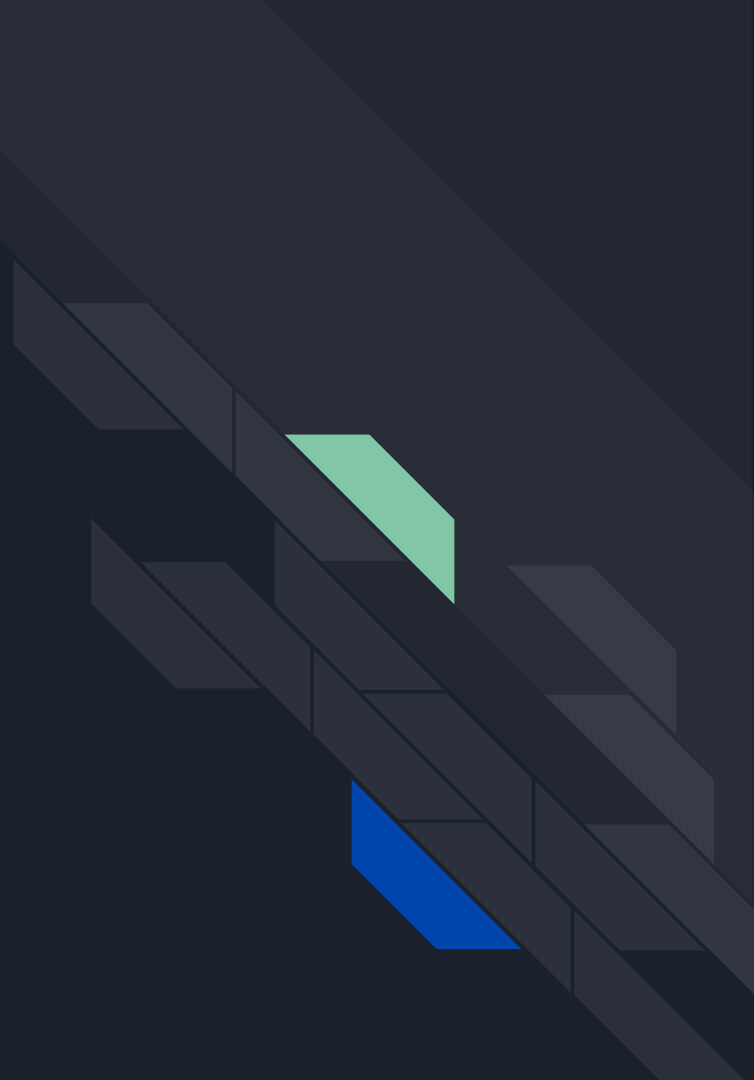


CAP 8

POINTERS



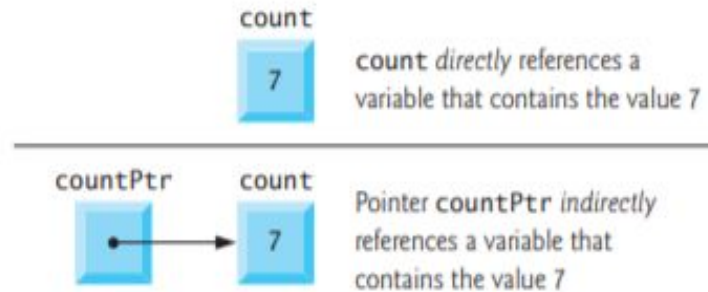


8.1 Introducción:

Este capítulo analiza los punteros, una de las capacidades más poderosas, pero desafiantes de usar, de C ++. Nuestros objetivos aquí son ayudarlo a determinar cuándo es apropiado usar punteros, y mostrar cómo utilizarlos de forma correcta y responsable. En el Capítulo 6, vimos que las referencias se pueden usar para realizar un paso por referencia. Los punteros también permiten pasar por referencia y se pueden usar para crear y manipular dinámicas estructuras de datos que pueden crecer y reducirse, como listas enlazadas, colas, pilas y árboles. Está el capítulo “explica los conceptos básicos de puntero.” El capítulo 19 presenta ejemplos de creación y utilizando estructuras de datos dinámicas basadas en punteros. También mostramos la íntima relación entre las matrices integradas y los punteros. C ++ heredó las matrices integradas del lenguaje de programación C. Como vimos en el Capítulo 7, C ++ La matriz y el vector de clases de biblioteca estándar proporcionan implementaciones más sólidas de matrices como objetos de pleno derecho. De manera similar, C ++ en realidad ofrece dos tipos de cadenas: objetos de clase de cadena (que estado utilizando desde el Capítulo 3) y cadenas basadas en punteros de estilo C (cadenas C). Este capítulo brevemente presenta cadenas C para profundizar su conocimiento de punteros y matrices integradas. Las cadenas C eran ampliamente utilizado en software C y C ++ más antiguo. Analizamos las cadenas de C en profundidad en el Apéndice F.

8.2 Pointer Variable Declarations and Initialization

¿Qué es lo que
normalmente sabemos?





Declaración de punteros:

```
type *var_name ;
```

```
int *iptr ;  
char *cptr ;  
float *fptr ;
```

Pointer initialization:

```
int *iptr = NULL;      //iptr se inicializa pero no a una direccion valida
```

```
~  
if (iptr != NULL)  
{  
:  
    //Usa el puntero si se apunta a una direccion valida  
}
```



8.3 Pointer Operators

Los operadores unarios ‘&’ y ‘*’ se utilizan para crear valores de puntero y punteros de "desreferencia", respectivamente.

Address (&) Operator:

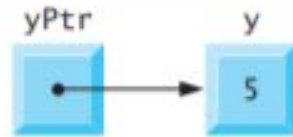
```
int y{5}; // declare variable y
int* yPtr{nullptr}; // declare pointer variable yPtr
```

Declaration:

```
yPtr = &y; // assign address of y to yPtr
```

Asigna la dirección de la variable ‘y’ a la variable puntero ‘yPtr’.

FIGURA 8.2:





8.3.2 Indirection (*) Operator

El operador unario *, comúnmente denominado operador de indirección u operador de desreferenciación, devuelve un valor l que representa el objeto al que apunta su operando puntero.

Por ejemplo (refiriéndose nuevamente a la Fig. 8.2), la declaración:

```
cout << *yPtr << endl;
```


Precedence and Associativity of the Operators Discussed So Far

Operators	Associativity	Type
:: ()	left to right <i>[See caution in Fig. 2.10 regarding grouping parentheses.]</i>	primary
() [] ++ -- static_cast<type>(operand)	left to right	postfix

Operators	Associativity	Type
++ -- + - ! & *	right to left	unary (prefix)
* / %	left to right	multiplicative
+ -	left to right	additive
<< >>	left to right	stream insertion/ extraction
< <= > >=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma



8.4 Pass-by-Reference with Pointers

En C++ existen 3 formas de pasar los argumentos a una función:

1. Pass-by-value
2. Pass-by-reference with a reference argument
3. Pass-by-reference with a pointer argument.

EJEMPLO PASS BY VALUE:

```
1 // Fig. 8.6: fig08_06.cpp
2 // Pass-by-value used to cube a variable's value.
3 #include <iostream>
4 using namespace std;
5
6 int cubeByValue(int); // prototype
7
8 int main() {
9     int number{5};
10
11     cout << "The original value of number is " << number;
12     number = cubeByValue(number); // pass number by value to cubeByValue
13     cout << "\nThe new value of number is " << number << endl;
14 }
15
16 // calculate and return cube of integer argument
17 int cubeByValue(int n) {
18     return n * n * n; // cube local variable n and return result
19 }
```

The original value of number is 5
The new value of number is 125

Ejemplo
Pass-By-Reference with
Pointers

```
1 // Fig. 8.7: fig08_07.cpp
2 // Pass-by-reference with a pointer argument used to cube a
3 // variable's value.
4 #include <iostream>
5 using namespace std;
6
7 void cubeByReference(int*); // prototype
8
9 int main() {
10     int number{5};
11 }
```

Fig. 8.7 | Pass-by-reference with a pointer argument used to cube a variable's value. (Part 1 of 2.)

```
12     cout << "The original value of number is " << number;
13     cubeByReference(&number); // pass number address to cubeByReference
14     cout << "\nThe new value of number is " << number << endl;
15 }
16
17 // calculate cube of *nPtr; modifies variable number in main
18 void cubeByReference(int* nPtr) {
19     *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
20 }
```

The original value of number is 5
The new value of number is 125

Fig. 8.7 | Pass-by-reference with a pointer argument used to cube a variable's value. (Part 2 of 2.)



Expresiones y aritmética de punteros

Inicializar puntero

```
int* vPtr{v};  
int* vPtr{&v[0]};
```

Sumar y restar enteros

Suma

```
vPtr += 1;
```

Resta

```
vPtr -= 1;
```



Expresiones y aritmética de punteros

```
int v [2] = { 1 , 2 };
```

```
int* vPtr{v};
```

```
cout << vPtr << endl;
```

```
vPtr += 1;
```

```
cout << vPtr << endl;
```

```
vPtr -= 1;
```

```
cout << vPtr << endl;
```

```
0x7bfe10
```

```
0x7bfe14
```

```
0x7bfe10
```



Expresiones y aritmética de punteros

```
int v [2] = { 1 , 2 };
```

```
int* vPtr{v};
```

```
cout << vPtr << endl;
```

```
++vPtr;
```

```
cout << vPtr << endl;
```

```
vPtr++;
```

```
cout << vPtr << endl;
```

```
0x7bfe10
```

```
0x7bfe14
```

```
0x7bfe18
```



Expresiones y aritmética de punteros

```
int v [2] = { 1 , 2 };
```

```
int* vPtr{v};
```

```
cout << vPtr << endl;
```

```
--vPtr;
```

```
cout << vPtr << endl;
```

```
vPtr--;
```

```
cout << vPtr << endl;
```

```
0x7bfe10
```

```
0x7bfe0c
```

```
0x7bfe08
```




Expresiones y aritmética de punteros

```
int v [2] = { 1 , 2 };  
int* vPtr{v};  
cout << vPtr << endl;  
int* v2Ptr = vPtr + 1;  
cout << v2Ptr << endl;  
int x = v2Ptr - vPtr;  
cout << x << endl;  
vPtr--;  
cout << vPtr << endl;
```

```
0x7bfe04
```

```
0x7bfe08
```

```
1
```



Expresiones y aritmética de punteros

```
int v [3] = { 1 , 2 , 3};  
int* vPtr; // int* vPtr{v};  
vPtr = v;  // bPtr = &b[0];  
vPtr = v;  
cout << vPtr << endl;  
int offset = *(vPtr + 2);  
cout << offset << endl;
```

```
0x7bfe08
```

```
3
```



Expresiones y aritmética de punteros

```
int v [3] = { 1 , 2 , 3};  
int* vPtr; // int* vPtr{v};  
vPtr = v;  // bPtr = &b[0];  
vPtr = v;  
cout << vPtr << endl;  
int offset = *(v + 2);  
cout << offset << endl;
```

```
0x7bfe08
```

```
3
```




Expresiones y aritmética de punteros

```
int v [3] = { 1 , 2 , 3};  
int* vPtr; // int* vPtr{v};  
vPtr = v;  // bPtr = &b[0];  
cout << vPtr << endl;  
int value = vPtr[2];  
cout << value << endl;
```

```
0x7bfe08
```

```
3
```



```
#include <iostream>

using namespace std;

int main()
{
    int b[] {10, 20, 30, 40}; // create 4-element built-in array b
    int *bPtr{b};           // set bPtr to point to built-in array b

    // output built-in array b using array subscript notation
    cout << "Array b displayed with:\n\nArray subscript notation\n";

    for (size_t i{0}; i < 4; ++i)
    {
        cout << "b[" << i << "] = " << b[i] << '\n';
    }

    // output built-in array b using array name and pointer/offset notation
    cout << "\nPointer/offset notation where "
        << "the pointer is the array name\n";

    for (size_t offset1{0}; offset1 < 4; ++offset1)
    {
        cout << "*(b + " << offset1 << ") = " << b[offset1] << '\n';
    }

    // output built-in array b using bPtr and array subscript notation
    cout << "\nPointer subscript notation\n";

    for (size_t j{0}; j < 4; ++j)
    {
        cout << "bPtr[" << j << "] = " << *(b + offset1) << '\n';
    }

    cout << "\nPointer/offset notation\n";

    // output built-in array b using bPtr and pointer/offset notation
    for (size_t offset2{0}; offset2 < 4; ++offset2)
    {
        cout << "*(bPtr + " << offset2 << ") = "
            << *(bPtr + offset2) << '\n';
    }
}
```



Array b displayed with:

Array subscript notation

b[0] = 10

b[1] = 20

b[2] = 30

b[3] = 40

Pointer/offset notation where the pointer is the array name

*(b + 0) = 10

*(b + 1) = 20

*(b + 2) = 30

*(b + 3) = 40

Pointer subscript notation

bPtr[0] = 10

bPtr[1] = 20

bPtr[2] = 30

bPtr[3] = 40

Pointer/offset notation

*(bPtr + 0) = 10

*(bPtr + 1) = 20

*(bPtr + 2) = 30

*(bPtr + 3) = 40



ARRAYS INTEGRADOS

- A diferencia de las listas y tablas, estas son estructuras de datos de tamaño fijo.
- El compilador guardará los espacios de memoria que nosotros le definamos.
- Podemos definir el tamaño del array, llenar sus espacios, o definirlos sin tamaño pero con elementos.

```
int c[12]; // c is a built-in array of 12 integers
```

```
int n[5]{50, 20, 30, 10, 40};
```

```
int n[] {50, 20, 30, 10, 40};
```



Arrays integrados en funciones.

- El nombre de un array es implícitamente la dirección del primer elemento de este, de esta manera para pasar a una función como argumento, es necesario pasar el nombre de este, lo que podemos usar para modificar los elementos o solamente no modificarlos declarando un `const`.



Función begin, end.

- Para ordenar un array con la función `sort`, debemos definir un inicio y un final del array, en el caso de que sea un objeto lo definimos con la función `begin` para el inicio y la función `end` para el final.
- De esta manera ordenamos un array integrado con `begin` y `end`,

```
sort(begin(n), end(n));
```



Limitaciones

Las limitaciones de los arrays integrados son:

- No se pueden comparar con operadores de igualdad: se debe hacer un bucle para comparar elemento por elemento.
- No se pueden asignar entre sí: el nombre de la matriz es un puntero const.
- No conocen su tamaño.
- No proporcionan límites: estos deben ser dados por el usuario.

8.6 Const en punteros.

- Al pasar punteros con const, estos no pueden ser modificados.
- Existen datos constantes y no constantes.
- Existen punteros constantes y no constantes.
- Un puntero no constante para un dato no constante se puede:
 - Los datos pueden modificarse a través del puntero y el puntero puede ser modificado para referenciar a otro dato.
- Un puntero no constante para un dato constante:
 - El puntero puede cambiar a otro dato pero el dato no se modifica.
 - En otro caso arroja un error de compilación.

```
5 void f(const int*); // prototype
```

```
6
7 int main() {
8     int y{0};
9
10    f(&y); // f will attempt an illegal modification
11 }
12
13 // constant variable cannot be modified through xPtr
14 void f(const int* xPtr) {
15     *xPtr = 100; // error: cannot modify a const object
16 }
```

GNU C++ compiler error message:

```
fig08_10.cpp: In function 'void f(const int*)':
fig08_10.cpp:17:12: error: assignment of read-only location '* xPtr'
```

Const en punteros.

- En el otro caso, donde sea un puntero constante y dato no constante.
- Puntero constante y dato constante:
- Sirve de lectura.

```
1 // Fig. 8.11: fig08_11.cpp
2 // Attempting to modify a constant pointer to nonconstant data.
3
4 int main() {
5     int x, y;
6
7     // ptr is a constant pointer to an integer that can be modified
8     // through ptr, but ptr always points to the same memory location.
9     int* const ptr{&x}; // const pointer must be initialized
10
11     *ptr = 7; // allowed: *ptr is not const
12     ptr = &y; // error: ptr is const; cannot assign to it a new address
13 }
```

Microsoft Visual C++ compiler error message:

'ptr': you cannot assign to a variable that is const

8.7 Operador SizeOf

- El operador unario de tiempo de compilación `sizeof` determina el tamaño de bytes en una matriz integrada o de cualquier otro tipo de datos.

```
6
7  size_t getSize(double*); // prototype
8
9  int main() {
10     double numbers[20]; // 20 doubles; occupies 160 bytes on our system
```

```
11
12     cout << "The number of bytes in the array is " << sizeof(numbers);
13
14     cout << "\nThe number of bytes returned by getSize is "
15         << getSize(numbers) << endl;
16 }
17
18 // return size of ptr
19 size_t getSize(double* ptr) {
20     return sizeof(ptr);
21 }
```

```
The number of bytes in the array is 160
The number of bytes returned by getSize is 4
```

- Así se obtiene la medida de un array integrado.

```
sizeof numbers / sizeof(numbers[0])
```

Análisis del tamaño de los datos fundamentales.

```
6 int main() {
7     char c; // variable of type char
8     short s; // variable of type short
9     int i; // variable of type int
10    long l; // variable of type long
11    long long ll; // variable of type long long
12    float f; // variable of type float
13    double d; // variable of type double
14    long double ld; // variable of type long double
15    int array[20]; // built-in array of int
16    int* ptr[array]; // variable of type int *
```

El operador sizeof se puede aplicar a cualquier expresión o nombre de tipo. Cuando sizeof es aplicado a un nombre de variable (que no es el nombre de una matriz incorporada) u otra expresión, el se devuelve el número de bytes utilizados para almacenar el tipo específico de expresión.

```
17
18     cout << "sizeof c = " << sizeof c
19         << "\\tsizeof(char) = " << sizeof(char)
20         << "\\nsizeof s = " << sizeof s
21         << "\\tsizeof(short) = " << sizeof(short)
22         << "\\nsizeof i = " << sizeof i
23         << "\\tsizeof(int) = " << sizeof(int)
24         << "\\nsizeof l = " << sizeof l
25         << "\\tsizeof(long) = " << sizeof(long)
26         << "\\nsizeof ll = " << sizeof ll
27         << "\\tsizeof(long long) = " << sizeof(long long)
28         << "\\nsizeof f = " << sizeof f
29         << "\\tsizeof(float) = " << sizeof(float)
30         << "\\nsizeof d = " << sizeof d
31         << "\\tsizeof(double) = " << sizeof(double)
32         << "\\nsizeof ld = " << sizeof ld
33         << "\\tsizeof(long double) = " << sizeof(long double)
34         << "\\nsizeof array = " << sizeof array
35         << "\\nsizeof ptr = " << sizeof ptr << endl;
36 }
```

sizeof c = 1	sizeof(char) = 1
sizeof s = 2	sizeof(short) = 2
sizeof i = 4	sizeof(int) = 4
sizeof l = 8	sizeof(long) = 8
sizeof ll = 8	sizeof(long long) = 8
sizeof f = 4	sizeof(float) = 4
sizeof d = 8	sizeof(double) = 8
sizeof ld = 16	sizeof(long double) = 16
sizeof array = 80	
sizeof ptr = 8	



CADENAS BASADAS EN PUNTEROS

CARACTERES && CARACTERES CONSTANTES

Cada programa se compone de un grupo de caracteres, que al agruparse el compilador llega a interpretarlos como instrucción y datos para realizar alguna tarea.

Y en un programa podemos encontrar “character constants”

Un carácter constante es un valor entero representado como char entre comillas simples(' ').

```
int main()
{
    const char letra = 'z';  //(z=122 ASCII)
    const char enter = '\n'; //(nueva linea)
}
```



STRINGS

Una cadena es una serie de caracteres tratados como uno solo.

Una cadena puede incluir letras, números, símbolos.

```
int main()
{
    const string palabra = "Hola 123 como estas #";
    cout << palabra << endl;
}
```

PUNTERO EN BASE STRING

Una cadena basada en punteros es un array integrado por caracteres donde el último carácter será '\0' donde indicará el final de la cadena en la memoria.

El tamaño de esa cadena incluirá el carácter '\0'

STRING LITERALES COMO INICIALIZADORES

```
char color[]{"blue"};  
const char* colorPtr{"blue"};
```

Un literal de cadena puede usarse como inicializador en la declaración de una matriz incorporada de chars o una variable de tipo `const char *`.

La primera declaración separa cada letra en caracteres incluyendo `'\0'` al final.

La segunda declaración crea un puntero que apunta a la letra `'b'`.

CARACTERES CONSTANTES COMO INICIALIZADORES


```
char color[]{'b', 'l', 'u', 'e', '\0'};
```

Es otra manera de colocar los caracteres.

ACCEDER A LOS CARACTERES

```
int main()  
{  
    char color[]{"blue"};  
    cout << color[0]<<endl;  
}
```

nombre variable[índice]



Leer cadenas en matrices integradas de *char* con *cin*

```
cin >> setw(20) >> word;
```

Uno puede indicar el número de caracteres que pueden ser colocados, en este caso solo admitiría 19 caracteres y dejaría el último para agregar el carácter nulo '\0'.

Lectura de líneas de texto en matrices integradas de caracteres con *cin.getline*

```
char sentence[80];  
cin.getline(sentence, 80, '\n');
```

Getline admite hasta 3 argumentos, variable donde almacenará el texto ingresado, uno puede indicar el número de caracteres que pueden ser colocados, en este caso solo admitiría 79, y el carácter que indicaría el fin del texto ingresado.