= 3.0; radius += 0.2)
    {
        std::cout << std::fixed << std::setprecision(2)
            << " radius =" << std::setw(5) << radius
            << " area =" << std::setw(6) << pi * radius * radius;
        if (perline == ++linecount) // Se debe escribir linea?...
        {
            std::cout << std::endl; // ...salto de linea...
            linecount = 0; // ..resetear contador
        }
    }
    std::cout << std::endl;
}
```



# Estructura repetitiva – Ciclo for

## Control de punto flotante

Resultados.

```
radio = 0.20 area =  0.13 radio = 0.40 area =  0.50 radio = 0.60 area =  1.13
radio = 0.80 area =  2.01 radio = 1.00 area =  3.14 radio = 1.20 area =  4.52
radio = 1.40 area =  6.16 radio = 1.60 area =  8.04 radio = 1.80 area = 10.18
radio = 2.00 area = 12.57 radio = 2.20 area = 15.21 radio = 2.40 area = 18.10
radio = 2.60 area = 21.24 radio = 2.80 area = 24.63
```

```
Process finished with exit code 0
```

El bucle termina antes de lo esperado porque cuando se suma 0.2 a 2.8, el resultado es mayor que 3.0. La razón de esto es un error muy pequeño en la representación de 0.2 como un número de punto flotante binario. 0.2 no se puede representar exactamente en punto flotante binario. El error está en el último dígito de precisión, por lo que si su compilador admite una precisión de 15 dígitos para el tipo doble, el error es del orden de  $10^{-15}$ .

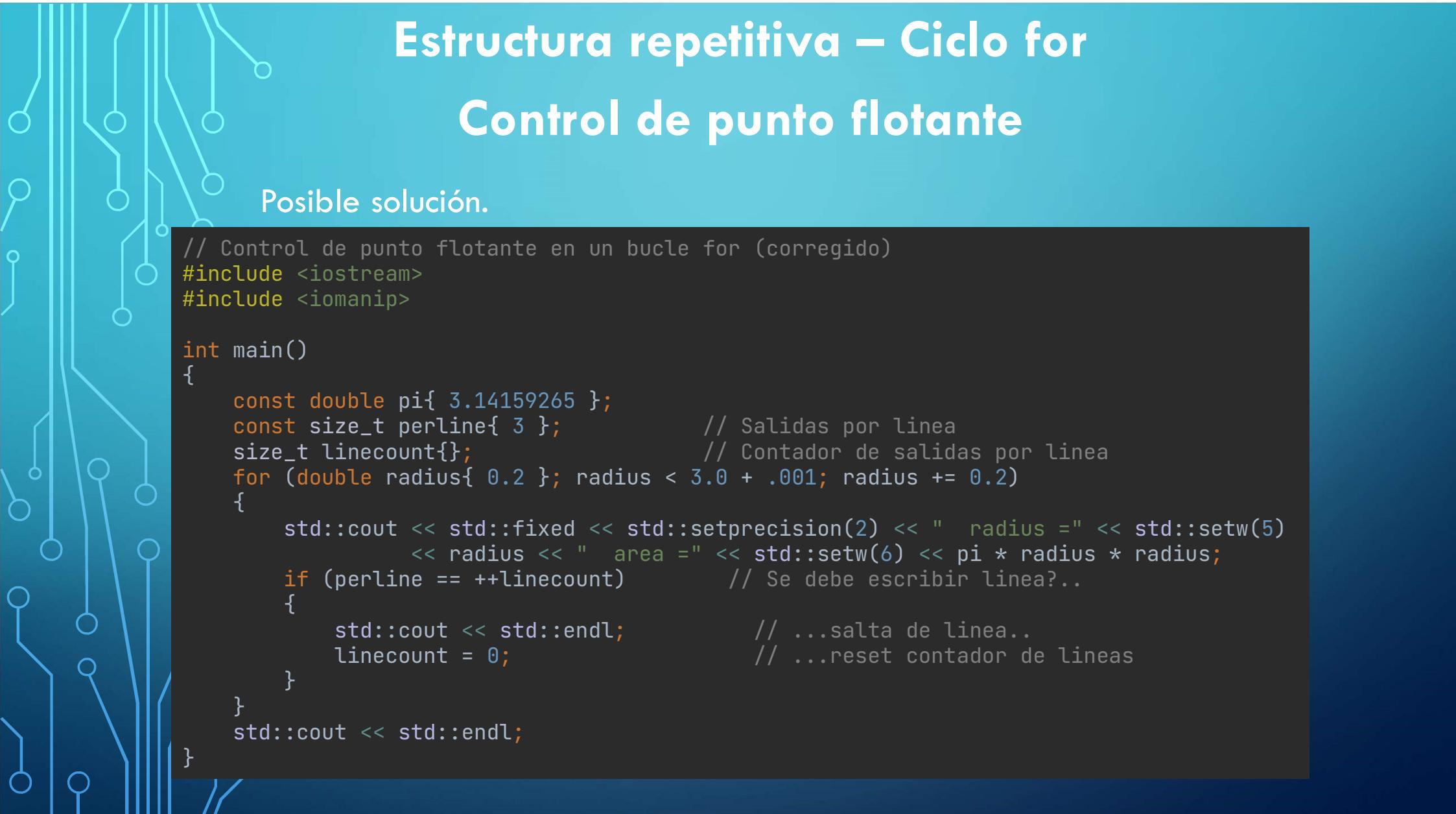
# Estructura repetitiva – Ciclo for

## Control de punto flotante

*Cualquier número que sea una fracción con un denominador impar no puede ser exactamente representado como un valor de punto flotante.*

Debe tener cuidado al usar una variable de punto flotante para controlar un bucle `for`. Es posible que los valores fraccionarios no se puedan representar exactamente como un número de punto flotante binario. Esto puede provocar algunos efectos secundarios no deseados, como demuestra este ejemplo completo:

<https://mrjaen.com/2021/10/24/algunos-desastres-atribuibles-al-mal-calculo-numerico-en-los-programas-informaticos/>



# Estructura repetitiva – Ciclo for

## Control de punto flotante

Possible solución.

```
// Control de punto flotante en un bucle for (corregido)
#include <iostream>
#include <iomanip>

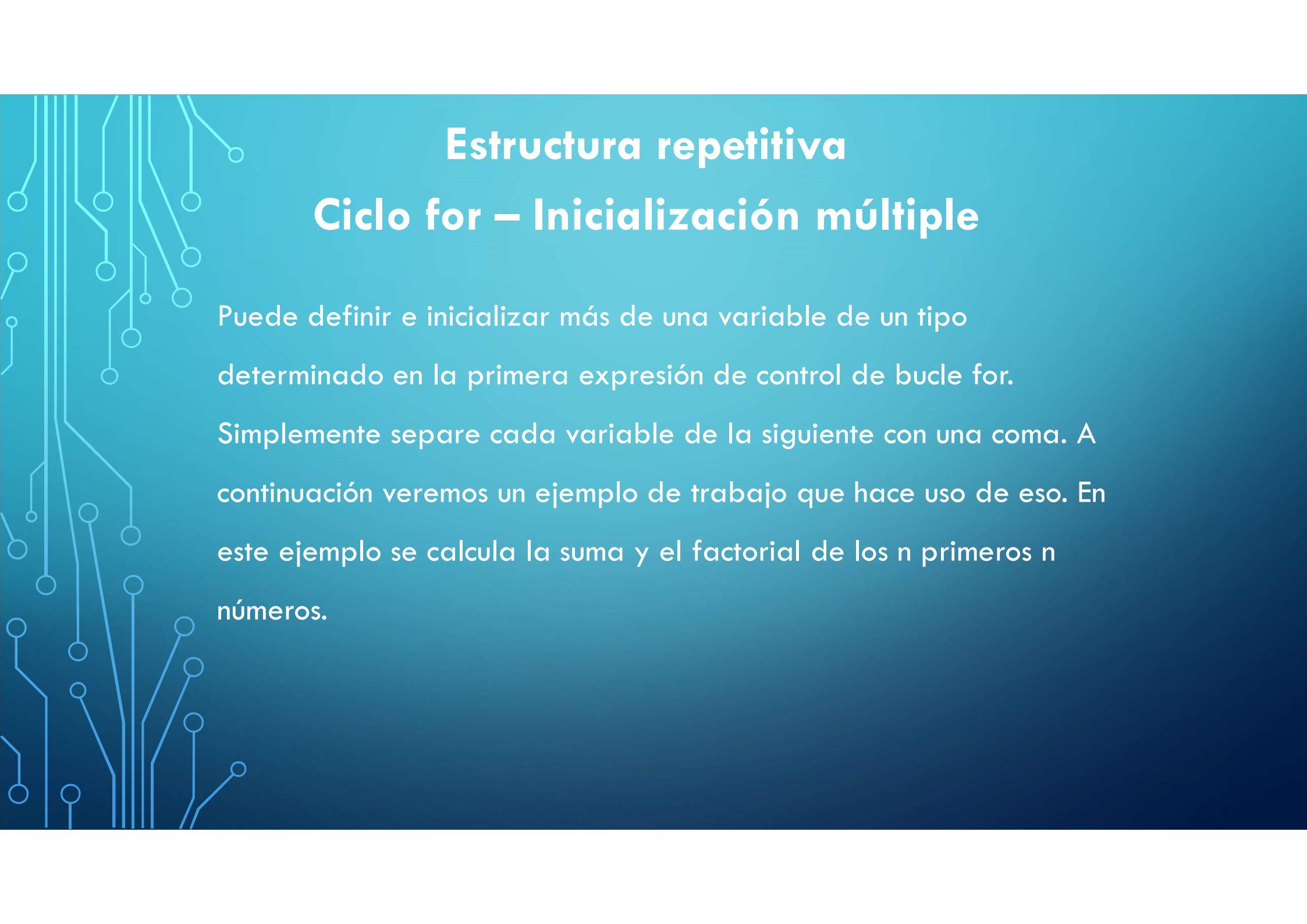
int main()
{
    const double pi{ 3.14159265 };
    const size_t perline{ 3 };           // Salidas por linea
    size_t linecount{};                // Contador de salidas por linea
    for (double radius{ 0.2 }; radius < 3.0 + .001; radius += 0.2)
    {
        std::cout << std::fixed << std::setprecision(2) << " radius =" << std::setw(5)
                  << radius << " area =" << std::setw(6) << pi * radius * radius;
        if (perline == ++linecount)       // Se debe escribir linea?..
        {
            std::cout << std::endl;      // ...salta de linea..
            linecount = 0;              // ...reset contador de lineas
        }
    }
    std::cout << std::endl;
}
```



# Estructura repetitiva – Ciclo for

## Control de punto flotante

*Comparar números de punto flotante puede ser complicado. siempre debe tener cuidado al comparar el resultado de los cálculos de punto flotante directamente con operadores como ==, <= o >=. los errores de redondeo casi siempre evitan que el valor de punto flotante sea exactamente igual al valor matemático exacto.*



## Estructura repetitiva

### Ciclo for – Inicialización múltiple

Puede definir e inicializar más de una variable de un tipo determinado en la primera expresión de control de bucle for. Simplemente separe cada variable de la siguiente con una coma. A continuación veremos un ejemplo de trabajo que hace uso de eso. En este ejemplo se calcula la suma y el factorial de los n primeros n números.

# Estructura repetitiva – Ciclo for – Inicialización múltiple

```
// Inicialización multiple en loop for.
#include <iostream>
#include <iomanip>

int main()
{
unsigned int limite {};

    std::cout << "Este programa calcula n! y la suma de los enteros"
    << " hasta n para valores desde 1 a limite.\n";
    std::cout << "Ingrese el limite de calculo: ";
    std::cin >> limite;

    // Encabezados de columnas de salida
    std::cout << std::setw(8) << "Entero" << std::setw(8) << " Suma"
        << std::setw(20) << " Factorial" << std::endl;

    for (unsigned long long n {1}, suma {}, factorial {1}; n <= limite; ++n)
    {
        suma += n; // Calcular suma para n actual.
        factorial *= n; // Calcular n! para n actual.
        std::cout << std::setw(8) << n << std::setw(8) << suma
            << std::setw(20) << factorial << std::endl;
    }
}
```

# El operador coma

Aunque la coma parece un simple separador, en realidad es un operador binario. Combina dos expresiones en una sola expresión, donde el valor de la operación es el valor de su operando derecho. Esto significa que en cualquier lugar donde pueda poner una expresión, también puede poner una serie de expresiones separadas por comas. Por ejemplo, considere las siguientes declaraciones:

```
int i {1};  
int value1 {1};  
int value2 {1};  
int value3 {1};  
std::cout << (value1 += ++i, value2 += ++i, value3 += ++i) <<  
std::endl;
```

Transcriba estas líneas en su compilador y vea los resultados.

## El operador coma

Las primeras cuatro declaraciones definen cuatro variables con un valor inicial de 1. La última declaración genera el resultado de tres expresiones de asignación que están separadas por el operador de coma. El operador de coma es asociativo a la izquierda y tiene la precedencia más baja de todos los operadores, por lo que la expresión se evalúa así:



```
((value1 += ++i), (value2 += ++i)), (value3 += ++i));
```

El efecto será que value1 se incrementará en 2 para producir 3, value2 se incrementará en 3 para producir 4 y value3 se incrementará en 4 para producir 5. El valor de la expresión compuesta es el valor de la expresión más a la derecha en la serie, por lo que el valor de salida es 5.

# El operador coma

En el ejemplo de factoriales podemos rescribir el bloque for de la siguiente manera utilizando el operador coma.

```
for (unsigned long long n {1}, sum {1}, factorial {1}; n <= limite;  
    ++n, sum += n, factorial *= n)  
{  
    std::cout << std::setw(8) << n << std::setw(8) << suma  
        << std::setw(20) << factorial << std::endl;  
}
```

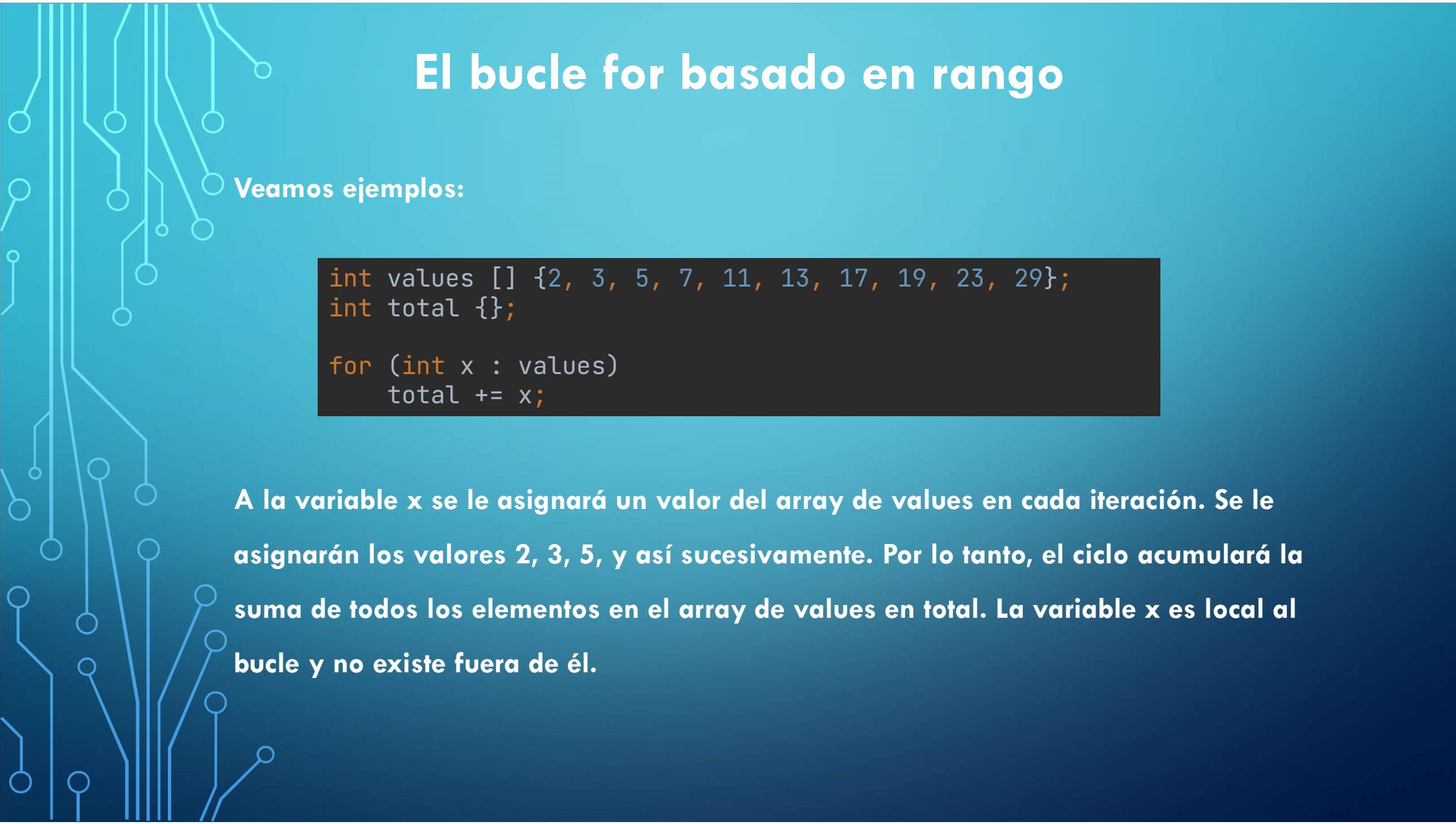
# El bucle for basado en rango

El bucle `for` basado en rango itera sobre todos los valores en un rango de valores. ¿Qué es un rango? Una array es un rango de elementos y una cadena es un rango de caracteres. Los contenedores proporcionados por la Biblioteca estándar también son todos los rangos. Esta es la forma general del bucle `for` basado en rango:

```
for (declaracion_rango : expresion_rango)
    instrucciones_bloque_for;
```

`declaracion_rango` identifica una variable a la que se le asignará cada uno de los valores en el rango, con un nuevo valor asignado en cada iteración.

`expresion_rango` identifica el rango que es el origen de los datos.



# El bucle for basado en rango

Veamos ejemplos:

```
int values [] {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};  
int total {};  
  
for (int x : values)  
    total += x;
```

A la variable **x** se le asignará un valor del array de **values** en cada iteración. Se le asignarán los valores 2, 3, 5, y así sucesivamente. Por lo tanto, el ciclo acumulará la suma de todos los elementos en el array de **values** en **total**. La variable **x** es local al bucle y no existe fuera de él.



# El bucle for basado en rango

```
int total {};
for (auto x : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29})
    total += x;
```

En este ejemplo tenemos un rango definido a través de inicialización con llaves.

Observe que utilizamos **auto**, es el compilador el que determina el tipo de datos de x en base a los valores del rango. El ejemplo de la página anterior lo podríamos reescribir de la siguiente manera.

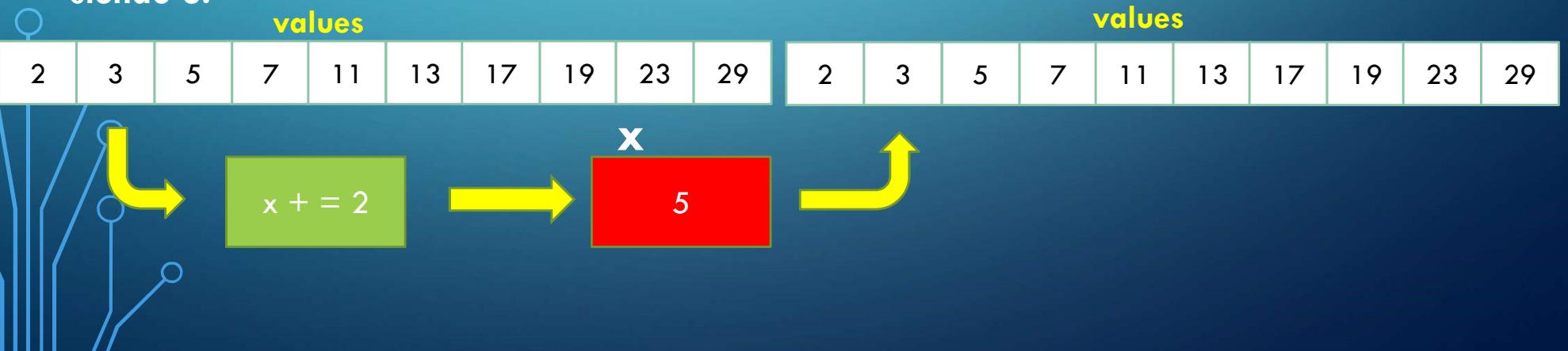
```
int values [] {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
int total {};

for (auto x : values)
    total += x;
```

# El bucle for basado en rango

```
for (auto x : values)  
    x += 2;
```

Una cuestión importante a tener en cuenta es que la expresión anterior, por ejemplo, no modifica los valores de **values**. La expresión **x+=2** modifica solo el valor de **x**, en este caso lo incrementa en 2. Supongamos que accedemos al tercer elemento de **values** (3), entonces **x** valdrá 5 luego de la expresión pero el tercer elemento de **values** seguirá siendo 3.



# El bucle while

Este tipo de bucle está gobernado por una expresión lógica, si la misma se evalúa a verdadero entonces el bloque de acciones es ejecutado. Este bucle es de tipo 0-N. Es útil cuando necesitamos plantear un ciclo que ejecute en forma repetida un bloque de acciones pero sin conocer previamente la cantidad de iteraciones a realizar.

## Sintaxis:

```
while ( condición de control)
{
    bloque de acciones
}
```

# El bucle while

Vamos a crear el mismo ejemplo de cálculo de factorial y suma de los n primeros números enteros que realizamos anteriormente pero esta vez utilizando **while**.

```
// Inicialización multiple en loop foor.  
#include <iostream>  
#include <iomanip>  
  
int main()  
{  
unsigned int limite {};  
  
    std::cout << "Este programa calcula n! y la suma de los enteros"  
        << " hasta n para valores desde 1 a limite.\n";  
    std::cout << "Ingrese el limite de calculo: ";  
    std::cin >> limite;  
  
    // Encabezados de columnas de salida  
    std::cout << std::setw(8) << "Entero" << std::setw(8) << " Suma"  
        << std::setw(20) << " Factorial" << std::endl;  
  
    std::cout << std::setw(8) << n << std::setw(8) << suma  
        << std::setw(20) << factorial << std::endl;
```

# El bucle while

```
unsigned int n {};
unsigned int sum {};
unsigned long long factorial {1ULL};

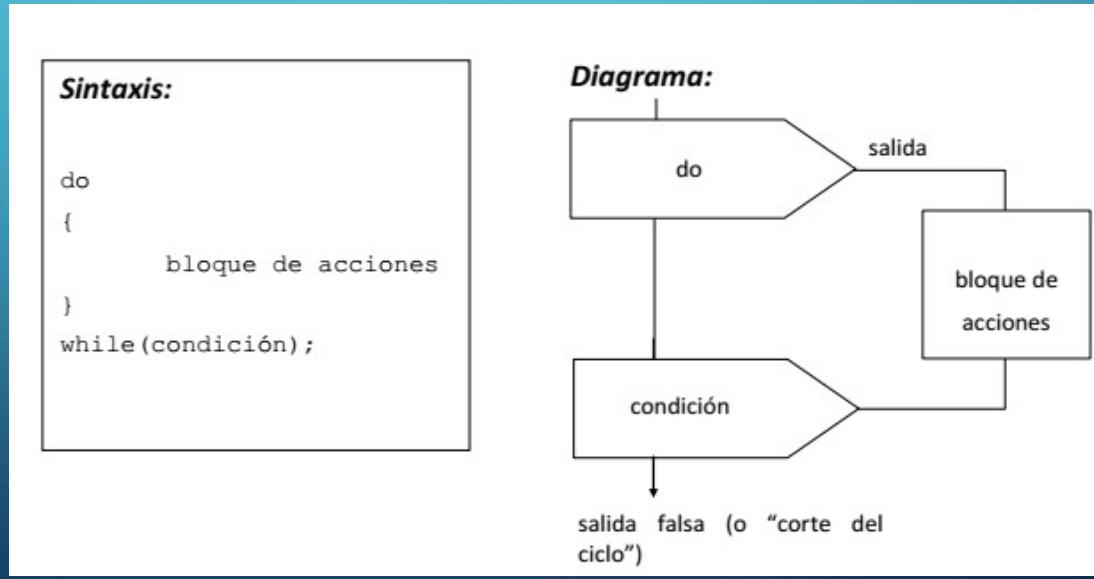
while (++n <= limite)
{
    suma += n; // // Calcular suma para n actual.
    factorial *= n; // Calcular n! para n actual.
    std::cout << std::setw(8) << n << std::setw(8) << sum
          << std::setw(20) << factorial << std::endl;
}
```

Este programa nos brinda el mismo resultado que el mismo ejemplo realizado con bucle **for**. Verifique es así ejecutándolo en su compilador.

**NOTA:** Cualquier bucle **for** puede ser reemplazado por un bucle **while** y viceversa.

# El bucle do-while

La característica básica del **ciclo do while** es que la condición de control se evalúa por primera vez después de ejecutar por primera vez el bloque de acciones. Esto implica que *al menos una vez* el bloque de acciones del ciclo siempre será ejecutado, independientemente del valor inicial de la condición de corte (por eso este ciclo es del tipo **1-N**: una vez como mínimo se ejecuta el bloque, y luego puede llegar a  $N$  repeticiones).



# El bucle do-while

*Ejemplo:*

```
/*
 * Ejemplo utilización do-while
 * para leer temperaturas hasta que
 * el usuario seleccione n.
 */

#include <iostream>
#include <cctype> // Para utilizar tolower()

int main()
{
    char respuesta {};// Almacena la respuesta y/n
    int contador {};// Cuenta el numero de valores ingresados
    double temperatura {};// Almacena el valor de entrada
    double total {};// Almacena la suma de todos los valores de entrada
```

# El bucle do-while

```
do
{
    std::cout << "Ingrese una lectura de temperatura: " // Solicitud de entrada
    std::cin >> temperatura; // Lee valor de entrada

    total += temperatura; // Acumular el total de valores
    ++contador; // Incrementar contador valores

    std::cout << "Desea ingresar otro valor? (y/n): ";
    std::cin >> respuesta; // Obtiene respuesta.

} while (std::tolower(respuesta) == 'y');

std::cout << "La temperatura promedio es " << total/contador << std::endl;
```



## Loops - continue

La instrucción **continue** se utiliza para omitir la iteración actual del bucle y el control del programa pasa a la siguiente iteración. La instrucción **continue** puede ubicarse dentro del cuerpo de un **if** o dentro del cuerpo de un **else**.



# Loops - continue

## Ejemplo de uso de **continue**.

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

int main() {

    for (unsigned int count{1}; count <= 10; count++) { // loop 10
times
        if (count == 5) {
            continue; // Si count es 5 omitir codigo restante y
saltar a proxima iteracion.
        }
        cout << count << " ";
    }

    cout << "\nSe utilizo continue para evitar imprimir 5." << endl;
}
```

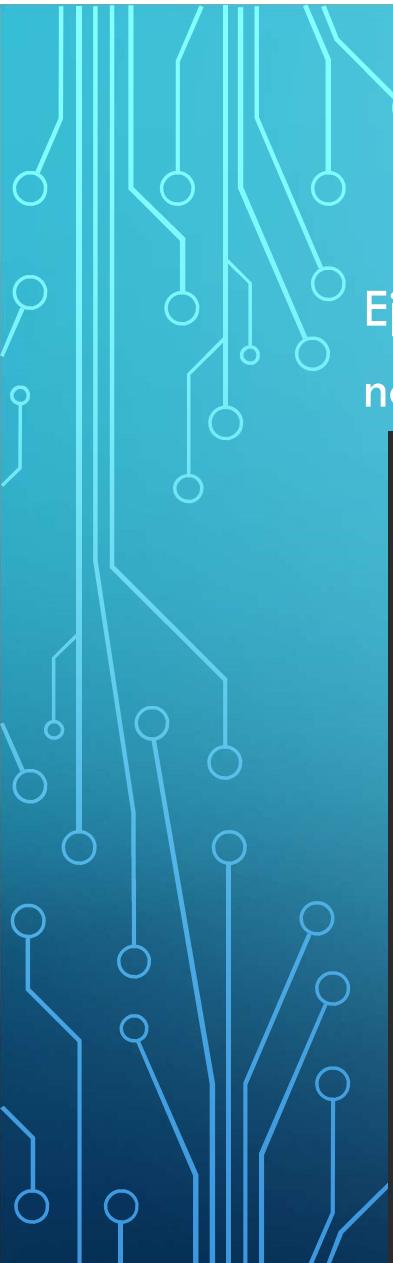
## Loops - continue

Ejemplo de uso de **continue**. Imprimir caracteres ASCII evitando los no imprimibles

```
// Usando la instrucción continue para mostrar códigos ASCII.

#include <iostream>
#include <iomanip>
#include <cctype>
#include <limits>

int main()
{
    // Output the column headings
    std::cout << std::setw(11) << "Caracter " <<
    std::setw(13) << "Hexadecimal "
                  << std::setw(9) << "Decimal " << std::endl;
    std::cout << std::uppercase; // Dígitos hex en mayúscula.
```



## Loops - continue

Ejemplo de uso de **continue**. Imprimir caracteres ASCII evitando los no imprimibles

```
// Mostrar caracteres y códigos correspondientes
unsigned char ch {};

do
{
    if (!std::isprint(ch)) // Si es carácter es no
imprimible.
        continue; // ...omitar esta iteración.

    std::cout << std::setw(6) << ch // Character
        << std::hex << std::setw(12) <<
static_cast<int>(ch) // Hexadecimal
        << std::dec << std::setw(10) <<
static_cast<int>(ch) // Decimal
        << std::endl;
} while (ch++ < std::numeric_limits<unsigned char>::max());
```

# Loops - break

La instrucción **break** se utiliza para finalizar el loop. Puede ubicarse dentro del cuerpo de un **if** o dentro del cuerpo de un **else**.

```
while (test expression) {  
    statement/s  
    if (test expression) {  
        break;  
    }  
    statement/s  
}
```

```
do {  
    statement/s  
    if (test expression) {  
        break;  
    }  
    statement/s  
}  
while (test expression);
```

```
for (initial expression; test expression; update expression) {  
    statement/s  
    if (test expression) {  
        break;  
    }  
    statements/  
}
```

# Loops - break

Ejemplo de uso de **break**.

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

int main() {

    unsigned int count {}; // variable de control

    for ( count = 1; count <= 10; count++) { // loop 10 times
        if (count == 5) {
            break; // Finalizar el bucle si count es 5.
        }

        cout << count << " ";
    }

    cout << "\nSalió del bucle en el conteo = " << count << endl;
}
```

# Ejercitación

**Ejercicio:** Crear un programa que genere las tablas de multiplicar. Para ello le solicitará al usuario que ingrese el tamaño de la tabla que desea, por ejemplo si ingresa 4 generaría las tablas desde 1 al tamaño ingresado. El tamaño mínimo de la tabla debe ser 2 y el máximo de 12. La siguiente imagen muestra el resultado esperado para una tabla de tamaño 7.

Ingrese tamaño de la tabla (entre 2 y 12): 7							
	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	2	4	6	8	10	12	14
3	3	6	9	12	15	18	21
4	4	8	12	16	20	24	28
5	5	10	15	20	25	30	35
6	6	12	18	24	30	36	42
7	7	14	21	28	35	42	49

# Ejercitación

**Ejercicio:** Crear un programa que genere la siguiente salida.

Demostracion de bucles anidados

F: 1 C: 1	F: 1 C: 2	F: 1 C: 3	F: 1 C: 4	F: 1 C: 5
F: 2 C: 1	F: 2 C: 2	F: 2 C: 3	F: 2 C: 4	F: 2 C: 5
F: 3 C: 1	F: 3 C: 2	F: 3 C: 3	F: 3 C: 4	F: 3 C: 5
F: 4 C: 1	F: 4 C: 2	F: 4 C: 3	F: 4 C: 4	F: 4 C: 5
F: 5 C: 1	F: 5 C: 2	F: 5 C: 3	F: 5 C: 4	F: 5 C: 5
F: 6 C: 1	F: 6 C: 2	F: 6 C: 3	F: 6 C: 4	F: 6 C: 5
F: 7 C: 1	F: 7 C: 2	F: 7 C: 3	F: 7 C: 4	F: 7 C: 5

# Ejercitación

**Ejercicio:** Diseñe un programa que simule el lanzamiento de 2 dados. Ejecute 50 lanzamientos, sume el valor obtenido en ambos dados y almacene dicha suma en un vector y luego imprima los valores tabulados como se muestra.

Lanzamiento	Resultado
1	2
2	7
...	
50	5

```
#include <random>
#include <iostream>

int main(){

    unsigned int rand_num;

    // Produce valores pseudo aleatorios uniformemente distribuidos
    std::default_random_engine engine{static_cast<unsigned int>(time(0))};
    std::uniform_int_distribution<unsigned int> randomInt{1, 6};

    ...
    rand_num = randomInt(engine);
    ...

}
```



## Loops Indefinidos

Un bucle indefinido puede potencialmente ejecutarse para siempre. Omitir la segunda expresión de control en un bucle `for` da como resultado un bucle que potencialmente ejecuta un número ilimitado de iteraciones. Tiene que haber alguna forma de terminar el bucle dentro del mismo bloque de bucle; de lo contrario, el ciclo se repite indefinidamente. Los bucles indefinidos tienen muchos usos prácticos, como programas que monitorean algún tipo de indicador de alarma, por ejemplo, o que recopilan datos de sensores en una planta industrial. Un ciclo indefinido puede ser útil cuando no sabe de antemano cuántas iteraciones de ciclo se requerirán, como se cuando está leyendo una cantidad variable de datos de entrada. En estas circunstancias, la salida del ciclo se codifica dentro del bloque del ciclo, no dentro de la expresión de control del ciclo.



# Loops Indefinidos

En la forma más común del bucle **for** indefinido, se omiten todas las expresiones de control, como se muestra en 1

También podemos generar un bucle **while** indefinido como se muestra en el segundo bloque de código. 2

```
for (;;)
{
    // Instrucciones para realizar alguna tarea.
    // ... incluir alguna forma de romper el ciclo. 1

while (true)
{
    // Instrucciones para realizar alguna tarea.
    // ... incluir alguna forma de romper el ciclo. 2
```

# Arrays de Caracteres

Un array de caracteres puede ser simplemente un array de caracteres, en la que cada elemento almacena un carácter, o puede representar una cadena. En el último caso, los caracteres de la cadena se almacenan en elementos de array sucesivos, seguidos de un carácter especial de terminación de cadena denominado carácter nulo que se escribe como '\0' que marca el final de la cadena. Un array caracteres que termina en '\0' se denomina **cadena de estilo C**.

```
char vocales[5] {'a', 'e', 'i', 'o', 'u'}; // array de caracteres
```

```
char vocales[6] {'a', 'e', 'i', 'o', 'u'}; // array de caracteres o cadena estilo C
```

En este ultimo ejemplo como se indica una dimensión mayor a la cantidad de caracteres, se inserto un cero en la ultima posición, por tanto se puede procesar como array de caracteres o como una cadena estilo C.

# Arrays de Caracteres

Un array de caracteres puede ser simplemente un array de caracteres, en la que cada elemento almacena un carácter, o puede representar una cadena. En el último caso, los caracteres de la cadena se almacenan en elementos de array sucesivos, seguidos de un carácter especial de terminación de cadena denominado carácter nulo que se escribe como '\0' que marca el final de la cadena. Un array caracteres que termina en '\0' se denomina **cadena de estilo C**.

```
char vocales[5] {'a', 'e', 'i', 'o', 'u'}; // array de 5 caracteres
```

```
char vocales[6] {'a', 'e', 'i', 'o', 'u'}; // array de 5 caracteres o cadena estilo C
```

En este ultimo ejemplo como se indica una dimensión mayor a la cantidad de caracteres, se inserto un cero ('\0') en al última posición, por tanto se puede procesar como array de caracteres o como una cadena estilo C.

# Arrays de Caracteres

Puede dejar que el compilador establezca el tamaño del array en la cantidad de valores de inicialización:

```
char vocales[] {'a', 'e', 'i', 'o', 'u'}; // array de 5 caracteres
```

También puede declarar un array de tipo char e inicializarlo con un literal de cadena, de la siguiente manera:

```
char nombre[10] {"Mae West"}; // Cadena estilo C.
```

No existe valor inicial para este elemento \_\_\_\_\_

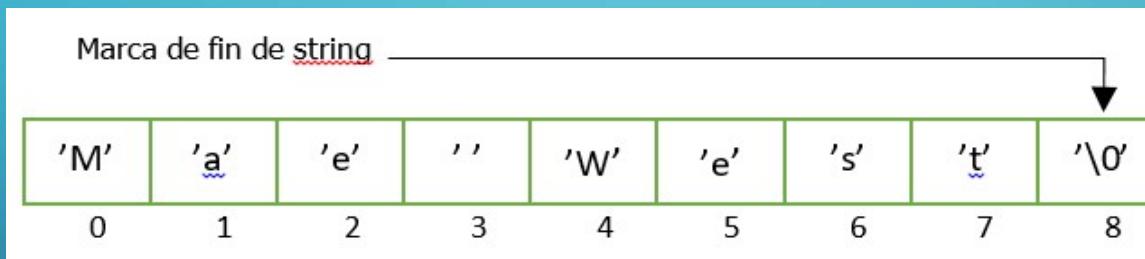
Marca de fin de string \_\_\_\_\_

'M'	'a'	'e'	' '	'W'	'e'	's'	't'	'\0'	'\0'
0	1	2	3	4	5	6	7	8	9

# Arrays de Caracteres

En realidad es innecesario indicar la dimensión del array para una cadena estilo C.  
Lo anterior se presentó a modo de ejemplo. La forma adecuada sería:

```
char nombre[] {"Mae West"}; // Cadena estilo C.
```



Puede imprimir una cadena almacenada en un array simplemente usando el nombre del array. La cadena en el array nombre, por ejemplo, podría imprimirse de la siguiente forma:

```
std::cout << nombre << std::endl;
```

# Arrays de Caracteres

Dado que el último carácter de una cadena estilo C es cero, podría acceder al conjunto de caracteres de la siguiente forma:

```
for (int i {}; text[i]; i++)  
{  
    ...
```

## Alternativas al uso de array

El estilo de array que hemos visto es una herencia del viejo C. La biblioteca estándar nos provee de dos contenedores: `std::array<>` y `std::vector<>`. Estos forman una alternativa directa a los array sencillos integrados en el lenguaje C++, pero es mucho más fácil trabajar con ellos, son mucho más seguros de usar y proporcionan una flexibilidad significativamente mayor que los array integrados de bajo nivel.

`std::array<>` y `std::vector<>` se definen como plantillas de clase (**templates**) tema que veremos mas adelante.

El compilador usa las plantillas `std::array<T,N>` y `std::vector<T>` para crear un tipo concreto basado en lo que se especifica en los parámetros de la plantilla, **T** y **N**. Por ejemplo, si define una variable de escriba `std::vector<int>`, el compilador generará una clase de contenedor `vector<>` que está diseñada específicamente para contener y manipular un array de valores `int`. El poder de las plantillas radica en el hecho de que se puede utilizar cualquier tipo **T**.

## Alternativas al uso de array

Por ahora utilizaremos estos dos contenedores con tipos primitivos, pero dada su naturaleza pueden utilizarse también con tipos definidos por el usuario.

Para utilizar **std::array** debemos incluir el encabezado **array**:

```
#include <array>
```

Así es como se crea un **array<>** de 100 elementos de tipo **double**:

```
std::array<double, 100> values;
```

```
std::array<double, 100> values {0.5, 1.0, 1.5, 2.0}; // el elemento 5 y los  
// subsequentes son 0.0.
```

```
std::array<double, 100> values {}; // Inicializa a cero los 100 elementos
```

## Alternativas al uso de array

También puede establecer fácilmente todos los elementos en cualquier otro valor dado usando la función **fill()** para el objeto **array<>**. La función es un miembro de la clase **array**. Aquí hay un ejemplo:

```
values.fill(3.14159265358979323846); // Establecer todos los elementos en pi
```

La función **size()** para un objeto **array<>** devuelve el número de elementos como tipo **size\_t**. Con la misma variable de valores de antes, la siguiente declaración da como resultado 100:

```
std::cout << values.size() << std::endl;
```

Puede acceder a los elementos del **array<>** de la misma manera que un array tipo C. Debido a que un objeto **array<>** es un rango, puede usar el bucle **for basado en rango**. *Ejemplo:*

# Alternativas al uso de array

Suma los elementos del array<>:

```
double total {};
for (auto value : values)
{
    total += value;
}
```

Uno de los importantes problemas que posee un array tipo C es que podemos por accidente intentar acceder a un elemento fuera del rango, ya describimos lo que puede ocurrir si escribimos fuera de dicha rango.

Acceder a los elementos de un objeto **array**<> mediante un índice entre corchetes no comprueba si hay valores de índice no válidos. La función **at()** para un objeto **array**<> lo hace y, por lo tanto, detectará los intentos de usar un valor de índice fuera del rango legítimo. El argumento de la función **at()** es un índice, al igual que cuando usa corchetes, por lo que podría escribir el ciclo for que totaliza los elementos de esta manera:

# Alternativas al uso de array

Suma los elementos del **array**:

```
double total {};
for (size_t i {}; i < values.size(); ++i)
{
    total += values.at(i);
}
```

La expresión **values.at(i)** es equivalente a **values[i]** pero con la seguridad añadida de que se comprobará el valor de i. Por ejemplo, este código fallará:

```
double total {};
for (size_t i {}; i <= values.size(); ++i)
{
    total += values.at(i);
}
```

## Alternativas al uso de array

La **plantilla array<>** también ofrece funciones convenientes para acceder al primer y último elemento. Dado el `array<> values`, la expresión **values.front()** es equivalente a **values[0]**, y **values.back()** es equivalente a **values[values.size() - 1]**.

**Ejercicio:** Cree un `array<>` del tipo que guste y pruebe los métodos antes mencionados.

# Alternativas al uso de array

## Operaciones en array<> como un todo

Puede comparar contenedores **array<>** completos utilizando cualquiera de los operadores de comparación, siempre que los contenedores tengan el mismo tamaño y almacenen elementos del mismo tipo. *Ejemplo:*

```
std::array<double, 4> array1 {1.0, 2.0, 3.0, 4.0};  
std::array<double, 4> array2 {1.0, 2.0, 3.0, 4.0};  
std::array<double, 4> array3 {1.0, 1.0, 5.0, 5.0};  
  
if (array1 == array2) std::cout << "array1 y array2 son iguales." << std::endl;  
if (array2 != array3) std::cout << "array2 y array3 no son iguales." << std::endl;  
if (array2 > array3) std::cout << "array2 es mayor que array3." << std::endl;  
if (array3 < array2) std::cout << "array3 es menor que array2." << std::endl;
```

# Alternativas al uso de array

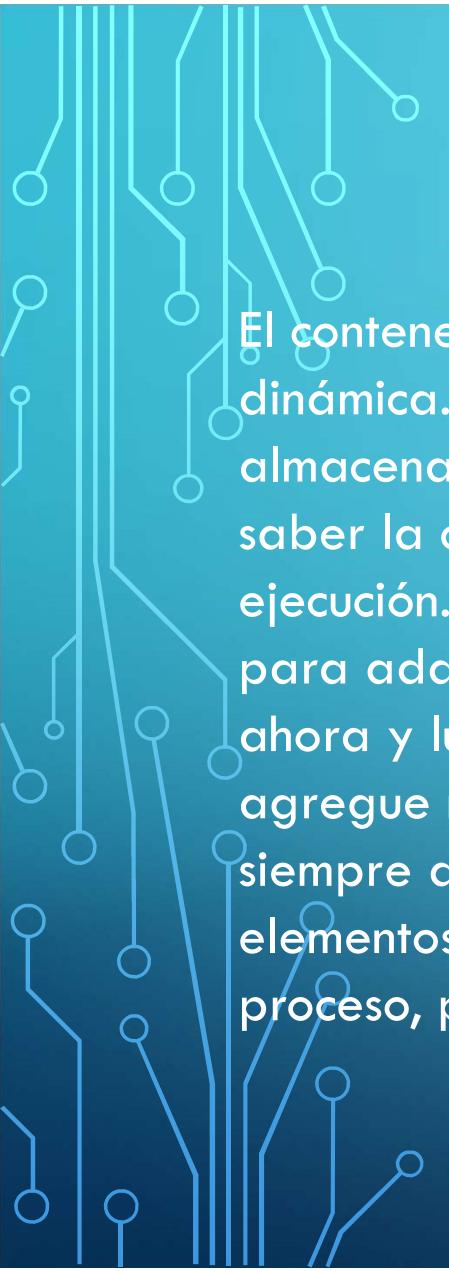
## Operaciones en array<> como un todo

También puede asignar un contenedor de `array<>` a otro, siempre que ambos almacenen la misma cantidad de elementos del mismo tipo. *Ejemplo:*

```
// Copia de std::arrays<>
std::array<unsigned int,HEIGHT_SIZE> height {26, 37, 47, 55, 62, 75};
std::array<unsigned int,HEIGHT_SIZE> copy{};

copy = height;
```

**NOTA:** Incluso si en su código usa contenedores `std::array<>`, sigue siendo perfectamente posible llamar a funciones heredadas que esperan array simples como entrada. Siempre puede acceder al array integrado que está encapsulado dentro del objeto `array<>` utilizando su miembro `data()`.



## Contenedores std::vector<T>

El contenedor `vector<T>` es un contenedor de secuencias. Es una estructura dinámica. No es necesario saber la cantidad de elementos que un `vector<>` almacenará de antemano, en tiempo de compilación. Tampoco es necesario saber la cantidad de elementos que almacenará de antemano, en tiempo de ejecución. Es decir, el tamaño de un `vector<>` puede crecer automáticamente para adaptarse a cualquier número de elementos. Puede agregar algunos ahora y luego algunos mucho después. Un `vector<>` crecerá a medida que agregue más y más elementos; el espacio adicional se asigna automáticamente siempre que sea necesario. Tampoco existe un número máximo real de elementos, además del impuesto por la cantidad de memoria disponible para su proceso, por supuesto, razón por la cual solo necesita el parámetro de tipo `T`.

## Contenedores std::vector<T>

El uso del contenedor **vector<>** necesita que el encabezado `vector` se incluya en su archivo fuente. Aquí hay un ejemplo de cómo crear un `vector<>` contenedor para almacenar valores de tipo `double`:

```
std::vector<double> values; // Forma típica. No se reserva espacio  
                           // para sus elementos
```

En su forma típica, no tiene espacio reservado para sus elementos, por lo que la memoria deberá asignarse dinámicamente cuando agregue el primer elemento de datos. Puede agregar un elemento usando la función `push_back()` para el objeto contenedor. **Ejemplo:**

```
values.push_back(3.1415); // Agrega un elemento al final del vector
```

## Contenedores std::vector<T>

La función **push\_back()** agrega el valor que pasa como argumento, 3.1415 en este caso, como un nuevo elemento al final de los elementos existentes. Dado que no hay elementos existentes aquí, este será el primero, lo que probablemente hará que se asigne memoria por primera vez. Puede inicializar un `vector<>` con un número predefinido de elementos, como este:

```
std::vector<double> values(20); // Vector contiene 20 valores dobles -  
// todos cero
```

A diferencia de un vector integrado o un objeto `array<>`, un contenedor `vector<>` siempre inicializa sus elementos. En este caso, nuestro contenedor comienza con 20 elementos que se inicializan con cero. Si no le gusta cero como valor predeterminado para sus elementos, puede especificar otro valor explícitamente:

```
std::vector<long> números(20, 99L); // Vector contiene 20 valores tipo  
// long (99)
```

## Contenedores std::vector<T>

El segundo argumento entre paréntesis especifica el valor inicial de todos los elementos, por lo que los 20 elementos serán 99L. A diferencia de la mayoría de los otros tipos de array que ha visto hasta ahora, el primer argumento que especifica la cantidad de elementos, 20 en nuestro ejemplo, no necesita ser una expresión constante. Podría ser el resultado de una expresión ejecutada en tiempo de ejecución o leída desde el teclado. Por supuesto, puede agregar nuevos elementos al final de este o cualquier otro vector usando la función **push\_back()**. Otra opción para crear un **vector<>** es usar una lista entre llaves para especificar los valores iniciales:

```
std::vector<unsigned int> primos { 2, 3, 5, 7, 11, 13, 17, 19 };
```

El vector primos se creará con ocho elementos con los valores iniciales dados.

# Contenedores std::vector<T>

**PRECAUCION:** Es posible que haya notado antes que no inicializamos los valores y números de los objetos `vector<>` usando la sintaxis de inicializador entre llaves habitual, sino que usamos paréntesis:

```
std::vector<double> values(20); // Vector contiene 20 valores dobles -  
                                // todos cero  
  
std::vector<long> números(20, 99L); // Vector contiene 20 valores tipo  
                                    // long - todos 99
```

Esto se debe a que el uso de inicializadores entre llaves aquí tiene un efecto significativamente diferente, como explican los comentarios junto a las declaraciones:

```
std::vector<double> values{20}; // Vector values contiene un 1 solo valor double: 20  
std::vector<long> numbers{20, 99L}; // Vector number contiene 2 valores long: 20 y 99
```

Cuando usa llaves para inicializar un `vector<>`, el compilador siempre lo interpreta como una secuencia de valores iniciales. Esto es para adaptarse a la inicialización de vectores de la misma manera que lo hizo antes con array regulares o contenedores `array <>`:

```
std::vector<int> six_initial_values{ 7, 9, 7, 2, 0, 4 };
```



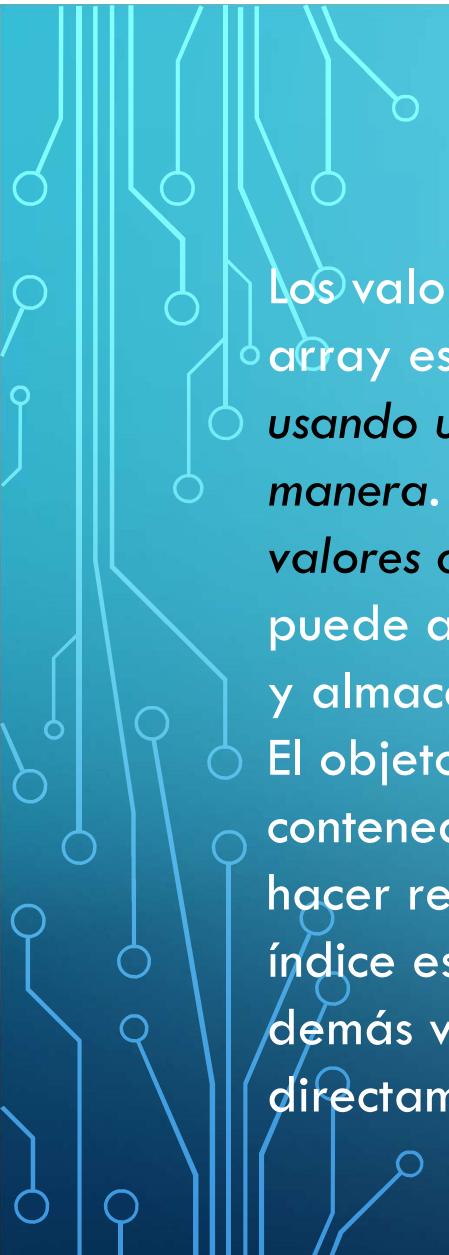
## Contenedores std::vector<T>

*Para inicializar un **vector**<> con un número dado de valores idénticos, sin repetir el mismo valor una y otra vez, es decir, no puede usar llaves. Si lo hace, el compilador lo interpreta como una lista de uno o dos valores iniciales.*

## Contenedores std::vector<T>

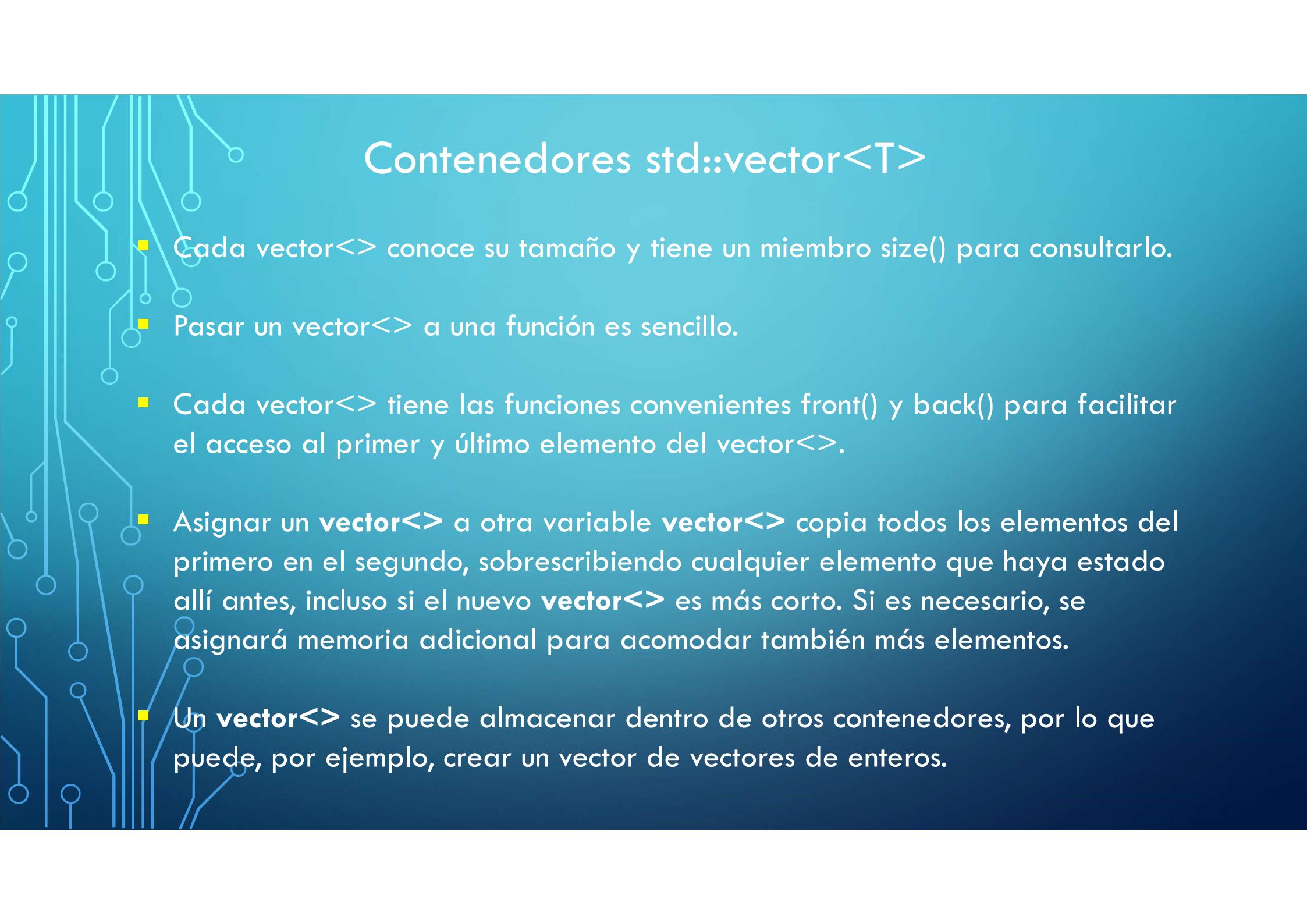
Al igual que los array, puede usar un índice entre corchetes para establecer un valor para un elemento existente o simplemente usar su valor actual en una expresión. **Ejemplo:**

```
values[0] = 3.14159265358979323846; // Pi
values[1] = 5.0; // Radio de un circulo
values[2] = 2.0*values[0]*values[1]; // Circunferencia de un circulo
```



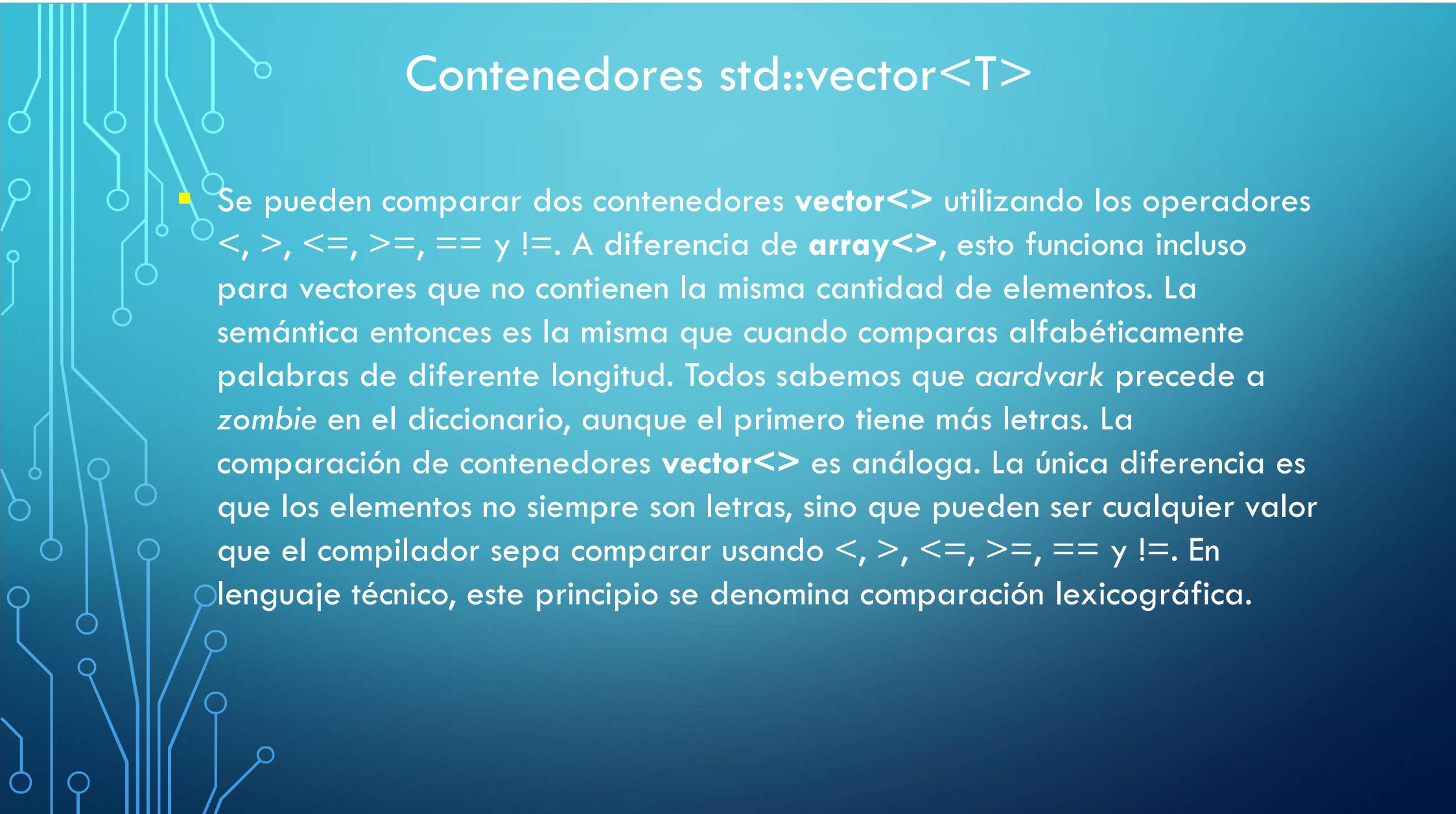
## Contenedores `std::vector<T>`

Los valores de índice para un `vector<>` comienzan desde 0, al igual que un `array` estándar. Siempre puede hacer referencia a elementos existentes usando un índice entre corchetes, pero no puede crear nuevos elementos de esta manera. Para eso, necesitas usar, por ejemplo, la función `push_back()`. Los valores de índice no se verifican cuando indexa un vector como este. Por lo tanto, puede acceder accidentalmente a la memoria fuera de la extensión del vector y almacenar valores en dichas ubicaciones utilizando un índice entre corchetes. El objeto `vector<>` también proporciona la función `at()`, al igual que un objeto contenedor `array<>`, por lo que podría considerar usar la función `at()` para hacer referencia a elementos siempre que exista la posibilidad de que el índice esté fuera del rango legal. Además de la función `at()`, casi todas las demás ventajas de los contenedores `array<>` también se transfieren directamente a `vector<>`:



## Contenedores std::vector<T>

- Cada `vector<>` conoce su tamaño y tiene un miembro `size()` para consultararlo.
- Pasar un `vector<>` a una función es sencillo.
- Cada `vector<>` tiene las funciones convenientes `front()` y `back()` para facilitar el acceso al primer y último elemento del `vector<>`.
- Asignar un `vector<>` a otra variable `vector<>` copia todos los elementos del primero en el segundo, sobrescribiendo cualquier elemento que haya estado allí antes, incluso si el nuevo `vector<>` es más corto. Si es necesario, se asignará memoria adicional para acomodar también más elementos.
- Un `vector<>` se puede almacenar dentro de otros contenedores, por lo que puede, por ejemplo, crear un vector de vectores de enteros.



## Contenedores std::vector<T>

Se pueden comparar dos contenedores `vector<T>` utilizando los operadores `<, >, <=, >=, ==` y `!=`. A diferencia de `array<T>`, esto funciona incluso para vectores que no contienen la misma cantidad de elementos. La semántica entonces es la misma que cuando comparas alfabéticamente palabras de diferente longitud. Todos sabemos que *aardvark* precede a *zombie* en el diccionario, aunque el primero tiene más letras. La comparación de contenedores `vector<T>` es análoga. La única diferencia es que los elementos no siempre son letras, sino que pueden ser cualquier valor que el compilador sepa comparar usando `<, >, <=, >=, ==` y `!=`. En el lenguaje técnico, este principio se denomina comparación lexicográfica.

# Contenedores std::vector<T>

Un **vector<>** no tiene un miembro **fill()**. En cambio, ofrece funciones de **assign()** que se pueden usar para reinicializar el contenido de un **vector<>**, como lo haría al inicializarlo por primera vez:

```
std::vector<long> numbers(20, 99L); // Vector contiene 20 valores long - todos 99  
numbers.assign(99, 20L);           // Vector contiene 99 valores long - todos 20  
numbers.assign({99L, 20L});        // Vector contiene 2 valores long - 99 y 20
```

Puede eliminar todos los elementos de un **vector<>** llamando a la función **clear()** para el objeto vector. *Ejemplo:*

## Como borrar elementos

Puede eliminar todos los elementos de un **vector<>** llamando a la función **clear()** para el objeto vectorial. *Ejemplo:*

```
std::vector<int> data(100, 99); // Contiene 100 elementos inicializados a 99  
data.clear();                  // Remueve todos los elementos
```

## Contenedores std::vector<T>

**CUIDADO:** Tanto `vector<>` como `array<>` también proporcionan una función de `empty()`, que a veces se llama incorrectamente en un intento de borrar un `vector<>`. Pero `empty()` no vacía un `vector<>`; `clear()` lo hace. En su lugar, el miembro `empty()` comprueba si un contenedor determinado está vacío. Es decir, se evalúa como el valor booleano verdadero si y solo si el contenedor no contiene elementos y no modifica el contenedor en absoluto en el proceso.

## Contenedores std::vector<T>

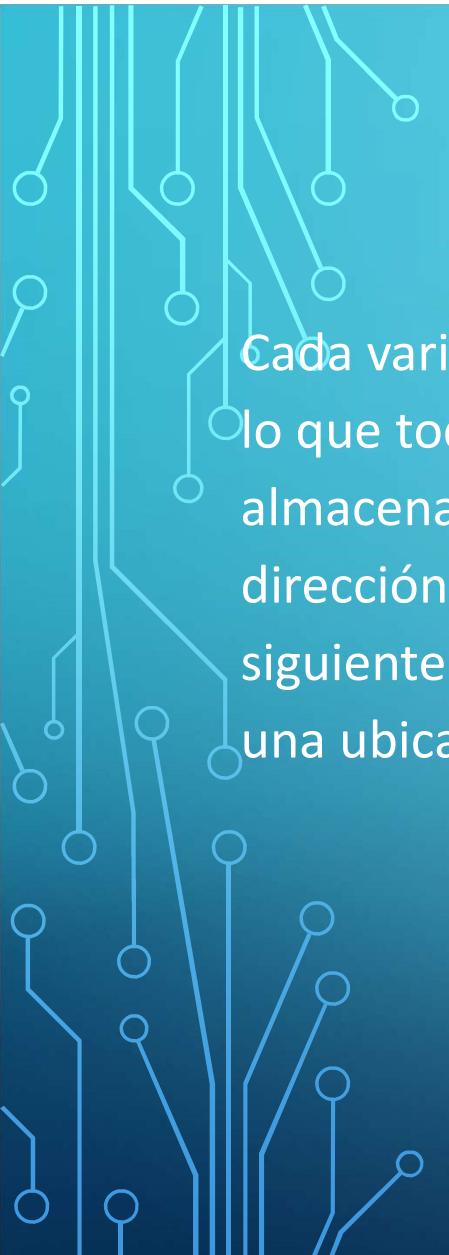
Puede eliminar el último elemento de un objeto vectorial llamando a su función **pop\_back()**. *Ejemplo:*

```
std::vector<int> data(100, 99); // Contiene 100 elementos inicializados a 99  
data.pop_back(); // Remueve el ultimo elemento
```

La segunda instrucción elimina el último elemento, por lo que el tamaño de los datos será 99. Esto no es todo lo que hay que hacer para usar contenedores **vector<>**. Por ejemplo, solo le mostramos cómo agregar o eliminar elementos del final de un **vector<>**, mientras que es perfectamente posible insertar o eliminar elementos en posiciones arbitrarias. Cuando veamos contenedores exploraremos mas el contenedor **vector<>**.

# Contenedores std::vector<T>

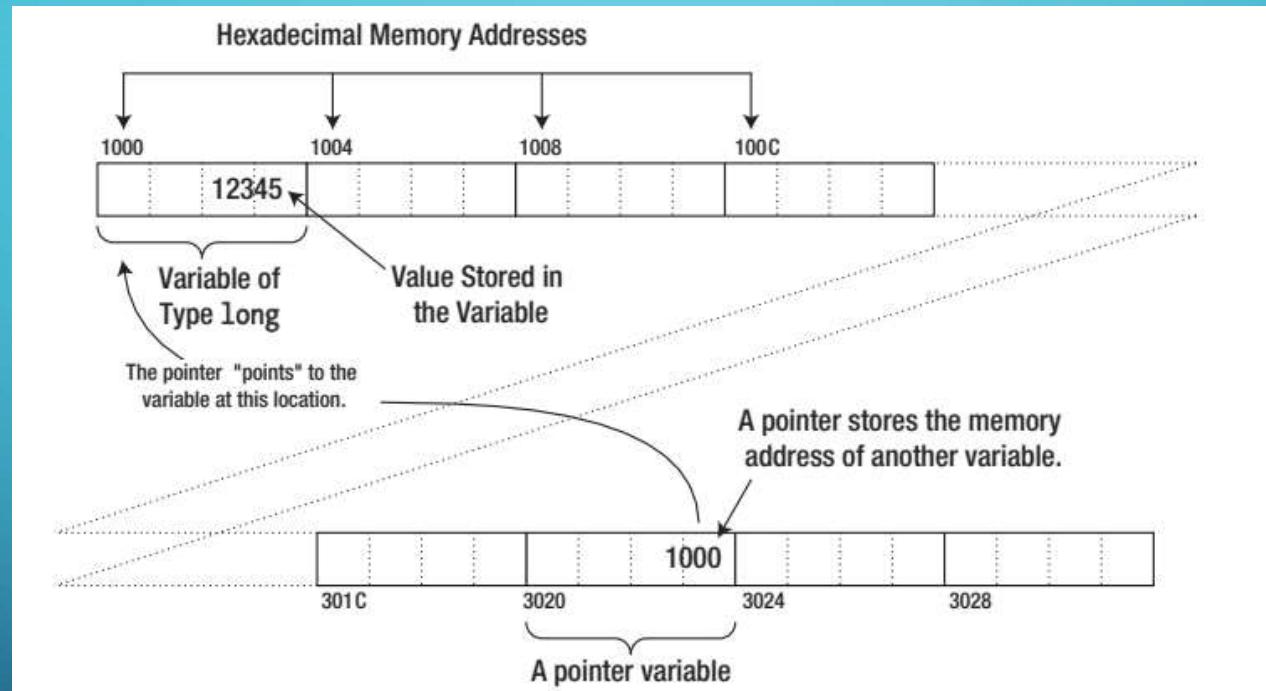
Métodos de la clase vector	
assign	asigna los elementos al vector
at	devuelve el elemento de una posición específica
back	devuelve una referencia al último elemento del vector
begin	devuelve un iterador al principio del vector
capacity	devuelve el número de elementos que pueden ser contenidos por el vector
clear	elimina todos los elementos del vector
empty	true si el vector está vacío
end	devuelve un iterador al final del vector
erase	elimina elementos del vector
front	regresa una referencia al primer componente del vector
insert	insertar componentes en el vector
max_size	regresa el número máximo de elementos soportados por el vector
pop_back	elimina el último elemento del vector
push_back	añade un elemento al final del vector
rbegin	devuelve un reverse_iterator hacia el final del vector
rend	devuelve un reverse_iterator hacia el inicio del vector
reserve	establece la capacidad mínima del vector
resize	cambia el tamaño del vector
size	devuelve el número de componentes en el vector
swap	intercambia el contenido de un vector con el de otro



## PUNTEROS Y REFERENCIAS

Cada variable en su programa está ubicada en algún lugar de la memoria, por lo que todas tienen una dirección única que identifica dónde están almacenadas. Un puntero es una variable que puede almacenar una dirección de otra variable, de algún dato en otra parte de la memoria. La siguiente figura muestra cómo un puntero obtiene su nombre: "apunta a" una ubicación en la memoria donde se almacena algún otro valor.

# PUNTEROS Y REFERENCIAS





## PUNTEROS Y REFERENCIAS

Como sabe, un número entero tiene una representación diferente de un valor de punto flotante, y la cantidad de bytes que ocupa un elemento de datos depende del tipo de dato. Entonces, para usar un elemento de datos almacenado en la dirección contenida en un puntero, *necesita saber el tipo de datos*. Sin conocer el tipo de datos, un puntero no sirve de mucho. Por lo tanto, cada puntero apunta a un tipo particular de elemento de datos en esa dirección. La definición de un puntero es similar a la de una variable ordinaria excepto que el nombre del tipo tiene un asterisco a continuación para indicar que es un puntero y no una variable de ese tipo. Así es como se define un puntero llamado **pnumber** que puede almacenar la dirección de una variable de tipo **long**:

```
long* pnumber {}; // Define un puntero a un tipo long.  
Long *pnumber {}; // Esta es la forma mas legible y recomendada.
```

# PUNTEROS Y REFERENCIAS

El tipo de **pnumber** es "puntero a long", que se escribe como **long\*** o **long \***. El compilador acepta cualquiera de las dos notaciones. Este puntero solo puede almacenar una dirección de una variable de tipo long. Un intento de almacenar la dirección de una variable que no sea de tipo largo no se compilará. Debido a que el inicializador está vacío, la declaración inicializa **pnumber** con el puntero equivalente a cero, que es una dirección especial que no apunta a nada. Este valor de puntero especial se escribe como **nullptr**, y podría especificarlo explícitamente como el valor inicial:

```
long* pnumber {nullptr};  
long *pnumber {nullptr};
```



# PUNTEROS Y REFERENCIAS

No está obligado a inicializar un puntero cuando lo define, pero es imprudente no hacerlo. Los punteros no inicializados son más peligrosos que las variables ordinarias que no están inicializadas.

Por lo tanto:

*Siempre debe inicializar un puntero cuando lo define. Si aún no puede darle el valor deseado, inicialice el puntero a `nullptr`.*

# PUNTEROS Y REFERENCIAS

Existe la posibilidad de confusión si mezcla definiciones de variables ordinarias y punteros en la misma instrucción. Trate de adivinar lo que hace esta declaración:

```
long* pnumber {}, number {};
```

Esto define dos variables: una llamada **pnumber** de tipo "puntero a long", que se inicializa con **nullptr**, y una llamada número de tipo **long**, **no** puntero a largo!, que se inicializa con 0L. La notación que yuxtapone el asterisco y el nombre del tipo no deja claro cuál será el tipo de la segunda variable.

Queda un poco más claro si define las dos variables de esta forma:

```
long *pnumber {}, number {};
```

# PUNTEROS Y REFERENCIAS

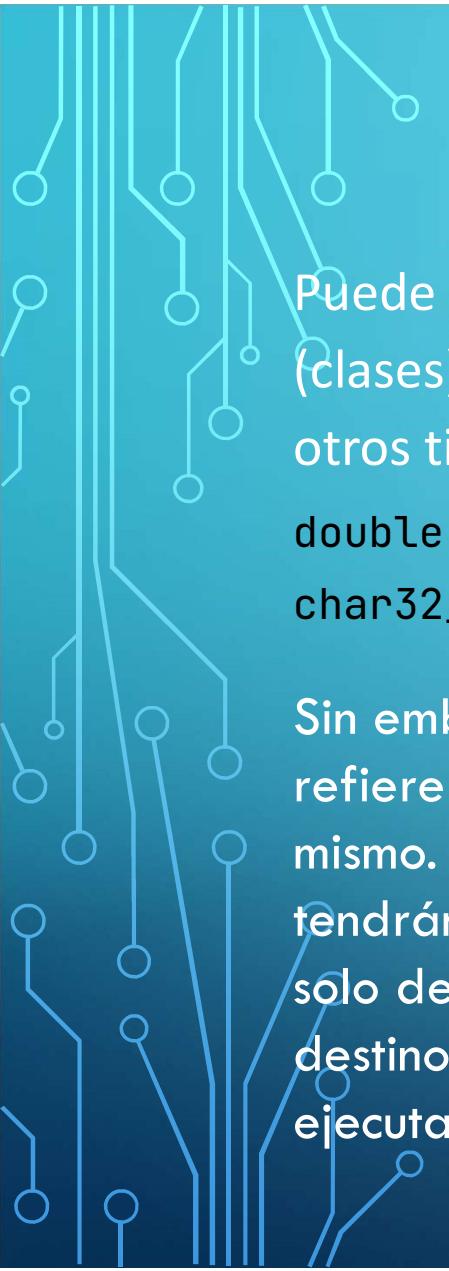
Esto es un poco menos confuso porque el asterisco ahora está más claramente asociado con la variable **pnumber**. Aun así, la única buena solución es evitar el problema en primer lugar. Es mucho mejor definir siempre punteros y variables ordinarias en sentencias separadas:

```
long number {}; // Variable de tipo long  
long* pnumber {}; // Variable de tipo 'puntero a long'
```

Ahora no hay posibilidad de confusión, y existe la ventaja adicional de que puede agregar comentarios para explicar cómo se usan las variables. Tenga en cuenta que si desea que **number** sea un segundo puntero, puede escribir lo siguiente:

```
long *pnumber {}, *number {}; // Definir dos variables de tipo  
// 'puntero a long'
```

# PUNTEROS Y REFERENCIAS



Puede definir punteros a cualquier tipo, incluidos los tipos que defina (clases). Aquí hay definiciones para variables de puntero de un par de otros tipos:

```
double *pvalue {}; // Puntero a un valor de tipo double  
char32_t *char_pointer {}; // Puntero a un carácter de 32-bit
```

Sin embargo, sin importar el tipo o el tamaño de los datos a los que se refiere un puntero, el tamaño de la variable puntero siempre será el mismo. Todas las variables puntero para una plataforma determinada tendrán el mismo tamaño. El tamaño de las variables puntero depende solo de la cantidad de memoria direccionable de su plataforma de destino. Para saber cuál es ese tamaño para su plataforma, puede ejecutar este pequeño programa:

# PUNTEROS Y REFERENCIAS

Ejecute este fragmento en su compilador y observe cual es el tamaño del puntero

```
#include <iostream>

// Tamaño de los punteros
int main()
{
    // Imprimir el tamaño (en número de bytes) de algunos tipos de datos
    // y los correspondientes tipos de puntero:
    std::cout << sizeof(double) << " > " << sizeof(char) << std::endl;
    std::cout << sizeof(double*) << " == " << sizeof(char*) << std::endl;
}
```

Para casi todas las plataformas actuales, el tamaño de las variables puntero será de 4 u 8 bytes (para arquitecturas informáticas de 32 y 64 bits, respectivamente. En principio, también puede encontrar otros valores, por ejemplo, si se dirige a sistemas embedded más especializados.

# PUNTEROS Y REFERENCIAS

## **Operador dirección (&)**

El **operador dirección (&)** es un operador unario que obtiene la dirección de memoria de su operando. Por ejemplo, teniendo en cuenta las siguientes declaraciones:

```
long number {12345L};  
long *pnumber {&number};
```

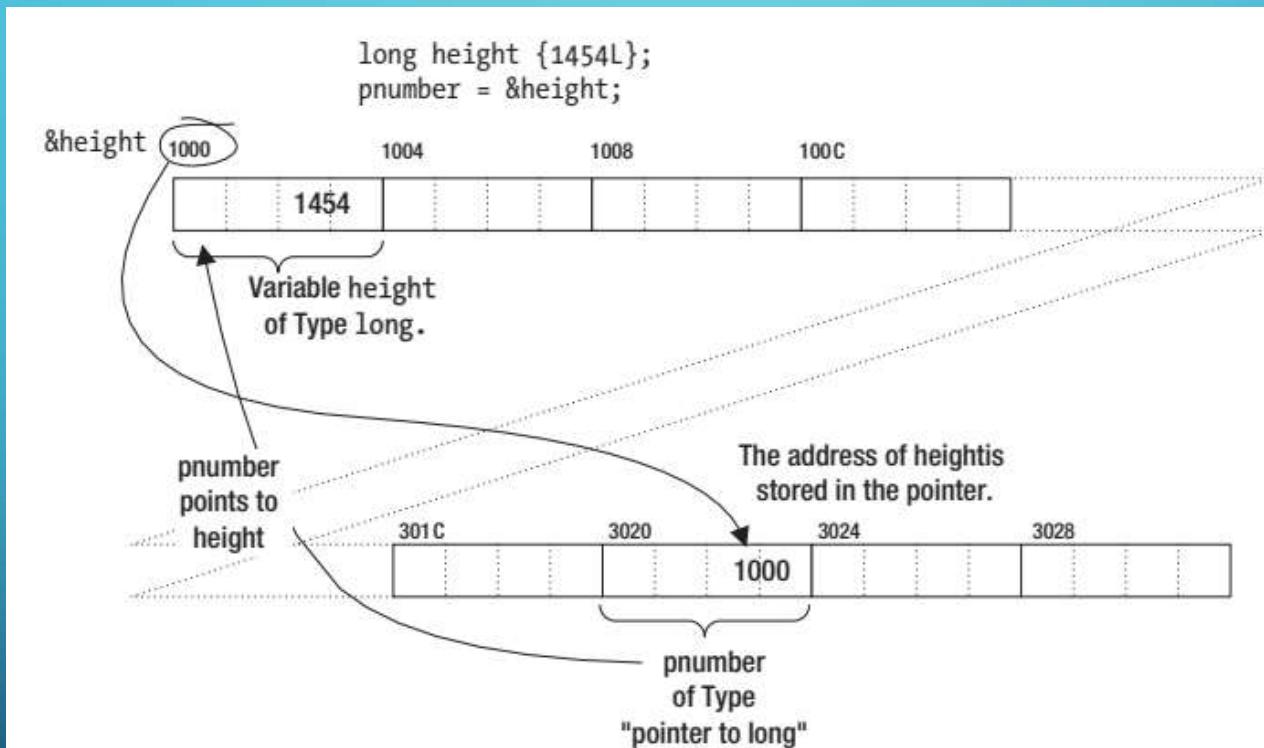
`&number` produce la dirección de `number`, por lo que `pnumber` tiene esta dirección como su valor inicial. `pnumber` puede almacenar la dirección de cualquier variable de tipo `long`, por lo que puede escribir la siguiente asignación:

```
long height {1454L}; // Stores the height of a building  
pnumber = &height; // Store the address of height in pnumber
```

El resultado de la sentencia es que `pnumber` contiene la dirección de `height`. El efecto se ilustra en la siguiente figura.

# PUNTEROS Y REFERENCIAS

## Operador dirección (&)



# PUNTEROS Y REFERENCIAS

## *Operador dirección (&)*

El operador `&` se puede aplicar a una variable de cualquier tipo, pero solo puede almacenar la dirección en un puntero del tipo apropiado. Si desea almacenar la dirección de una variable `double`, por ejemplo, el puntero debe haber sido declarado como tipo `double*`, que es "puntero a double".

Naturalmente, puede hacer que el compilador deduzca el tipo por usted también usando la palabra clave `auto`:

```
auto pmynumber {&height}; // tipo deducido: long* (puntero a long)
```

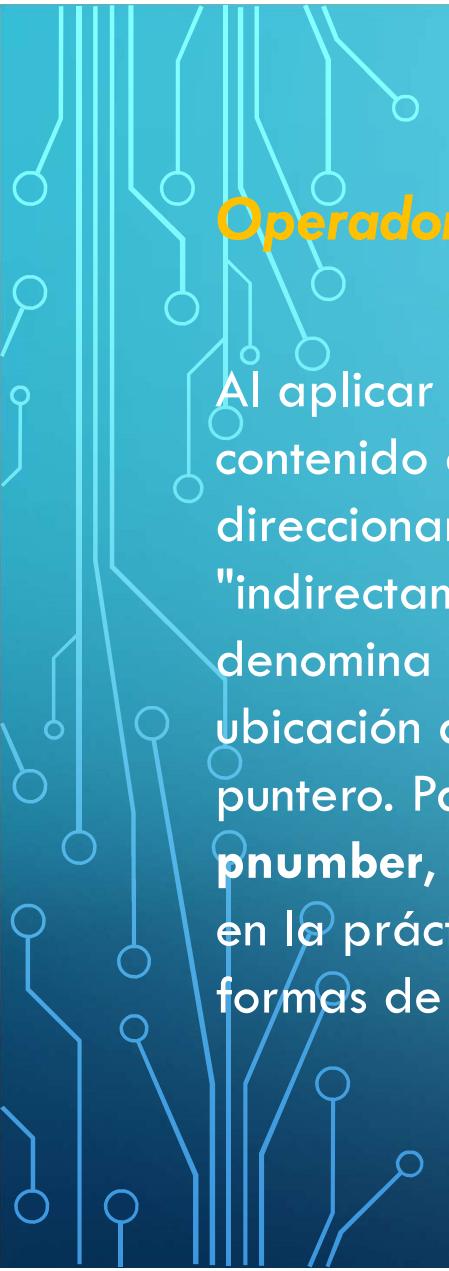
Es recomendable utilizar `auto*` aquí para dejar claro en la declaración que se trata de un puntero. Usando `auto*`, define una variable de un tipo de puntero deducido por el compilador:

```
auto* mynumber {&height};
```

# PUNTEROS Y REFERENCIAS

## *Operador dirección (&)*

Una variable declarada con `auto*` solo se puede inicializar con un valor de puntero. Inicializarlo con un valor de cualquier otro tipo dará como resultado un error de compilación. Tomar la dirección de una variable y almacenarla en un puntero está muy bien, pero lo realmente interesante es cómo usarla. Acceder a los datos en la ubicación de memoria a la que apunta el puntero es fundamental, y lo hace utilizando el *operador de indirección*



# PUNTEROS Y REFERENCIAS

## *Operador de indirección(\*)*

Al aplicar el operador de direccionamiento indirecto, `*`, a un puntero se accede al contenido de la ubicación de memoria a la que apunta. El nombre de operador de direccionamiento indirecto proviene del hecho de que se accede a los datos "indirectamente". El operador de direccionamiento indirecto a menudo también se denomina **operador de desreferencia**, y el proceso de acceder a los datos en la ubicación de memoria a la que apunta un puntero se denomina **desreferenciar** el puntero. Para acceder a los datos en la dirección contenida en el puntero `pnumber`, utilice la expresión `*pnumber`. Veamos cómo funciona la desreferenciación en la práctica con un ejemplo. El ejemplo está diseñado para mostrar varias formas de usar punteros.

# PUNTEROS Y REFERENCIAS

## *Operador de indirección(\*)*

```
#include <iostream>

// Ejemplo uso de punteros.

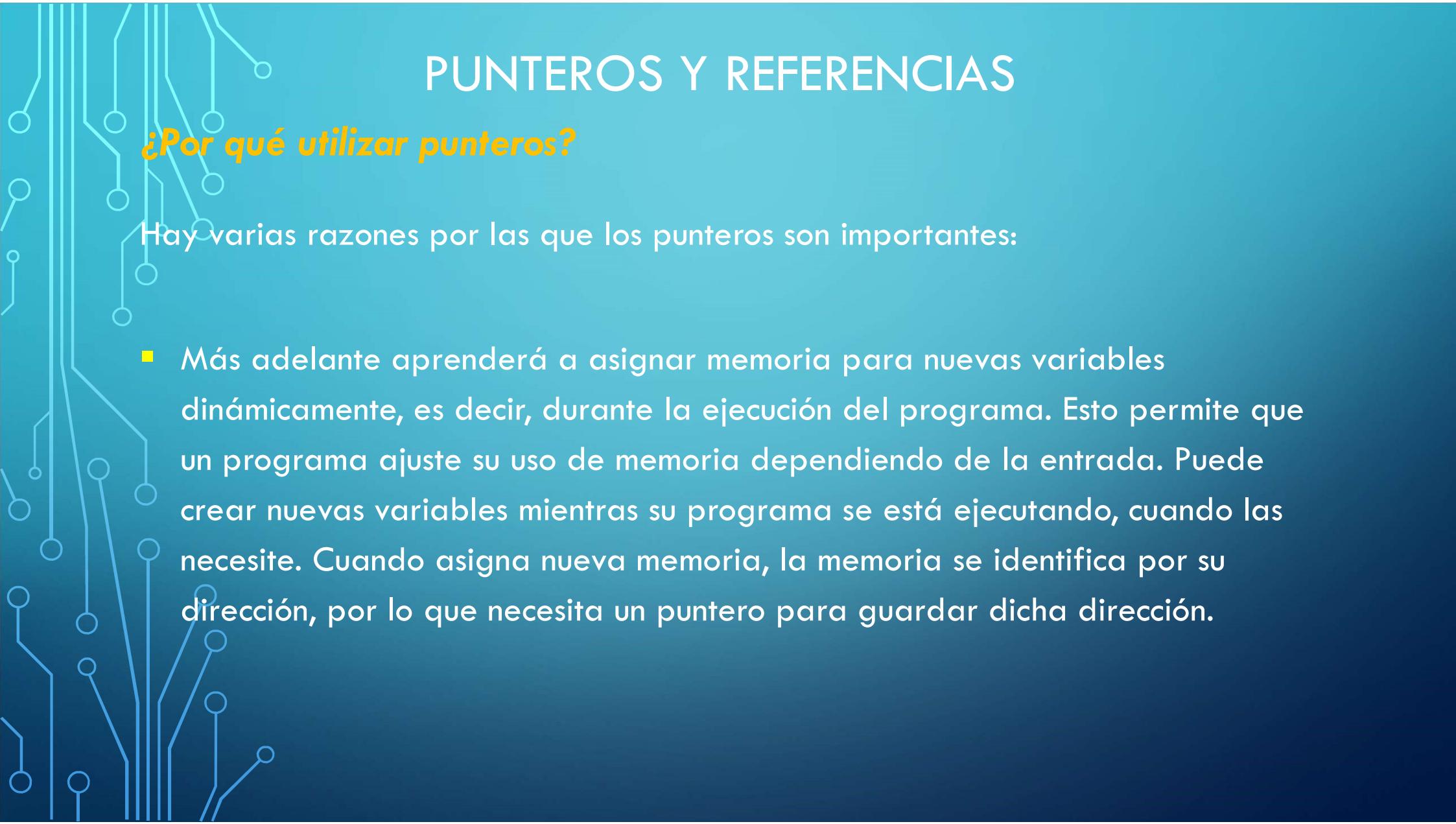
int main()
{
    unsigned int precio_unitario {450}; // Precio unitario del articulo.
    unsigned int cant_unidades {};      // Resguarda cantidad a comprar ingresada.
    unsigned int *ptr_cant_unidades {&cant_unidades};

    std::cout << "Ingrese cantidad de unidades a comprar: ";
    // Datos ingresados se almacenan en cant_unidades a través del puntero.
    std::cin >> *ptr_cant_unidades;

    unsigned int *ptr_precio_unit {&precio_unitario}; // Puntero a variable precio_unitario

    unsigned int precio_total { *ptr_precio_unit * *ptr_cant_unidades};

    std::cout << "El precio total es: " << precio_total << std::endl;
}
```

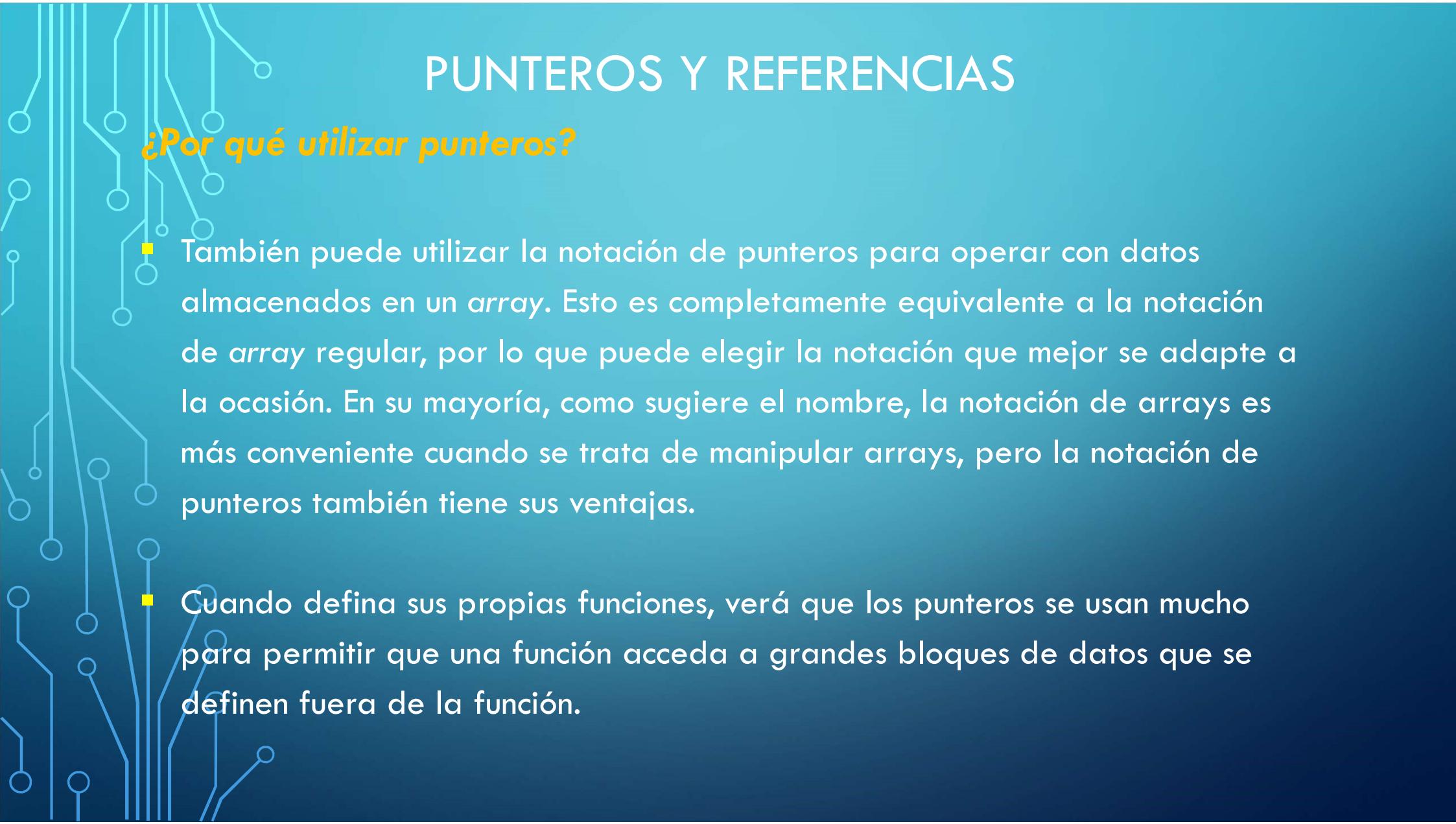


# PUNTEROS Y REFERENCIAS

## *¿Por qué utilizar punteros?*

Hay varias razones por las que los punteros son importantes:

- Más adelante aprenderá a asignar memoria para nuevas variables dinámicamente, es decir, durante la ejecución del programa. Esto permite que un programa ajuste su uso de memoria dependiendo de la entrada. Puede crear nuevas variables mientras su programa se está ejecutando, cuando las necesite. Cuando asigna nueva memoria, la memoria se identifica por su dirección, por lo que necesita un puntero para guardar dicha dirección.



# PUNTEROS Y REFERENCIAS

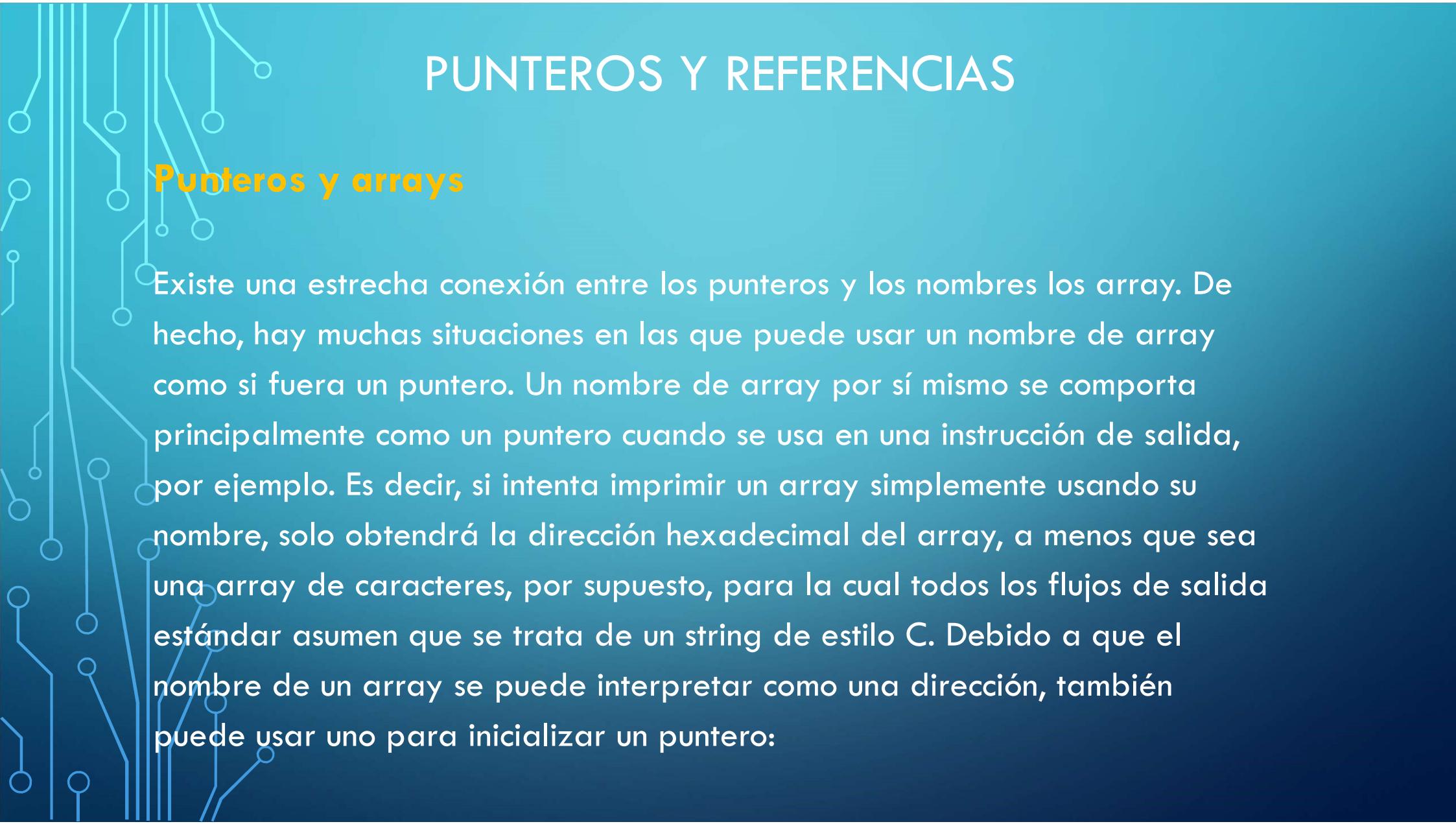
## *¿Por qué utilizar punteros?*

- También puede utilizar la notación de punteros para operar con datos almacenados en un array. Esto es completamente equivalente a la notación de array regular, por lo que puede elegir la notación que mejor se adapte a la ocasión. En su mayoría, como sugiere el nombre, la notación de arrays es más conveniente cuando se trata de manipular arrays, pero la notación de punteros también tiene sus ventajas.
- Cuando defina sus propias funciones, verá que los punteros se usan mucho para permitir que una función acceda a grandes bloques de datos que se definen fuera de la función.

# PUNTEROS Y REFERENCIAS

## *¿Por qué utilizar punteros?*

Los punteros son fundamentales para permitir que funcione el polimorfismo. El polimorfismo es quizás la capacidad más importante proporcionada por el enfoque de programación orientado a objetos.



# PUNTEROS Y REFERENCIAS

## Punteros y arrays

Existe una estrecha conexión entre los punteros y los nombres los array. De hecho, hay muchas situaciones en las que puede usar un nombre de array como si fuera un puntero. Un nombre de array por sí mismo se comporta principalmente como un puntero cuando se usa en una instrucción de salida, por ejemplo. Es decir, si intenta imprimir un array simplemente usando su nombre, solo obtendrá la dirección hexadecimal del array, a menos que sea un array de caracteres, por supuesto, para la cual todos los flujos de salida estándar asumen que se trata de un string de estilo C. Debido a que el nombre de un array se puede interpretar como una dirección, también puede usar uno para inicializar un puntero:

# PUNTEROS Y REFERENCIAS

## Punteros y arrays

```
double values[10];  
double *pvalue {values};
```

Esto almacenará la dirección del array `values` en el puntero `pvalue`. Aunque el nombre de un array representa una dirección, no es un puntero. Puede modificar la dirección almacenada en un puntero, mientras que la dirección que representa un nombre de array es fija.

*En C/C++ cada vez que pasamos un array como argumento de función, la función siempre lo trata como un puntero.*



# PUNTEROS Y REFERENCIAS

## Punteros y arrays - Aritmética de punteros

Puede realizar operaciones aritméticas en un puntero para modificar la dirección que contiene. *Está limitado a sumas y restas para modificar la dirección contenida en un puntero, pero también puede comparar punteros para producir un resultado lógico.* Puede sumar un número entero (o una expresión que se evalúe como un número entero) a un puntero y el resultado es una dirección. Puede restar un número entero de un puntero, y eso también da como resultado una dirección. Puede restar un puntero de otro y el resultado es un número entero, no una dirección. *Ninguna otra operación aritmética con punteros es legal.*

# PUNTEROS Y REFERENCIAS

## Punteros y arrays - Aritmética de punteros

Operación	Resultado	Comentario
<code>pt1++</code>	puntero	Desplazamiento ascendente de 1 elemento
<code>pt1--</code>	puntero	Desplazamiento descendente de 1 elemento
<code>pt1 + n</code>	puntero	Desplazamiento ascendente n elementos
<code>pt1 - n</code>	puntero	Desplazamiento descendente n elementos
<code>pt1 - pt2</code>	entero	Distancia entre elementos
<code>pt1 == nullptr</code>	booleano	Siempre se puede comprobar la igualdad o desigualdad con NULL
<code>pt1 != nullptr</code>	booleano	
<code>pt1 = pt2</code>	puntero	Asignación
<code>pt1 = void</code>	puntero genérico	Asignación

LA SUMA DE PUNTEROS NO ESTÁ DEFINIDA EN C++!!!

Es decir: `ptr1 + ptr2`

# PUNTEROS Y REFERENCIAS

## Aritmética de punteros

La aritmética con punteros funciona de una manera especial. Suponga que agrega 1 a un puntero con una declaración como esta:

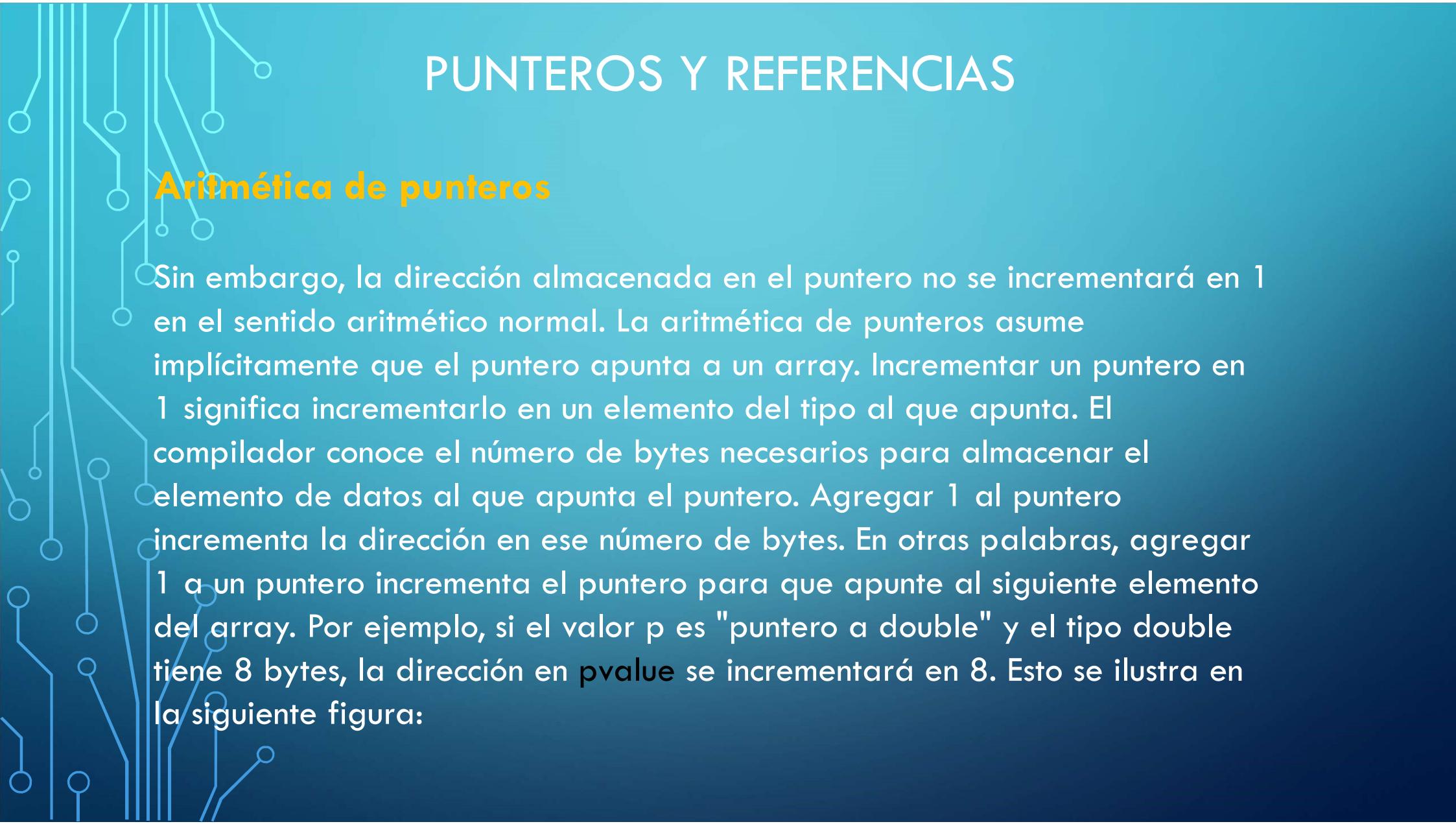
```
++pvalue;
```

Recuerde que:

```
double values[10];
double *pvalue {values};
```

También puede usar una asignación o el operador `+=` para obtener el mismo efecto:

```
pvalue += 1
```



# PUNTEROS Y REFERENCIAS

## Aritmética de punteros

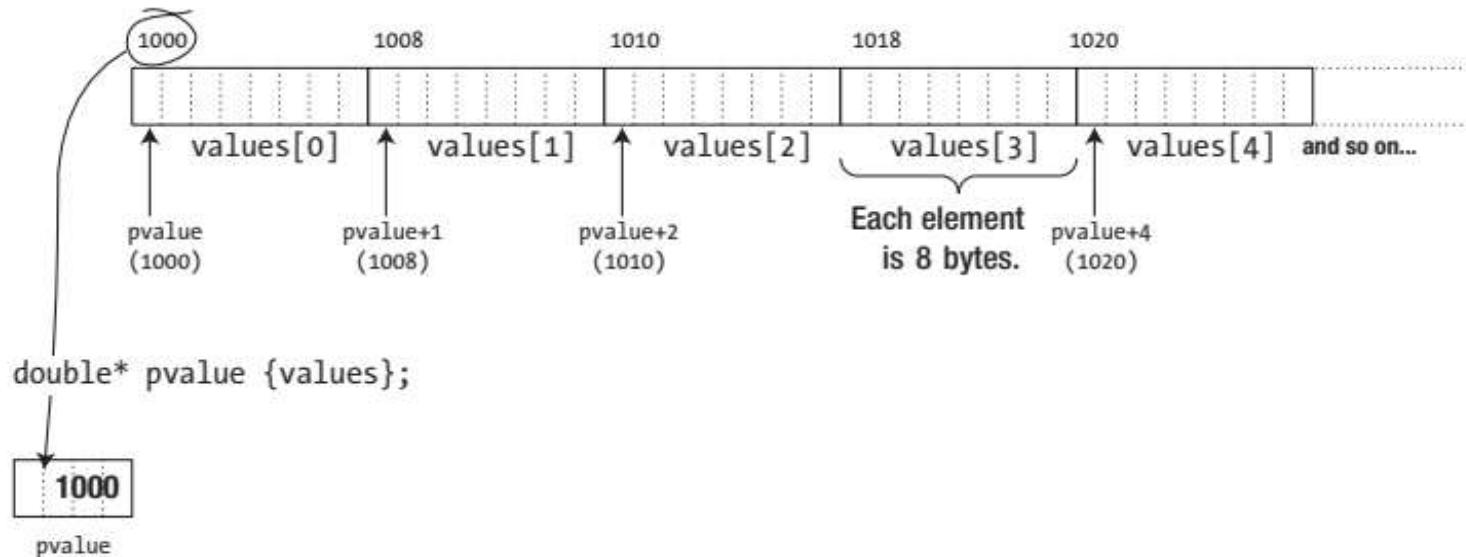
Sin embargo, la dirección almacenada en el puntero no se incrementará en 1 en el sentido aritmético normal. La aritmética de punteros asume implícitamente que el puntero apunta a un array. Incrementar un puntero en 1 significa incrementarlo en un elemento del tipo al que apunta. El compilador conoce el número de bytes necesarios para almacenar el elemento de datos al que apunta el puntero. Agregar 1 al puntero incrementa la dirección en ese número de bytes. En otras palabras, agregar 1 a un puntero incrementa el puntero para que apunte al siguiente elemento del array. Por ejemplo, si el valor `p` es "puntero a double" y el tipo `double` tiene 8 bytes, la dirección en `pvalue` se incrementará en 8. Esto se ilustra en la siguiente figura:

# PUNTEROS Y REFERENCIAS

## Aritmética de punteros

```
double values[10];
```

Array values - memory addresses are hexadecimal.



# PUNTEROS Y REFERENCIAS

## Aritmética de punteros

Como muestra la figura, `pvalue` comienza con la dirección del primer elemento del arreglo. Agregar 1 al valor incrementa la dirección que contiene en 8, por lo que el resultado es la dirección del siguiente elemento del array. De ello se deduce que al incrementar el puntero en 2 mueve el puntero dos elementos a lo largo. Por supuesto, `pvalue` no necesariamente apunta al comienzo del array `values`. Puede almacenar la dirección del tercer elemento del array en el puntero con esta declaración:

```
pvalue = &values[2];
```

Ahora, la expresión `pvalue + 1` se evaluaría como la dirección de `values[3]`, el cuarto elemento del array `values`, por lo que podría hacer que el puntero apunte a este elemento con esta declaración:

```
pvalue += 1;
```

# PUNTEROS Y REFERENCIAS

## Aritmética de punteros

En general, la expresión `pvalue + n`, en la que `n` puede ser cualquier expresión que resulte en un número entero, sumará `n * sizeof(double)` a la dirección en `pvalue` porque `pvalue` es del tipo “puntero a double”.

*Incrementar o disminuir un puntero funciona en términos del tipo de objeto al que apunta.*

Por ejemplo, incrementar un puntero a `long` por 1 cambia su contenido a la siguiente dirección `long` y así incrementa la dirección en `sizeof(long)`. Disminuyéndolo en 1, disminuye la dirección en `sizeof(long)`.

# PUNTEROS Y REFERENCIAS

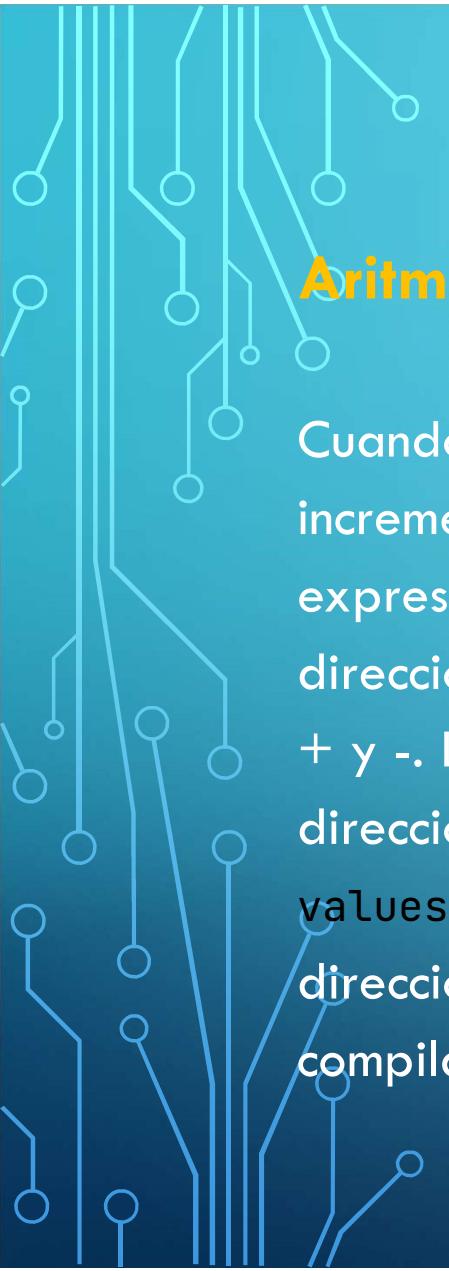
## Aritmética de punteros

Por supuesto, puede desreferenciar un puntero en el que ha realizado operaciones aritméticas. Por ejemplo, considere esta sentencia:

```
*(pvalue + 1) = *(pvalue + 2);
```

Asumiendo que **pvalue** todavía apunta a **values[3]**, esta declaración es equivalente a lo siguiente:

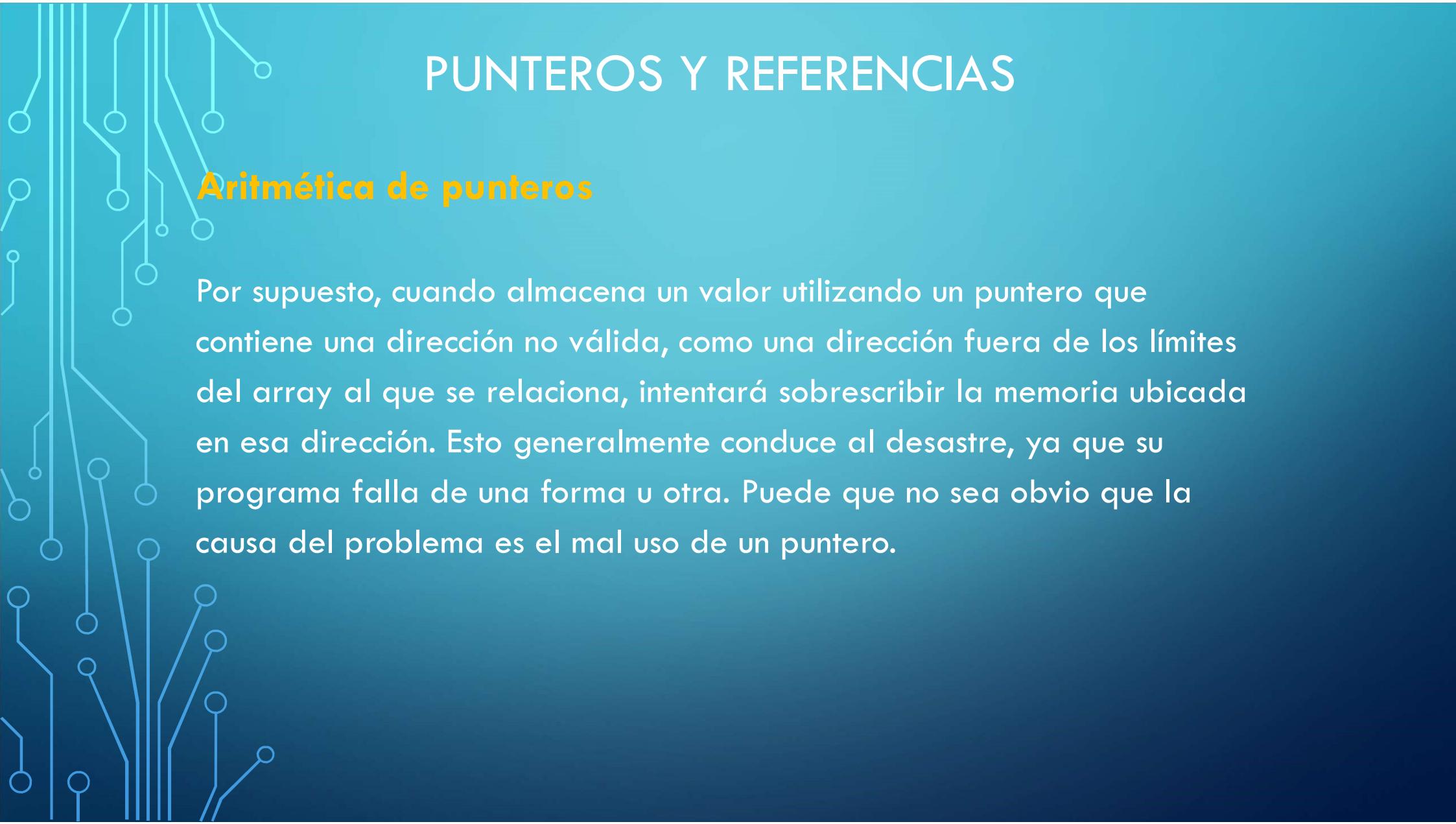
```
values[4] = values[5];
```



# PUNTEROS Y REFERENCIAS

## Aritmética de punteros

Cuando se desreferencia la dirección resultante de una expresión que incrementa o decrementa un puntero, los paréntesis alrededor de la expresión son esenciales porque la precedencia del operador de direccionamiento indirecto es mayor que la de los operadores aritméticos, + y -. La expresión `*pvalue + 1` suma 1 al valor almacenado en la dirección contenida en `pvalue`, por lo que es equivalente a ejecutar `values[3] + 1`. El resultado de `*pvalue + 1` es un valor numérico, no una dirección; su uso en la declaración de asignación anterior haría que el compilador generara un mensaje de error.



# PUNTEROS Y REFERENCIAS

## Aritmética de punteros

Por supuesto, cuando almacena un valor utilizando un puntero que contiene una dirección no válida, como una dirección fuera de los límites del array al que se relaciona, intentará sobrescribir la memoria ubicada en esa dirección. Esto generalmente conduce al desastre, ya que su programa falla de una forma u otra. Puede que no sea obvio que la causa del problema es el mal uso de un puntero.



# PUNTEROS Y REFERENCIAS

## La diferencia entre punteros

Restar un puntero de otro tiene sentido solo cuando son del mismo tipo y apuntan a elementos en el mismo array. Supongamos que tiene un array unidimensional llamado **numbers**, de tipo **long** definido de la siguiente manera:

```
long numbers[] {10, 20, 30, 40, 50, 60, 70, 80};
```

Suponga que define e inicializa dos punteros tales como estos:

```
long *pnum1 {&numbers[6]}; // Apunta al septimo elemento de  
// numbers.  
long *pnum2 {&numbers[1]}; // Apunta al segundo elemento de  
// numbers
```



# PUNTEROS Y REFERENCIAS

## La diferencia entre punteros

Puede calcular la diferencia entre estos dos punteros así:

```
auto difference {pnum1 - pnum2}; // El resultado es 5
```

El tamaño de las variables de puntero, como `pnum1` y `pnum2`, es específico de la plataforma; por lo general, es de 4 u 8 bytes. Esto, por supuesto, implica que la cantidad de bytes necesarios para almacenar los offsets de puntero tampoco puede ser la misma en todas las plataformas. Por lo tanto, el lenguaje C++ prescribe que restar dos punteros da como resultado un valor de tipo `std::ptrdiff_t`, un alias de tipo específico de la plataforma para uno de los tipos enteros con signo definidos por el encabezado `cstddef`. Así que:

```
std::ptrdiff_t difference2 {pnum2 - pnum1}; // El resultado es -5
```



# PUNTEROS Y REFERENCIAS

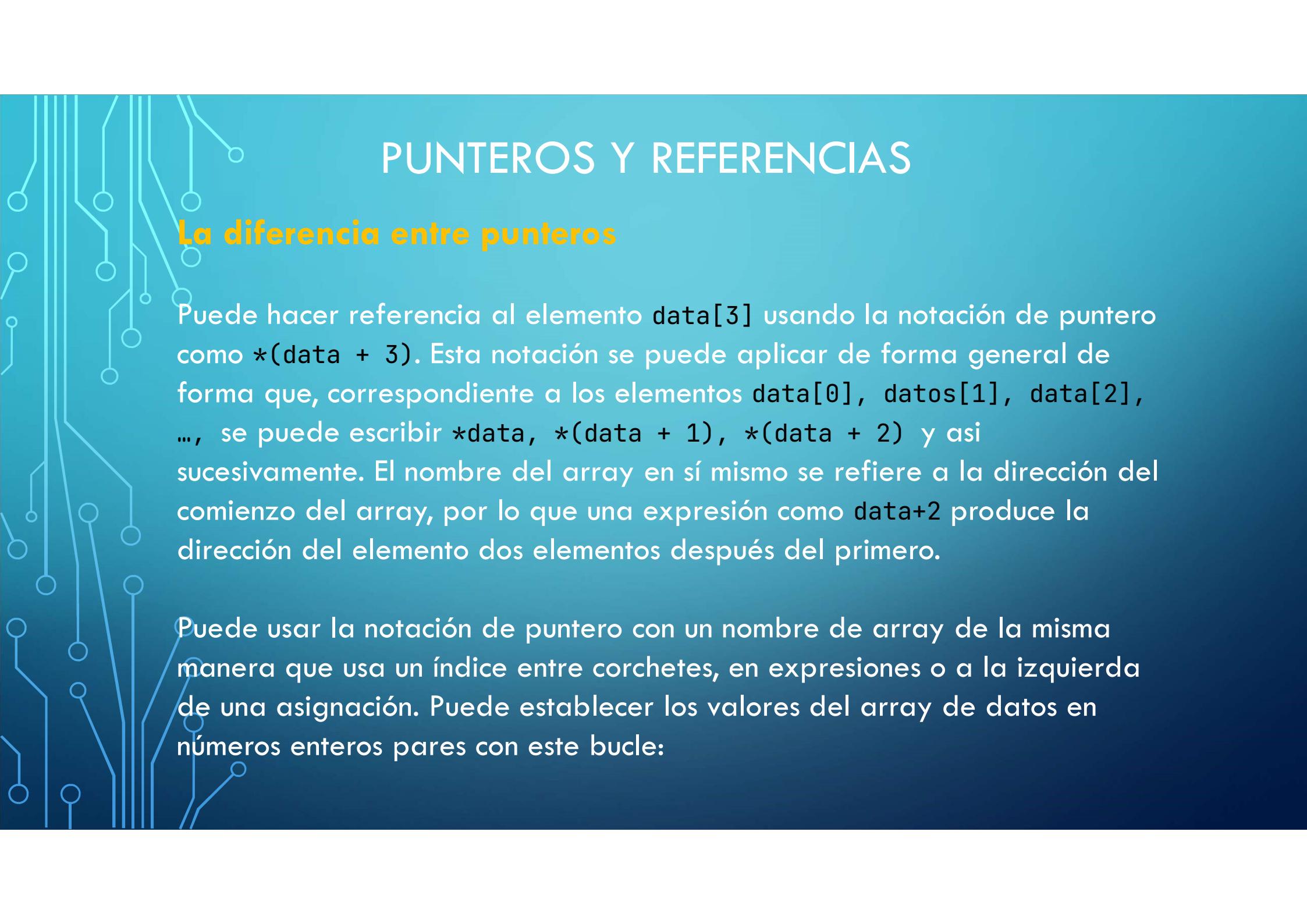
## La diferencia entre punteros

Según la plataforma de destino, `std::ptrdiff_t` suele ser un alias para `int`, `long` o `long long`.

## Uso de la notación de puntero con un nombre de array

Puede usar un nombre de array como si fuera un puntero para dirigirse a los elementos del array. Supongamos que define este array:

```
long data[5] {};
```



# PUNTEROS Y REFERENCIAS

## La diferencia entre punteros

Puede hacer referencia al elemento `data[3]` usando la notación de puntero como `*(data + 3)`. Esta notación se puede aplicar de forma general de forma que, correspondiente a los elementos `data[0]`, `datos[1]`, `data[2]`, ..., se puede escribir `*data`, `*(data + 1)`, `*(data + 2)` y así sucesivamente. El nombre del array en sí mismo se refiere a la dirección del comienzo del array, por lo que una expresión como `data+2` produce la dirección del elemento dos elementos después del primero.

Puede usar la notación de puntero con un nombre de array de la misma manera que usa un índice entre corchetes, en expresiones o a la izquierda de una asignación. Puede establecer los valores del array de datos en números enteros pares con este bucle:

# PUNTEROS Y REFERENCIAS

## La diferencia entre punteros

```
for (size_t i {}; i < std::size(data); ++i)
{
    *(data + i) = 2 * (i + 1);
```

La expresión `*(data + i)` se refiere a elementos sucesivos del array: `*(data + 0)`, que es lo mismo que `*data`, corresponde a `data[0]`, `*(data + 1)` se refiere a `data[1]`, y así. El ciclo establecerá los valores de los elementos del arreglo en 2, 4, 6, 8 y 10. Podría sumar los elementos del arreglo así:

```
long sum {};
for (size_t i {}; i < std::size(data); ++i) {
    sum += *(data + i);
```

# PUNTEROS Y REFERENCIAS

## NOTAS FINALES

Un puntero tipo `const` no permite alterar el contenido de la dirección a la que apunta, o sea no permite cambiar el valor de la variable a la que apunta. Podemos utilizarlo cuando solo deseamos leer los valores de una variable.

```
double data {61.7};  
const double* pdata {&data};  
  
*pdata = 77.2; // Ilegal, en un puntero tipo const.
```



# Ejercitación

**Ejercicio:** Realice un programa que realice las siguientes tareas:

1. Crear un array A de 10 elementos y cargarlo con los valores que usted deseé.
2. Crear un segundo array B también de 10 elementos.
3. Utilizando punteros copiar los elementos de A a B en forma inversa o sea el primer elemento de A será el último de B y el último elemento de A será el primero de B.

# Ejercitación

*Ejercicio:* Diseñe un programa que genere 100 números al azar entre 1 y 10. Mostrar un gráfico de distribución de los valores obtenidos. Realícelo con asteriscos. Realice el ejercicio utilizando punteros.

1 \*\*\*\*\*  
2 \*\*\*\*  
...  
5 \*\*\*\*\*  
...  
10 \*\*\*

# Ejercitación

**Ejercicio:** Diseñe un programa que genera una tabla con índices de masa corporal. Para ser breves, se muestra como debería verse el resultado final.  
Utilice `std::array<>`

	40	45	50	55	60	65	70	75	80	85	90	95	100	105	110	115
150 cm	17.8	20.0	22.2	24.4	26.7	28.9	31.1	33.3	35.6	37.8	40.0	42.2	44.4	46.7	48.9	51.1
155 cm	16.6	18.7	20.8	22.9	25.0	27.1	29.1	31.2	33.3	35.4	37.5	39.5	41.6	43.7	45.8	47.9
160 cm	15.6	17.6	19.5	21.5	23.4	25.4	27.3	29.3	31.2	33.2	35.2	37.1	39.1	41.0	43.0	44.9
165 cm	14.7	16.5	18.4	20.2	22.0	23.9	25.7	27.5	29.4	31.2	33.1	34.9	36.7	38.6	40.4	42.2
170 cm	13.8	15.6	17.3	19.0	20.8	22.5	24.2	26.0	27.7	29.4	31.1	32.9	34.6	36.3	38.1	39.8
175 cm	13.1	14.7	16.3	18.0	19.6	21.2	22.9	24.5	26.1	27.8	29.4	31.0	32.7	34.3	35.9	37.6
180 cm	12.3	13.9	15.4	17.0	18.5	20.1	21.6	23.1	24.7	26.2	27.8	29.3	30.9	32.4	34.0	35.5
185 cm	11.7	13.1	14.6	16.1	17.5	19.0	20.5	21.9	23.4	24.8	26.3	27.8	29.2	30.7	32.1	33.6
190 cm	11.1	12.5	13.9	15.2	16.6	18.0	19.4	20.8	22.2	23.5	24.9	26.3	27.7	29.1	30.5	31.9
195 cm	10.5	11.8	13.1	14.5	15.8	17.1	18.4	19.7	21.0	22.4	23.7	25.0	26.3	27.6	28.9	30.2
200 cm	10.0	11.2	12.5	13.8	15.0	16.2	17.5	18.8	20.0	21.2	22.5	23.8	25.0	26.2	27.5	28.8
205 cm	9.5	10.7	11.9	13.1	14.3	15.5	16.7	17.8	19.0	20.2	21.4	22.6	23.8	25.0	26.2	27.4

IMC de 18.5 a 24.9 es normal

## Ejercitación

Realice un programa que permita el ingreso de valores de temperatura. Se ingresarán datos hasta que el operador decida finalizar la entrada. Deberá presentar el mensaje: Desea continuar? (y/n). Luego para la muestra obtenida deberá calcularse la varianza y mostrarla en pantalla. La información deberá presentarse con 2 decimales.

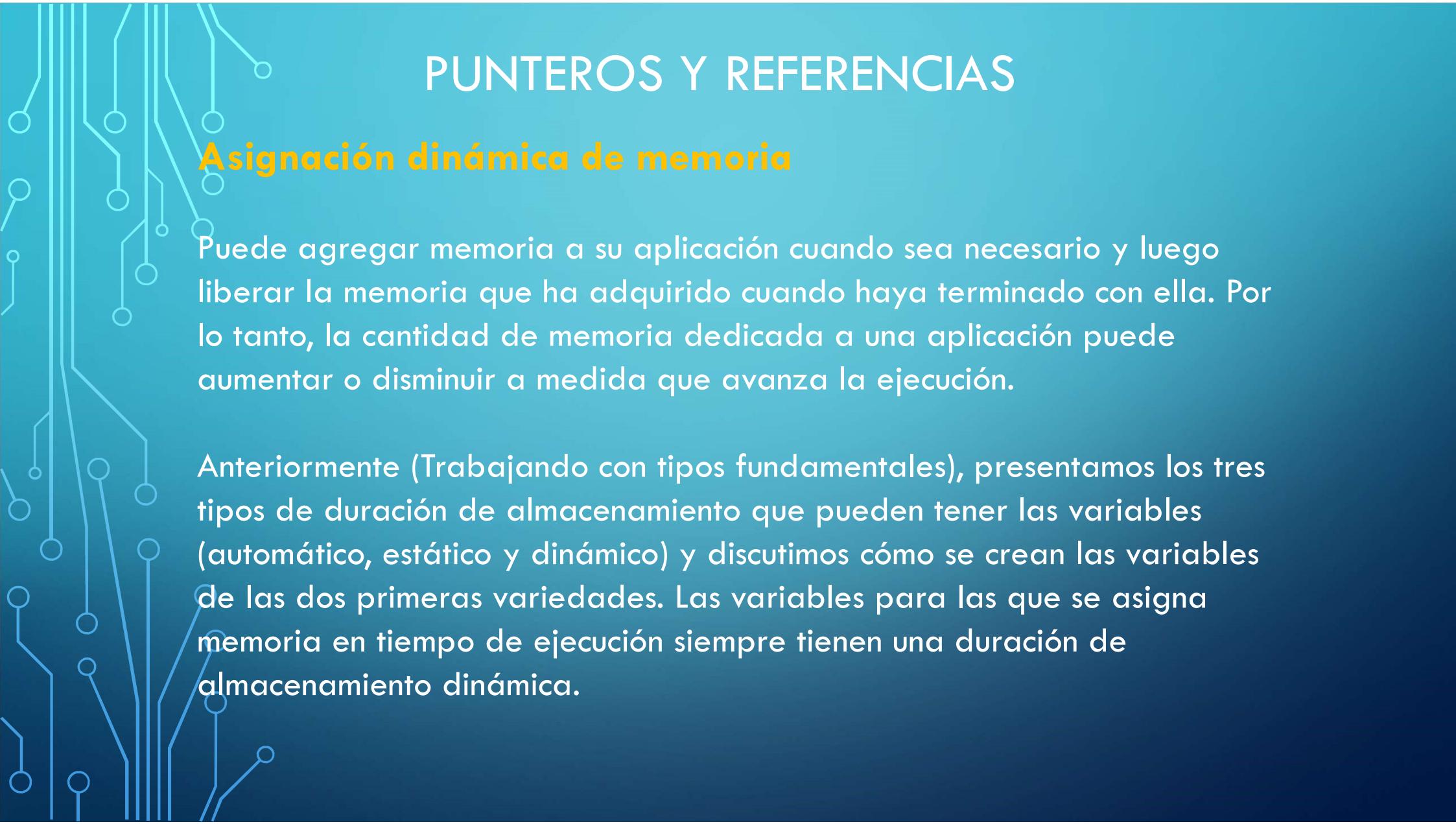
$$Var(X) = \frac{\sum_1^n (x_i - \bar{X})^2}{n}$$



# PUNTEROS Y REFERENCIAS

## Asignación dinámica de memoria

La asignación de memoria dinámica es *asignar la memoria que necesita para almacenar los datos con los que está trabajando en tiempo de ejecución*, en lugar de tener la cantidad de memoria predefinida cuando se compila el programa. Puede cambiar la cantidad de memoria que su programa le ha dedicado a medida que avanza la ejecución. Por definición, las variables asignadas dinámicamente no se pueden definir en tiempo de compilación, por lo que no se pueden nombrar en su programa fuente. Cuando asigna memoria dinámicamente, el espacio disponible se identifica por su dirección. El lugar obvio y único para almacenar esta dirección es un puntero.

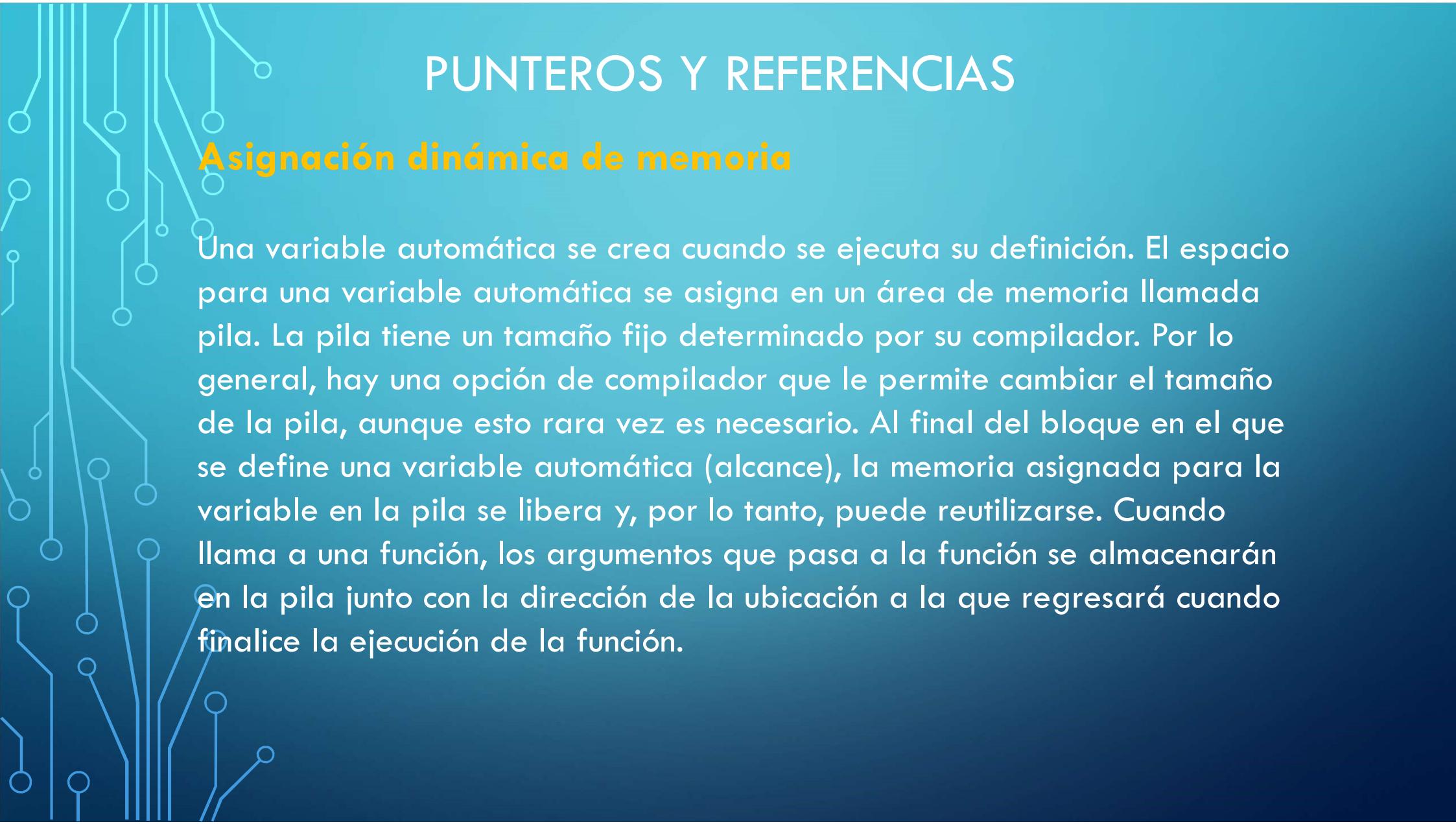


# PUNTEROS Y REFERENCIAS

## Asignación dinámica de memoria

Puede agregar memoria a su aplicación cuando sea necesario y luego liberar la memoria que ha adquirido cuando haya terminado con ella. Por lo tanto, la cantidad de memoria dedicada a una aplicación puede aumentar o disminuir a medida que avanza la ejecución.

Anteriormente (Trabajando con tipos fundamentales), presentamos los tres tipos de duración de almacenamiento que pueden tener las variables (automático, estático y dinámico) y discutimos cómo se crean las variables de las dos primeras variedades. Las variables para las que se asigna memoria en tiempo de ejecución siempre tienen una duración de almacenamiento dinámica.



# PUNTEROS Y REFERENCIAS

## Asignación dinámica de memoria

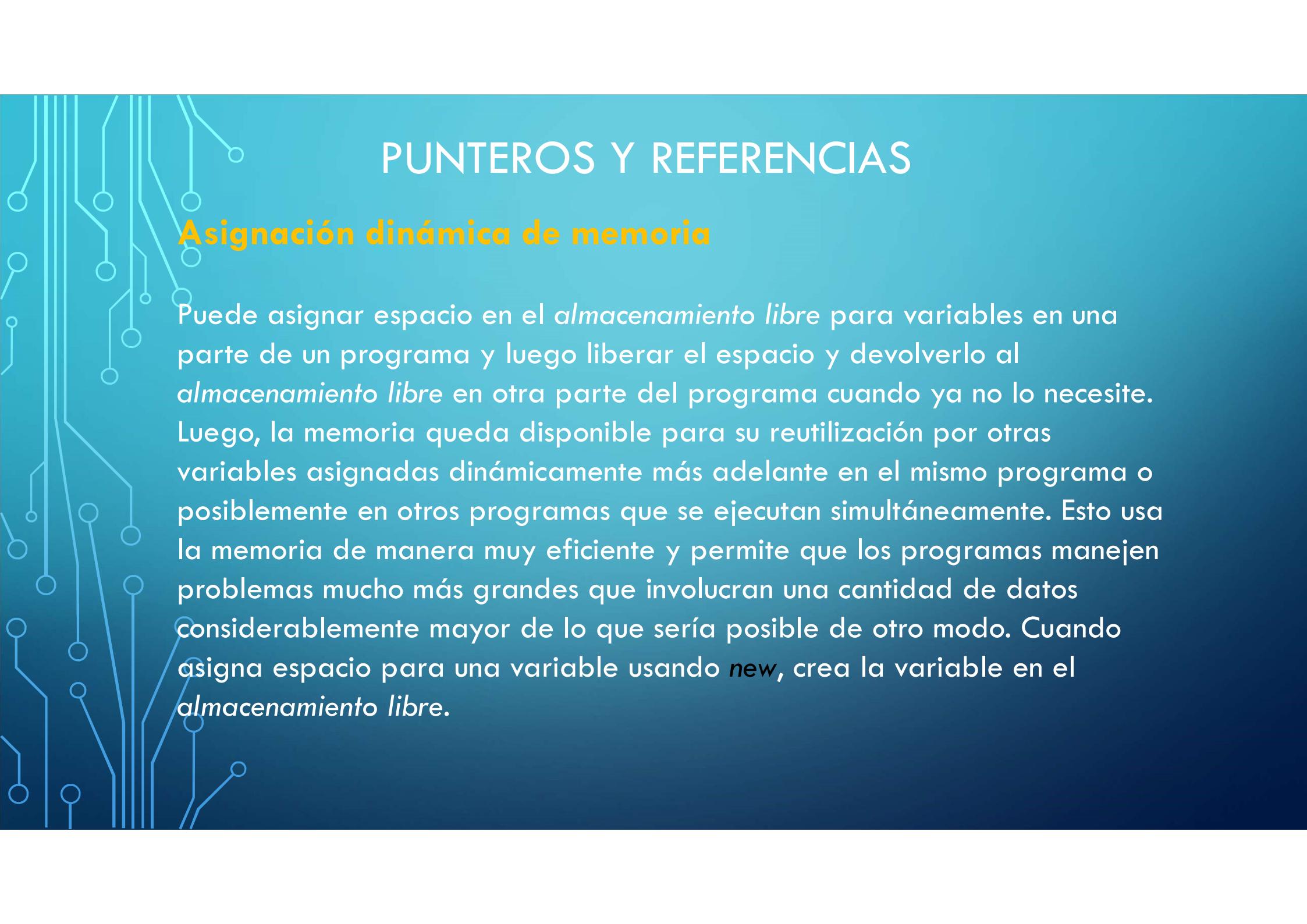
Una variable automática se crea cuando se ejecuta su definición. El espacio para una variable automática se asigna en un área de memoria llamada pila. La pila tiene un tamaño fijo determinado por su compilador. Por lo general, hay una opción de compilador que le permite cambiar el tamaño de la pila, aunque esto rara vez es necesario. Al final del bloque en el que se define una variable automática (alcance), la memoria asignada para la variable en la pila se libera y, por lo tanto, puede reutilizarse. Cuando llama a una función, los argumentos que pasa a la función se almacenarán en la pila junto con la dirección de la ubicación a la que regresará cuando finalice la ejecución de la función.



# PUNTEROS Y REFERENCIAS

## Asignación dinámica de memoria

La memoria que no está ocupada por el sistema operativo u otros programas que están actualmente cargados se denomina *almacenamiento libre*. Puede solicitar que se asigne espacio dentro del *almacenamiento libre* en tiempo de ejecución para una nueva variable de cualquier tipo. Hace esto usando el operador `new`, que devuelve la dirección del espacio asignado, y almacena la dirección en un puntero. El operador `new` se complementa con el operador `delete`, que libera memoria que previamente asignó con `new`. Tanto `new` como `delete` son palabras clave, por lo que no debe usarlas para otros fines.



# PUNTEROS Y REFERENCIAS

## Asignación dinámica de memoria

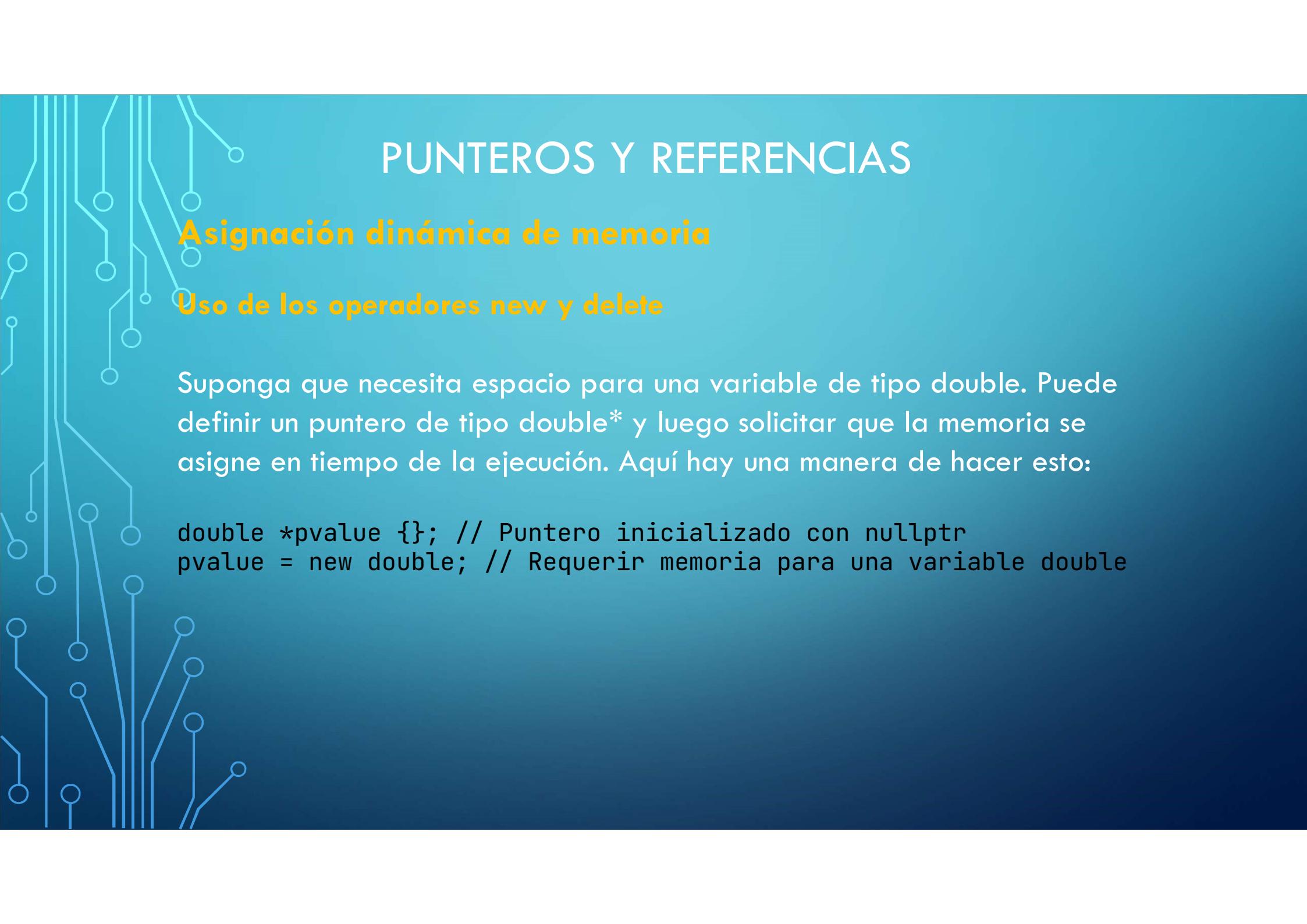
Puede asignar espacio en el *almacenamiento libre* para variables en una parte de un programa y luego liberar el espacio y devolverlo al *almacenamiento libre* en otra parte del programa cuando ya no lo necesite. Luego, la memoria queda disponible para su reutilización por otras variables asignadas dinámicamente más adelante en el mismo programa o posiblemente en otros programas que se ejecutan simultáneamente. Esto usa la memoria de manera muy eficiente y permite que los programas manejen problemas mucho más grandes que involucran una cantidad de datos considerablemente mayor de lo que sería posible de otro modo. Cuando asigna espacio para una variable usando `new`, crea la variable en el *almacenamiento libre*.



# PUNTEROS Y REFERENCIAS

## Asignación dinámica de memoria

La variable permanece reservada para usted hasta que el operador de eliminación libera la memoria que ocupa. Hasta el momento en que lo libere usando `delete`, el bloque de memoria asignado para su variable ya no podrá ser usado por llamadas posteriores de `new`. Tenga en cuenta que la memoria continúa reservada independientemente de si aún registra su dirección. Si no usa `delete` para liberar la memoria, se liberará automáticamente cuando finalice la ejecución del programa.



# PUNTEROS Y REFERENCIAS

**Asignación dinámica de memoria**

**Uso de los operadores new y delete**

Suponga que necesita espacio para una variable de tipo double. Puede definir un puntero de tipo double\* y luego solicitar que la memoria se asigne en tiempo de la ejecución. Aquí hay una manera de hacer esto:

```
double *pvalue {}; // Puntero inicializado con nullptr  
pvalue = new double; // Requerir memoria para una variable double
```



# PUNTEROS Y REFERENCIAS

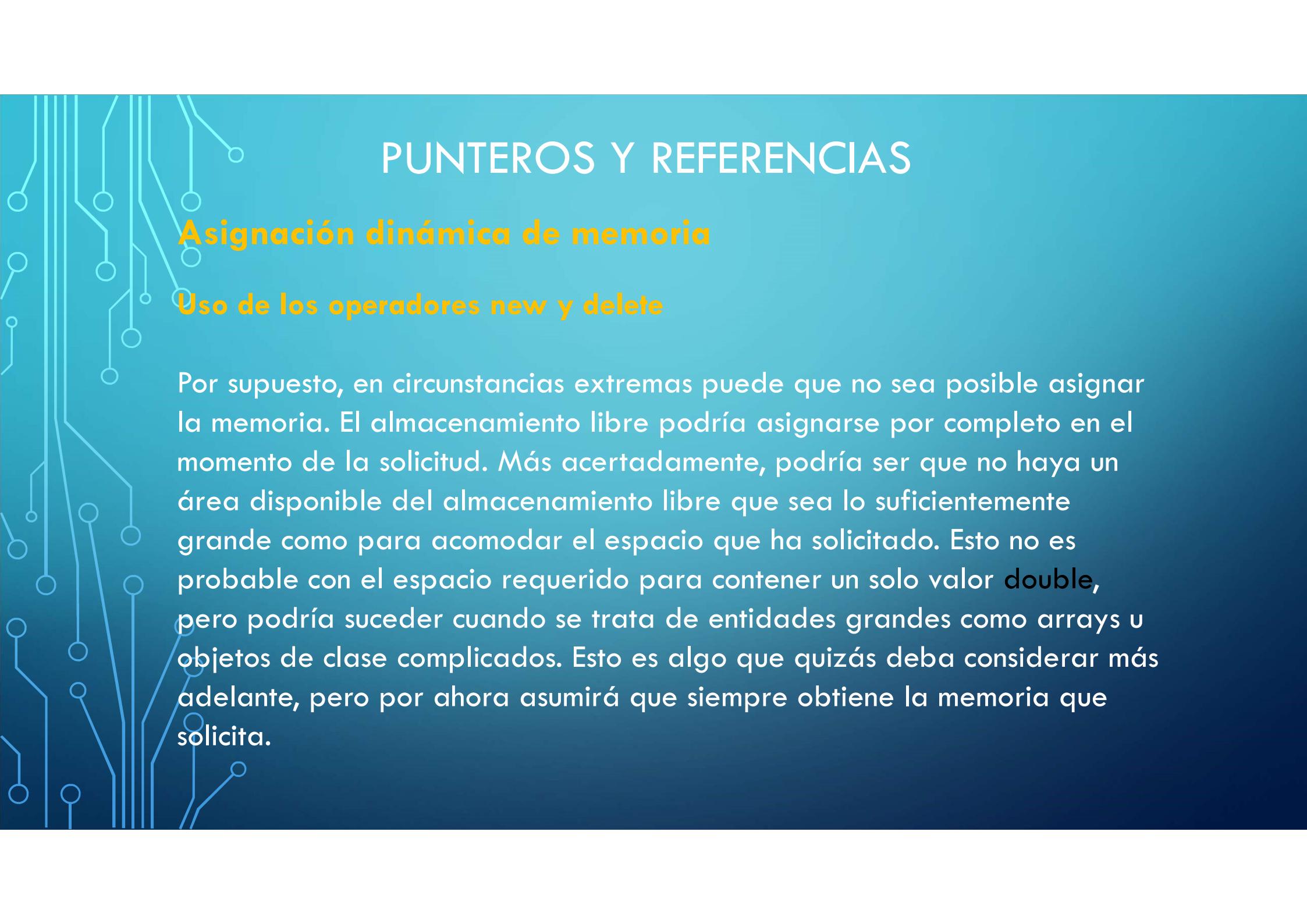
## Asignación dinámica de memoria

### Uso de los operadores new y delete

Recuerde todos los punteros deben inicializarse. El uso de la memoria de forma dinámica generalmente implica tener muchos punteros flotando alrededor, y es importante que no contengan valores falsos. Siempre debe asegurarse de que un puntero contenga nullptr si no contiene una dirección legal.

El operador `new` en la segunda línea del código devuelve la dirección de la memoria en el almacenamiento libre asignado a una variable `double`, y esto se almacena en `pvalue`. Puede usar este puntero para hacer referencia a la variable en el almacenamiento libre usando el operador indirecto, como ha visto. Aquí hay un ejemplo:

```
*pvalue = 3.14;
```



# PUNTEROS Y REFERENCIAS

## Asignación dinámica de memoria

### Uso de los operadores new y delete

Por supuesto, en circunstancias extremas puede que no sea posible asignar la memoria. El almacenamiento libre podría asignarse por completo en el momento de la solicitud. Más acertadamente, podría ser que no haya un área disponible del almacenamiento libre que sea lo suficientemente grande como para acomodar el espacio que ha solicitado. Esto no es probable con el espacio requerido para contener un solo valor double, pero podría suceder cuando se trata de entidades grandes como arrays u objetos de clase complicados. Esto es algo que quizás deba considerar más adelante, pero por ahora asumirá que siempre obtiene la memoria que solicita.



# PUNTEROS Y REFERENCIAS

**Asignación dinámica de memoria**

**Uso de los operadores new y delete**

Cuando no hay memoria disponible para completar la solicitud, el operador `new` lanza una excepción que de forma predeterminada finalizará el programa. Mas adelante profundizaremos en tema de las excepciones.

Puede inicializar una variable que cree en el almacenamiento libre.



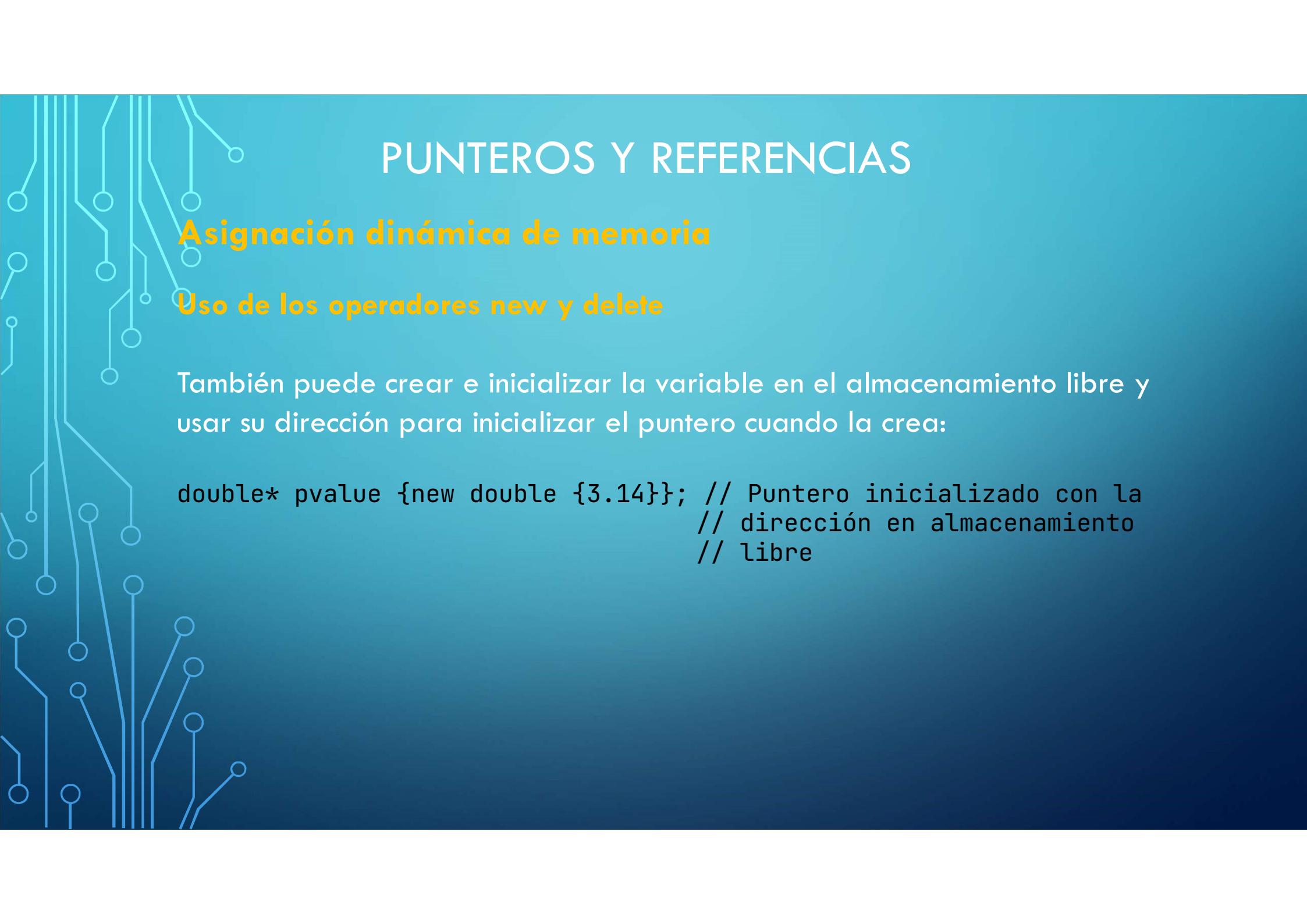
# PUNTEROS Y REFERENCIAS

## Asignación dinámica de memoria

### Uso de los operadores new y delete

Reconsideremos el ejemplo anterior: la variable double asignada por new, con su dirección almacenada en pvalue. El espacio reservado de memoria para la variable double en sí (normalmente de 8 bytes de tamaño) todavía contiene los bits que había antes. Como siempre, una variable no inicializada contiene basura. Sin embargo, podría haber inicializado su valor a, por ejemplo, 3.14, ya que se creó utilizando esta declaración:

```
pvalue = new double {3.14}; // Reservar un double e inicializarlo
```



# PUNTEROS Y REFERENCIAS

**Asignación dinámica de memoria**

**Uso de los operadores new y delete**

También puede crear e inicializar la variable en el almacenamiento libre y usar su dirección para inicializar el puntero cuando la crea:

```
double* pvalue {new double {3.14}}; // Puntero inicializado con la  
// dirección en almacenamiento  
// libre
```



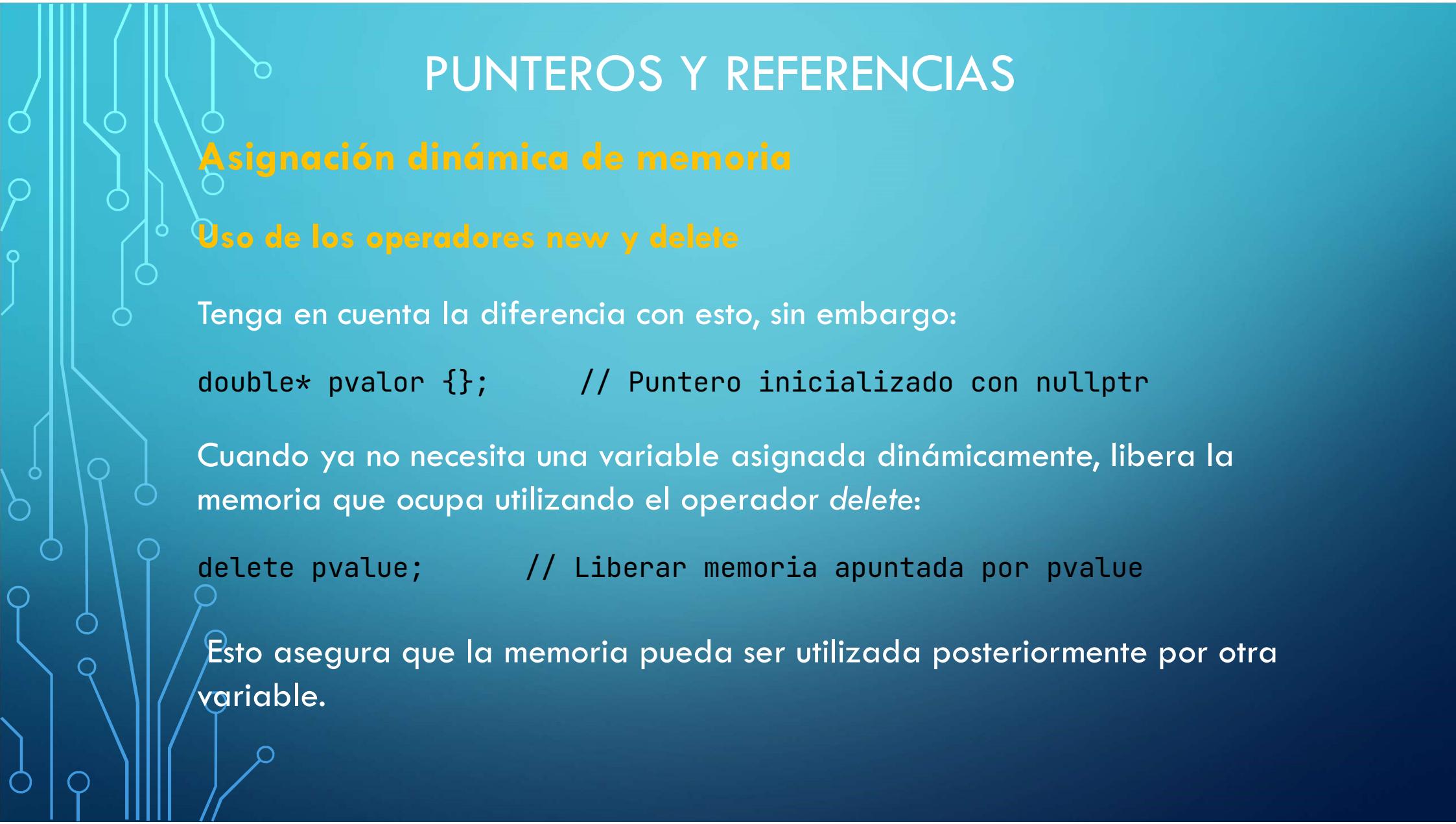
# PUNTEROS Y REFERENCIAS

## Asignación dinámica de memoria

### Uso de los operadores new y delete

Esto crea el puntero **pvalue**, asigna espacio para una variable double en el almacenamiento libre, inicializa la variable en la tienda gratuita con 3.14 e inicializa **pvalue** con la dirección de la variable. Ya no debería sorprendernos que lo siguiente inicialice la variable double a la que apunta **pvalue** a cero (0.0):

```
double* pvalue {new double {}}; // Puntero inicializado con la dirección  
// en almacenamiento libre  
// pvalue apunta a una variable double  
// inicializada con 0.0
```



# PUNTEROS Y REFERENCIAS

## Asignación dinámica de memoria

### Uso de los operadores new y delete

Tenga en cuenta la diferencia con esto, sin embargo:

```
double* pvalor {}; // Puntero inicializado con nullptr
```

Cuando ya no necesita una variable asignada dinámicamente, libera la memoria que ocupa utilizando el operador *delete*:

```
delete pvalue; // Liberar memoria apuntada por pvalue
```

Esto asegura que la memoria pueda ser utilizada posteriormente por otra variable.

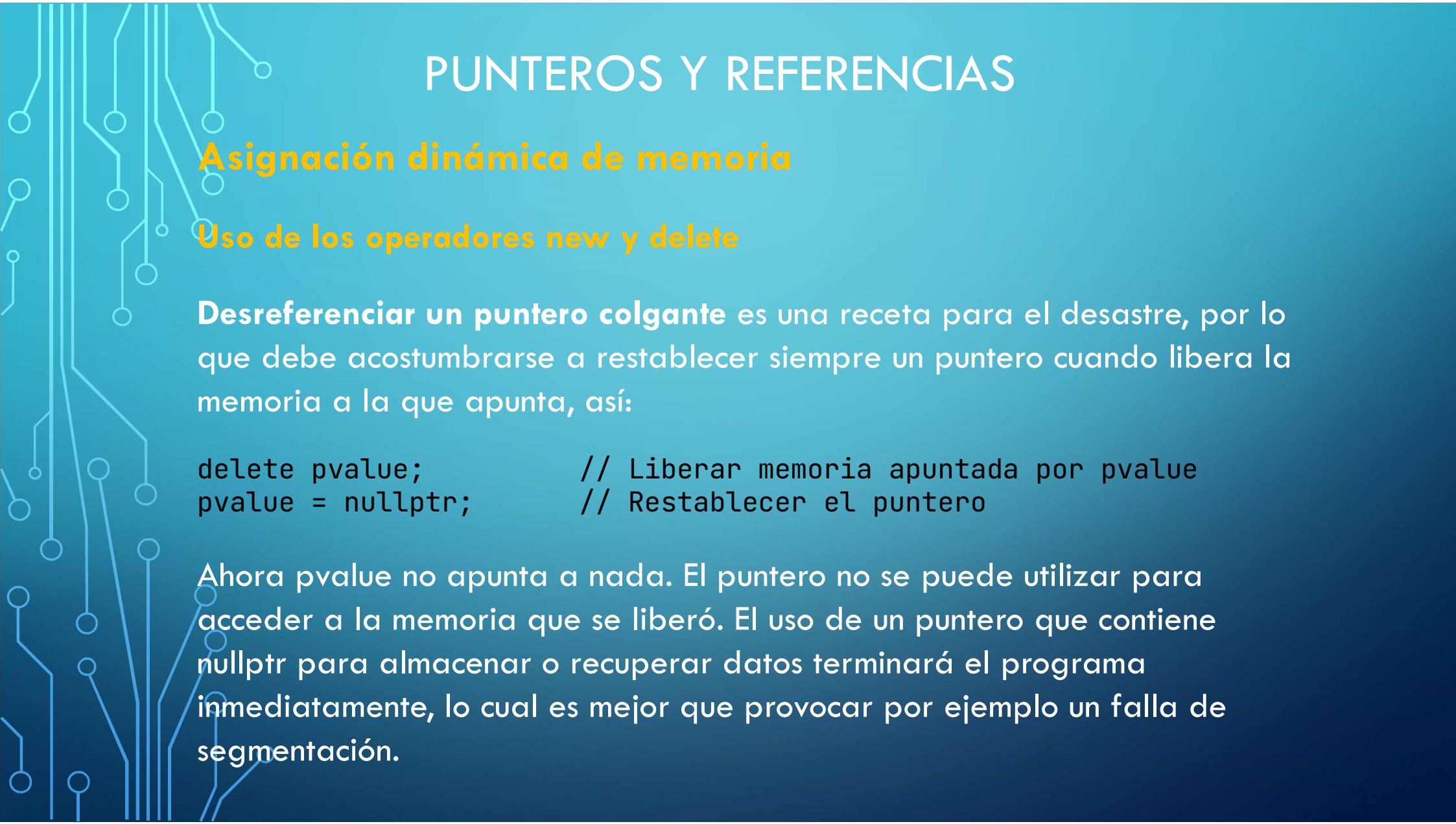


# PUNTEROS Y REFERENCIAS

## Asignación dinámica de memoria

### Uso de los operadores new y delete

Si no usa `delete` y almacena una dirección diferente en `pvalue`, será imposible liberar la memoria original porque se habrá perdido el acceso a la dirección. La memoria se conservará para que la utilice su programa hasta que finalice el programa. Por supuesto, no puede usarlo porque ya no tiene la dirección. Tenga en cuenta que el operador `delete` libera la memoria pero no cambia el puntero. Después de que se haya ejecutado la declaración anterior, `pvalue` todavía contiene la dirección de la memoria que se asignó, pero la memoria ahora está libre y se puede asignar inmediatamente a otra cosa. Un puntero que apunta a una ubicación de la memoria que se ha eliminado (o liberado) se llama *puntero colgante*.



# PUNTEROS Y REFERENCIAS

**Asignación dinámica de memoria**

**Uso de los operadores new y delete**

**Desreferenciar un puntero colgante** es una receta para el desastre, por lo que debe acostumbrarse a restablecer siempre un puntero cuando libera la memoria a la que apunta, así:

```
delete pvalue;           // Liberar memoria apuntada por pvalue  
pvalue = nullptr;        // Restablecer el puntero
```

Ahora `pvalue` no apunta a nada. El puntero no se puede utilizar para acceder a la memoria que se liberó. El uso de un puntero que contiene `nullptr` para almacenar o recuperar datos terminará el programa inmediatamente, lo cual es mejor que provocar por ejemplo un falla de segmentación.

# PUNTEROS Y REFERENCIAS

**Asignación dinámica de memoria**

**Asignación dinámica de arrays**

Asignar memoria dinámica para un array es bastante simple. Supongamos que deseamos reservar espacio para 100 elementos e tipo double:

```
double* data {new double[100]}; // Asigna 100 valores double values
```

Estos 100 valores contienen basura. Podemos inicializar la memoria asignada:

```
double* data {new double[100] {}}; // Los 100 valores a 0.0
```

```
int* one_two_three {new int[3] {1, 2, 3}}; // 3 enteros inicializados con lista
```

```
float* fdata{ new float[20] { .1f, .2f }}; // 20 elementos, los ultimos 18  
// inicializados a 0.0
```

```
auto *idata{ new int[150]{} }; // 150 elementos int inicializados a 0.0
```

# PUNTEROS Y REFERENCIAS

**Asignación dinámica de memoria**

**Asignación dinámica de arrays**

El compilador no puede deducir el tamaño del array, por tanto lo siguiente no compila:

```
int* one_two_three {new int[] {1, 2, 3}}; // No compila
```

Para liberar la memoria reservada hacemos uso del operador `delete`. Los corchetes son importantes porque indican que está eliminando un array. Al eliminar arrays de la almacenamiento libre, debe incluir los corchetes o los resultados serán impredecibles. No debe especificar tamaño en los corchetes.

```
delete[] data; // Libera el array apuntado por data
```

Recuerde la buena práctica de restablecer el puntero:

```
data = nullptr; // Reset the pointer
```

# PUNTEROS Y REFERENCIAS

## Ejercicio:

Para saber si un número es primo basta con probar si el número no es divisible por los primos hasta su raíz cuadrada. Ejemplo:

Es 149 primo?

La raiz cuadrada de 149 es 12.2... por lo que solo es necesario probar los números primos hasta 12.2..

probamos con 2,3,5,7,11

$149 : 2 = 74,5..$  (no es divisible por 2)

$149 : 3 = 49,6..$  (no es divisible por 3)

$149 : 5 = 29,8..$  (no es divisible por 5)

$149 : 7 = 21,2..$  (no es divisible por 7)

$149 : 11 = 13,5..$  (no es divisible por 11)

149 no puede ser dividido por estos primos por tanto 149 es un número primo.

# PUNTEROS Y REFERENCIAS

## Ejercicio:

O sea, para saber si un numero es primo se toma su raíz cuadrada y se verifica que dicho numero sea divisible por los valores del rango [2 a  $\sqrt{\text{numero}}$ ].

Realice un programa que solicite un valor al usuario que representará la cantidad de primos que se deberán imprimir, por ejemplo si el usuario ingresa 70, se listaran todos los primos desde 2 en adelante hasta que la cantidad de primos sea 70.  
Utilice asignación dinámica de arrays.

# PUNTEROS Y REFERENCIAS

## Selección de miembros a través de un puntero

Un puntero puede almacenar la dirección de un objeto de un tipo de clase, como un contenedor `vector<T>`. Los objetos generalmente tienen funciones miembro que operan en el objeto, hemos visto que el contenedor `vector<T>` tiene una función `at()` para acceder a los elementos y una función `push_back()` para agregar un elemento, por ejemplo. Supongamos que `pdata` es un puntero a un contenedor `vector<>`. Este contenedor podría asignarse en el almacenamiento libre con una declaración como esta:

```
auto* pdata {new std::vector<int>{}};
```

Pero también podría ser la dirección de un objeto local, obtenida usando el operador dirección&.

```
std::vector<int> datos;
auto* pdata = &datos;
```

# PUNTEROS Y REFERENCIAS

## Selección de miembros a través de un puntero

Mas alla de como se creo el vector, podemos acceder a sus métodos de la siguiente forma:

```
(*pdata).push_back(66); // Agrega un elemento de valor 66 al vector
```

El paréntesis es necesario dado que el operador punto (.) , es de mayor precedencia que el operador \*.

Esta expresión ocurriría con mucha frecuencia cuando se trabaja con objetos, razón por la cual C++ proporciona un operador que combina la desreferenciación de un puntero a un objeto y luego la selección de un miembro del objeto. Puedes escribir la declaración anterior así:

```
pdata->push_back(66); // Agrega un elemento de valor 66 al vector
```

El operador -> se denomina operador de flecha u operador de selección indirecta de miembros.

# PUNTEROS Y REFERENCIAS

## Peligros de la asignación dinámica de memoria

### Punteros colgantes y desasignaciones múltiples

Un puntero colgante, como sabe, es una variable de puntero que todavía contiene la dirección para liberar la memoria de almacenamiento que ya ha sido desasignada por `delete` o `delete[]`. Eliminar la referencia de un puntero colgante hace que lea o, a menudo peor, escriba en la memoria que ya podría estar asignada y utilizada por otras partes de su programa, lo que da como resultado todo tipo de resultados impredecibles e inesperados. Las *desasignaciones múltiples*, que ocurren cuando desasigna un puntero ya desasignado (y por lo tanto colgado) por segunda vez usando `delete` o `delete[]`, es otra receta para el desastre.

# PUNTEROS Y REFERENCIAS

## Peligros de la asignación dinámica de memoria

### Punteros colgantes y desasignaciones múltiples

En programas más complejos, sin embargo, diferentes partes del código a menudo colaboran accediendo a la misma memoria (un objeto o un array de objetos) a través de distintas copias del mismo puntero. En tales casos, nuestra simple estrategia se queda corta rápidamente. ¿Qué parte del código llamará a `delete/delete[]`? ¿Y cuando? Es decir, ¿cómo puede estar seguro de que ninguna otra parte del código sigue utilizando la misma memoria asignada dinámicamente?

# PUNTEROS Y REFERENCIAS

## Peligros de la asignación dinámica de memoria

### Desajuste de asignación/desasignación

Un array asignado dinámicamente, asignado mediante `new[]`, se captura en una variable de puntero normal. Pero también lo es un único valor asignado que se asigna usando `new`:

```
int* single_int{ new int{123} };      // Puntero a un entero simple,  
                                         // inicializado con valor 123  
int* array_of_ints{ new int[123] }; // Puntero a un array de 123  
                                         // enteros no inicializados
```

Después de esto, el compilador no tiene forma de distinguir entre los dos, especialmente una vez que dicho puntero pasa por diferentes partes del programa. Esto significa que las siguientes dos sentencias se compilarán sin errores (y en muchos casos incluso sin advertencia):

# PUNTEROS Y REFERENCIAS

## Peligros de la asignación dinámica de memoria

### Desajuste de asignación/desasignación

Un array asignado dinámicamente, asignado mediante `new[]`, se captura en una variable de puntero normal. Pero también lo es un único valor asignado que se asigna usando `new`:

```
delete[] single_int; // Mal!
delete array_of_ints; // Mal!
```

Lo que sucederá si no coincide con sus operadores de asignación y desasignación depende completamente de la implementación asociada con su compilador. Pero no será nada bueno.

# PUNTEROS Y REFERENCIAS

## Peligros de la asignación dinámica de memoria

Desajuste de asignación/desasignación

**Precaución** cada new debe estar emparejado con un solo delete; cada new[] debe estar emparejado con un solo delete[]. cualquier otra secuencia de eventos conduce a un comportamiento indefinido o pérdidas de memoria (discutido a continuación).



# PUNTEROS Y REFERENCIAS

## Peligros de la asignación dinámica de memoria

### Memory leaks – Fugas de memoria

Se produce una fuga de memoria cuando asigna memoria usando `new` o `new[]` y no la libera. Puede ocurrir que la dirección almacenada en el puntero a la zona reservada se pierda porque ha sido sobreescrita. Esto a menudo ocurre en un bucle, y es más fácil crear este tipo de problema de lo que piensa. El efecto es que su programa consume gradualmente más y más del almacenamiento libre, con el programa potencialmente ralentizándose cada vez más, o incluso fallando en el punto en que se ha asignado toda la almacenamiento libre. Cuando se trata de alcance, los punteros son como cualquier otra variable. La vida útil de un puntero se extiende desde el punto en el que lo define en un bloque hasta la llave de cierre del bloque. Después de eso, ya no existe, por lo que la dirección que contenía ya no es accesible. Si un puntero que contiene la dirección de un bloque de memoria en el almacenamiento libre queda fuera del alcance, ya no es posible eliminar la memoria.

# PUNTEROS Y REFERENCIAS

## Peligros de la asignación dinámica de memoria

### Memory leaks – Fugas de memoria

A que nos referimos cuando decimos que el problema de fugas de memoria pueden ocurrir en un bucle?

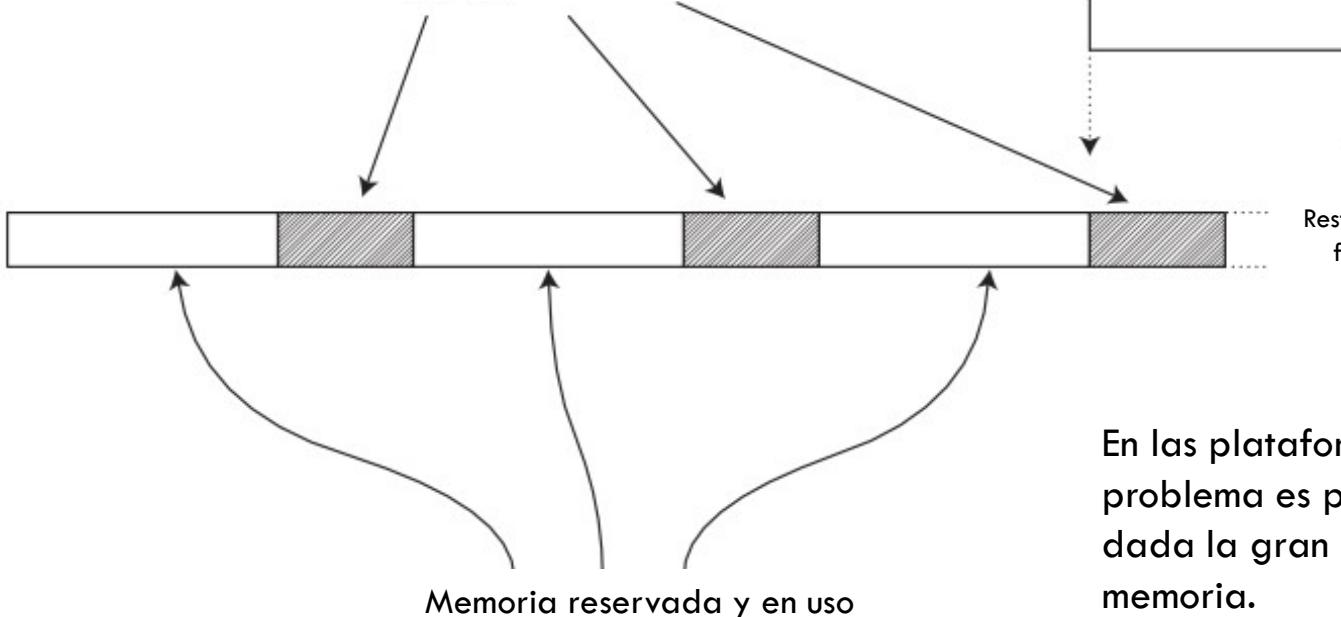
Suponga que tenemos una función que reserva memoria dinámicamente y que no la libera por una perdida de referencia del puntero al bloque reservado. Si llamamos múltiples veces a dicha función, dicha fuga ocurre una y otra vez ocasionando en algún momento la falla del programa por el consumo excesivo de la memoria libre. Uno de los síntomas que podemos apreciar es la paulatina ralentización del programa y en el caso extremo un final abrupto como consecuencia de una excepción.

# PUNTEROS Y REFERENCIAS

## Peligros de la asignación dinámica de memoria

### Fragmentación del almacenamiento libre

Bloques de memoria que se han liberado pero que son más pequeños que el siguiente bloque cuya asignación se solicitó. La cantidad total de memoria libre puede ser grande, pero todo está en pequeños bloques.



Se solicita un nuevo bloque de memoria que es mucho más pequeño que la memoria total que está libre, pero no se puede asignar porque es más grande que cualquier bloque disponible.



Resto del almacenamiento libre fragmentado de la misma manera.

En las plataformas modernas este problema es poco frecuente dada la gran disponibilidad de memoria.

# PUNTEROS Y REFERENCIAS

## REFERENCIAS

Una referencia es un nombre que puede usar como alias para otra variable. Las referencias son variables que contienen una dirección de memoria, igual que los punteros. A diferencia de los punteros, las referencias no pueden ser inicializadas a un valor arbitrario, además se **referencian y desreferencian automáticamente**. No es necesario utilizar los operadores \* o &. *Ejemplo:*

```
long var_to_be_ref {460};  
  
long& refvar {var_to_be_ref};  
  
cout << "valor var_to_be_ref: " << var_to_be_ref  
<< std::endl;  
cout << "valor refvar: " << refvar << std::endl;
```

La salida será la misma para ambas sentencias de impresión.

# PUNTEROS Y REFERENCIAS

## REFERENCIAS

Por otra lado, el siguiente fragmento provoca un error de compilación. La referencia siempre debe ser inicializada con la dirección de un variable. Recuerde que es un alias por tanto debemos proporcionarle el objeto del cual va a ser alias.

```
long& refvar {};  
  
refvar = var_to_be_ref;  
  
cout << "valor var_to_be_ref: " <<  
var_to_be_ref << std::endl;  
cout << "valor refvar: " << refvar <<  
std::endl;
```

En cuanto a las funciones, la referencia nos permite pasar un objeto por referencia pero mantener la ilusión de que lo estamos pasando por valor. La creación de las referencias esta relacionada con la ortogonalidad del lenguaje. C++ intenta ser un lenguaje altamente ortogonal.

## REFERENCIAS

# PUNTEROS Y REFERENCIAS

Además, una referencia no se puede modificar para que sea un alias de otra cosa. Una vez que una referencia se inicializa como un alias para alguna variable, sigue refiriéndose a esa misma variable durante el resto de su vida útil.

```
double data {3.5};  
double& rdata {data}; // Define una referencia a la variable data.
```

El ampersand que sigue al nombre del tipo indica que la variable que se está definiendo, rdata, es una referencia a una variable de tipo double. La variable que representa se especifica en el inicializador entre llaves. Por lo tanto, rdata es del tipo "referencia al doble". Puede utilizar la referencia como alternativa al nombre de la variable original.

# PUNTEROS Y REFERENCIAS

## REFERENCIAS

Aquí hay un ejemplo:

```
rdatos += 2.5;
```

Esto incrementa los datos en 2,5. No es necesaria ninguna de las desreferenciaciones que necesita con un puntero; solo usa el nombre de la referencia como si fuera una variable. Una referencia siempre actúa como un verdadero alias, no se puede distinguir de la variable original.

## REFERENCIAS

# PUNTEROS Y REFERENCIAS

Con una referencia, no hay necesidad de desreferenciar; simplemente esto no aplica. En cierto modo, una referencia es como un puntero al que ya se le ha quitado la referencia, aunque tampoco se puede cambiar para que haga referencia a otra cosa. Sin embargo, dada nuestra variable de referencia rdata de antes, el siguiente fragmento compila:

```
double other_data = 5.0; // Crea una segunda variable
                         // llamada other_data
rdata = other_data; // Asigna el valor actual de
                    // other_data a data (a traves de rdata)
```

## REFERENCIAS

# PUNTEROS Y REFERENCIAS

La clave es que esta última declaración no hace que rdata se refiera a la variable other\_data. La variable de referencia rdata se define como un alias para datos y siempre será un alias para datos. Una referencia es y siempre será el equivalente completo de la variable a la que se refiere. En otras palabras, la segunda declaración actúa exactamente como si escribiera esto:

```
data = other_data;
```

Una referencia tipo const no permite modificar la variable referenciada, en este caso la variable data.

```
const double& const_ref{ data };
```

# PUNTEROS Y REFERENCIAS

## REFERENCIAS

**Usar una variable tipo referencia en un for basado en rangos**

Como vimos podemos usar un bucle for basado en rango para iterar sobre todos los elementos en un array.

```
double sum {};
unsigned count {};
double temperatures[] {45.5, 50.0, 48.2, 57.0, 63.8};
for (auto t : temperatures)
{
    sum += t;
    ++count;
}
```

# PUNTEROS Y REFERENCIAS

## REFERENCIAS

### Usar una variable tipo referencia en un for basado en rangos

La variable `t` se inicializa con el valor del elemento del array actual en cada iteración, comenzando con la primera. La variable `t` no accede a ese elemento en sí. Es solo una copia local con el mismo valor que el elemento. Por lo tanto, tampoco puede usar `t` para modificar el valor de un elemento. Sin embargo, puede cambiar los elementos del array si usa una referencia:

```
const double F2C {5.0/9.0}; // Constante de conversion Fahrenheit a
                           // Celsius
for (auto& t : temperatures) // t es de tipo referencia
{
    t = (t - 32.0) * F2C;
```

# PUNTEROS Y REFERENCIAS

## REFERENCIAS

### Usar una variable tipo referencia en un for basado en rangos

La variable de bucle `t`, ahora es de tipo `double&`, por lo que es un alias para cada elemento del array. La variable de bucle se redefine en cada iteración y se inicializa con el elemento actual, por lo que la referencia nunca cambia después de inicializarse. Este ciclo cambia los valores en el array de temperaturas de Fahrenheit a Celsius.

El uso de una referencia en un bucle for basado en rango es eficiente cuando se trabaja con colecciones de objetos. La copia de objetos puede ser costosa, por lo que evitar la copia mediante el uso de un tipo de referencia hace que su código sea más eficiente. Cuando usa un tipo de referencia para la variable en un bucle for basado en rango y no necesita modificar los valores, puede usar un tipo de referencia a const para la variable de bucle:

# PUNTEROS Y REFERENCIAS

## REFERENCIAS

Usar una variable tipo referencia en un for basado en rangos

```
for (const auto& t : temperatures)
{
    std::cout << std::setw(6) << t;
    std::cout << std::endl;
}
```