Punteros en C++

Ing José Luis MARTÍNEZ

22 de octubre de 2021

1. Punteros

El puntero (apuntador), es una herramienta que puede utilizar en sus programas para hacerlos más eficientes y flexibles. Los punteros son, una de las razones fundamentales para que el lenguaje C++ sea tan potente y tan utilizado.

Una variable puntero (o puntero, como se llama normalmente) es una variable que contiene direcciones de otras variables. Todas las variables vistas hasta este momento contienen valores de datos, por el contrario, las variables punteros contienen valores que son direcciones de memoria donde se almacenan datos. Utilizando punteros su programa puede realizar muchas tareas que no sería posible utilizando tipos de datos estándar .

2. Concepto de puntero

Cuando una variable se declara, se asocian tres atributos fundamentales con la misma: su nombre, su tipo y su dirección en memoria. Al valor, o contenido de una variable se accede por medio de su nombre. A la dirección de la variable se accede por medio del operador de dirección &. Una referencia es un alias de otra variable. Se declara utilizando el operador de referencia (&) que se añade al tipo de la referencia.

Por ejemplo obtenemos el valor y la dirección de una variable de referencia.

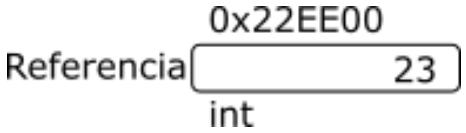


Figura 1: Dirección, tipo de dato y nombre de la variable 'referencia'

El operador & se refiere a la dirección en que se almacena el valor. El carácter & tiene diferentes usos en C++:

- Cuando se utiliza como prefijo de un nombre de una variable, devuelve la dirección de esa variable.
- Cuando se utiliza como un sufijo de un tipo en una declaración de una variable, declara la variable como sinónimo de la variable que se ha inicializado. 3. Cuando se utiliza como sufijo de un tipo en una declaración de parámetros de una función, declara el parámetro referencia de la variable que se pasa a la función.

Un puntero es una variable que contiene una dirección de una posición de memoria que puede corresponder o no a una variable declarada en el programa. La declaración de una variable puntero debe indicar el tipo de dato al que apunta; para ello se hace preceder a su nombre con un asterisco (*):

```
<tipo de dato apuntado> *<identificador de puntero>
```

C++ no inicializa los punteros cuando se declaran y es preciso inicializarlos antes de su uso. Después de la inicialización, se puede utilizar el puntero para referenciar los datos direccionados. Para asignar una dirección de memoria a un puntero se utiliza el operador &. Este método de inicialización, denominado estático, requiere:

Operador	Propósito
&	Obtiene la dirección de una variable.
*	Define una variable como puntero.
*	Obtiene el contenido de una variable puntero.

Cuadro 1: Operadores de punteros

- Asignar memoria estáticamente definiendo una variable y, a continuación, hacer que el puntero apunte al valor de la variable.
- Asignar un valor a la dirección de memoria.

Existe un segundo método para inicializar un puntero: asignación dinámica de memoria . Este método utiliza los operadores **new** y **delete**, y se analizará en el siguiente capítulo, aunque el operador **new** de reserva de memoria se usa ya en este capítulo.

El uso de un puntero para obtener el valor al que apunta, es decir, su dato apuntado, se denomina indireccionar el puntero ("desreferenciar el puntero"); para ello, se utiliza el operador de indirección *. La Tabla 2 resume los operadores de punteros.

Siempre que aparezca un asterisco (*) en una definición de variable, ésta es una variable puntero.
Siempre que aparezca un asterisco (*) delante de una variable puntero se accede a la variable contenido del puntero.
El operador & devuelve la dirección de la variable a la cual se aplica.
C++ requiere que las variables puntero direccionen realmente variables del mismo tipo de dato que está ligado a los punteros en sus declaraciones.

Un puntero puede apuntar a otra variable puntero. Este concepto se utiliza con mucha frecuencia en programas largos y complejos de C++. Para declarar un puntero a un puntero se hace preceder a la variable con dos asteriscos (**). En el siguiente código, ptr5 es un puntero a un puntero.

```
int valor_e = 100;
int *ptr1 = &valor_e;
int **ptr5 = &ptr1;
```

Ejemplo. Asignar a una variable puntero una dirección, y a su contenido un valor.

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main( int argc, char *argv[])
{
   int var; // define una variable entera var
   int *pun; //define un puntero a un entero pun
   pun = &var; //asigna la dirección de var a pun
   *pun = 60; // asigna al contenido de p 60
   cout << " &var. Direccion de var = " << &var << endl;
   cout << " pun. Contenido de pun es la misma direccion de var ";
   cout << pun << endl;
   cout <<" var. Contenido de var = " << var << endl;
   cout << " *pun. El contenido de *pun es el mismo que el de var: ";
   cout << *pun << endl;
   cin.get();
   return EXIT_SUCCESS;
}
```

Ejemplo. Escriba los números ASCII de los caracteres A, B, C, D, E, F.

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main( int argc, char *argv[])
{
   char *punteroc; // un puntero a una variable carácter
   char caracter;
   punteroc = &caracter;
   cout << " ASCII caracter" << endl;
   for (caracter = 'A'; caracter <= 'F'; caracter++)
      cout << " "<< ( int) caracter << " " << *punteroc << endl;
   cin.get();
   return EXIT_SUCCESS;
}
```

3. Punteros NULL y void

Un puntero nulo no apunta a ningún dato válido, se utiliza para proporcionar a un programa un medio de conocer cuando una variable puntero no direcciona a un dato válido. Para declarar un puntero nulo se utiliza la macro NULL.
En C++ se puede declarar un puntero de modo que apunte a cualquier tipo de dato, es decir, no se asigna a un tipo de dato específico. El método es: declarar el puntero como un puntero void. Un puntero de tipo void puede direccionar cualquier posición en memoria, pero el puntero no está unido a un tipo de dato específico.

> Un puntero nulo no direcciona ningún dato válido. Un puntero void direcciona datos de un tipo no especificado. Un puntero void se puede igualar a nulo si no se direcciona ningún dato válido.NULL es un valor; void es un tipo de dato.

Ejemplo. Los punteros void pueden apuntar a cualquier tipo de dato.

```
int x, *px = &x, &rx = x;
char* c = "Cadena larga";
float *z = NULL;
void *r = px, *s = c, *t = z;
```

x es una variable entera; px es un puntero a una variable entera inicializado a la dirección de x; rx es una referencia a un entero inicializada a x.
c (puntero a carácter) es una cadena de caracteres de longitud 10.
z es un puntero a un real inicializado a NULL.
r es un puntero void incializado a un puntero a entero; s es un puntero void, inicializado a un puntero a char; t es un puntero void inicializado a un puntero a float.

4. Punteros y arrays

Los arrays y los punteros están fuertemente relacionados en el lenguaje C++. El nombre de un array es un puntero que contiene la dirección en memoria de comienzo de la secuencia de elementos que forman el array. Este nombre del array es un puntero constante ya que no se puede modificar, sólo se puede acceder para indexar a los elementos del array. Para visualizar, almacenar o calcular un elemento de un array , se puede utilizar notación de subíndices o notación de punteros, ya que a un puntero p se le puede sumar un entero n, desplazándose el puntero tantos bytes como ocupe el tipo de dato. Si se tiene la siguiente declaración de array int V[6] = 1, 11, 21, 31, 41, 51; , su almacenamiento en memoria será el siguiente:

	V[0]	V[1]	V[2]	V[3]	V[4]	V[5]
memoria	1	11	21	31	41	51
	*V	*(V + 1)	*(V + 2)	*(V + 3)	*(V + 4)	*(V + 5)

Ejemplo. Inicialización y visualización de un array con punteros.

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main( int argc, char *argv[])
{
   float V[6];
   for ( int j = 0; j < 6; j++)
      *(V+j) = (j + 1) * 10 + 1;
   cout << " Direccion Contenido" << endl;
   for ( int j= 0; j < 6; j++)
   {
      cout << " V+" << j << " = "<< V + j;
      cout << " V[" << j <<"] = " << *(V+j)<< "\n";
   }
   cin.get();
   return EXIT_SUCCESS;
}
```

Se puede declarar un array de punteros, como un array que contiene punteros como elementos, cada uno de los cuales apuntará a otro dato específico.

Ejemplo. Inicialización y visualización de un array de punteros.

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main( int argc, char *argv[])
{
   float V[6], *P[6];
   for ( int j = 0; j < 6; j++)
   {
      *(V+j) = (5-j) * 10 + 1;
      *(P+j) = V+j; // inicialización de array de punteros
   }
   cout << " Direccion Contenido" << endl;
   for ( int j = 0; j<6; j++)
   {
      cout << " V+" << j << " = " << *(P+j) << " = *(P+" << j << ")";
      cout << " V[" << j <<"] = " << **(P+j) << "\n";
   }
   cin.get();
   return EXIT_SUCCESS;
}
```

5. Punteros de cadenas

Considérese la siguiente declaración de un array de caracteres que contiene las veintiséis letras del alfabeto internacional.

```
char alfabeto[27] = "abcdefghijklmnopqrstuvwxyz";
```

Si p es un puntero a char. Se establece que p apunta al primer carácter de alfabeto escribiendo

```
p = alfabeto; // o bien p = &alfabeto[0];
```

Es posible, entonces, considerar dos tipos de definiciones de cadena

```
char cadena1[]="Las estaciones"; //array contiene una cadena
char *pCadena = "del anyo son:"; //puntero a cadena
```

También es posible declarar un array de cadenas de caracteres:

```
char* Estaciones[4] ={"Primavera", "Verano", "Otonyo", "Invierno"};
// array de punteros a cadena
```

6. Aritmética de punteros

El manejo de la aritmética de punteros es similar a como se hace en C, veamos un repaso.

A un puntero se le puede sumar o restar un entero n; esto hace que apunte n posiciones adelante, o atrás de la actual. A una variable puntero se le puede aplicar el operador ++, o el operador −−. Esta operación hace que el operador contenga la dirección del siguiente, o anterior elemento. Se pueden sumar o restar una constante puntero a o desde un puntero y sumar o restar un entero. Sin embargo, no tiene sentido sumar o restar una constante de coma flotante.

Operaciones no válidas con punteros: no se pueden sumar dos punteros; no se pueden multiplicar dos punteros; no se pueden dividir dos punteros.

Ejemplo. El programa lee una cadena de caracteres, y mediante una variable puntero, inicializada a la primera posición del array de caracteres, se van cambiando las letras mayúsculas por minúsculas y recíprocamente. El bucle while itera hasta que se llegue al final de la cadena de caracteres. La sentencia *puntero = toupper(*puntero) convierte el contenido de de la dirección donde apunta a mayúscula, lo mismo hace la función tolower() conviertiendo a minúscula. Posteriormente, el puntero avanza una posición (un byte por ser de tipo char).

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main( int argc, char *argv[])
{
    char *puntero;
    char Cadena[81];  //
    cout << "Introduzca cadena a convertir:\n\n";
    cin.getline(Cadena, 80);
    puntero = Cadena; // puntero apunta al primer carácter de la cadena
    while (*puntero) // mientras puntero no apunte a \0
    {
        if ((*puntero >= 'A') && (*puntero <= 'Z'))
        {
            *puntero = tolower(*puntero); // función de biblioteca para convertir en minúscula
            puntero++;
        }
```

```
        else if ((*puntero >= 'a') && (*puntero <= 'z'))
        {
            *puntero = toupper(*puntero);
            puntero++; // función de biblioteca  para convertir en mayúscula
        }

        else
            puntero++;
    }
    cout << "La cadena convertida es:\n" << endl;
    cout << Cadena << endl;
    cin.get();
    return EXIT_SUCCESS;
}
```

Un puntero constante es un puntero que no se puede cambiar , pero que los datos apuntados por el puntero pueden ser cambiados. Para crear un puntero que no pueda ser modificado o puntero constante se debe utilizar el siguiente formato:

```
<tipo de dato > *const <nombre puntero> = <dirección de variable >;
```

No puede cambiarse el valor del puntero, pero puede cambiarse el contenido almacenado en la posición de memoria a donde apunta.

Un puntero a una constante se puede modificar para apuntar a una constante diferente, pero los datos apuntados por el puntero no se pueden cambiar. El formato para definir un puntero a una constante es:

```
const <tipo de dato elemento> *<nombre puntero> = <dirección de constante >;
```

Cualquier intento de cambiar el contenido almacenado en la posición de memoria a donde apunta creará un error de compilación, pero puede cambiarse el valor del puntero.

Nota: Una definición de un puntero constante tiene la palabra reservada *const* delante del nombre del puntero, mientras que el puntero a una definición constante requiere que la palabra reservada *const* se sitúe antes del tipo de dato.
Así, la definición en el primer caso se puede leer como “punteros constante o de constante”, mientras que en el segundo caso la definición se lee “puntero a tipo constante de dato ”.

El último caso a considerar es crear punteros constantes a datos constantes utilizando el formato siguiente:

```
const <tipo de dato elemento> *const <nombre puntero> = <dirección de constante >;

/*Puntero constante y puntero a constante.
Muestra las operaciones válidas y no válidas de un puntero constante puntero1, y un puntero a

int x, y;
const int z = 25; // constante entera
const int t = 50; // constante entera
int *const puntero1 = &x; //puntero1 es un puntero constante
const int *puntero2 = &z; // puntero2 es un puntero a constante
*puntero1 = y; // sentencia válida puede cambiarse su contenido
puntero1 = &y; //sentencia ilegal no puede cambiarse el puntero
puntero2 = &t; // sentencia válida puede cambiarse puntero2
*puntero2 = 15; // sentencia ilegal no puede cambiarse su contenido
```

7. Punteros en los arrays de dos dimensiones

Para apuntar a un array bidimensional como tal,o lo que es lo mismo, para apuntar a su inicio, el compilador de C++ considera que un array bidimensional es en realidad un array de punteros a los arrays que forman sus filas. Por tanto, será necesario un puntero doble o puntero a puntero, que contendrá la dirección del primer puntero del array de punteros a cada una de las filas del array bidimensional o matriz. Si *a* se ha definido como un array bidimensional, el nombre del array a es un puntero constante que apunta a la primera fila a[0] . El puntero a+1 apunta a la segunda fila a[1] , etc. A su vez a[0] es un puntero que apunta al primer elemento de la fila 0 que es a[0][0] . El puntero a[1] es un puntero que apunta al primer elemento de la fila 1 que es a[1][0] , etc.

Ejemplo Array bidimensional, punteros y posiciones de memoria. Dada la declaración float A[5][3] que define un array bidimensional de cinco filas y tres columnas, se tiene la siguiente estructura:

Puntero a puntero fila		**Puntero a fila**		**Array bidimensional** float A[4][3]		
A	→	A[0]	→	A[0][0]	A[0][1]	A[0][2]
A+1	→	A[1]	→	A[1][0]	A[1][1]	A[1][2]
A+2	→	A[2]	→	A[2][0]	A[2][1]	A[2][2]
A+3	→	A[3]	→	A[3][0]	A[3][1]	A[3][2]
A+4	→	A[4]	→	A[4][0]	A[4][1]	A[4][2]

A es un puntero que apunta a un array de 5 punteros A[0], A[1], A[2], A[3], A[4] .
A[0] es un puntero que apunta a un array de tres elementos A[0][0], A[0][1], A[0][2] .
A[1] es un puntero que apunta a un array de tres elementos A[1][0], A[1][1], A[1][2] .
A[2] es un puntero que apunta a un array de tres elementos A[2][0], A[2][1], A[2][2] .
A[3] es un puntero que apunta a un array de tres elementos A[3][0], A[3][1], A[3][2] .
A[4] es un puntero que apunta a un array de tres elementos A[4][0], A[4][1], A[4][2] .
A[i][j] es equivalente a las siguientes expresiones:

- $*(A[i]+j)$ el contenido del puntero a la fila i más el número de columna j.
- $*((*(A+i))+j)$. Si se cambia $A[i]$ por $*(A+i)$ se tiene la siguiente expresión anterior.
- $*(\&A[0][0]+3*i+j)$.

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

int main(int argc, char* argv[])
{
    int matrizA[3][3];
    for(int i=0; i<3; i++)
    {
        for(int j=0; j<3; j++)
        {
            cout << "Ingrese la fila " << i << " y la columna " << j << endl;
            cin >> matrizA[i][j];
        }
    }

    for (int i=0; i<3; i++)
    {
        cout << "Esta en la fila " << i << endl;
        for(int j=0; j<3 ; j++)
        {
            cout << " \t columna " << j << endl;
```

```
        }
    }

    cout << "\t Prueba con *(matrizA[i]+j)"<< endl;
    for (int i=0; i<3; i++)
    {
        for(int j=0; j<3; j++)
        {
            cout << *(matrizA[i]+j) << "\t";
        }
        cout << endl;
    }

    cout << "\t Prueba con *(*(matrizA + i) + j)"<< endl;
    for(int i=0; i<3;i++)
    {
        for(int j=0; j<3; j++)
        {
            cout << *(*(matrizA +i)+j) << "\t";

        }
        cout << endl;
    }

    cout << "\t Prueba con *(&A[0][0]+ 3*i+j)" << endl;
    for(int i=0; i<3 ; i++)
    {
        for(int j=0; j<3;j++)
        {
            cout << *(&matrizA[0][0]+ 3*i+j) << "\t";
        }
        cout << endl;
    }

    return EXIT_SUCCESS;
}
```

A es un puntero que apunta a A[0].
A[0] es un puntero que apunta a A[0][0].
Si A[0][0] se encuentra en la dirección de memoria 100 y teniendo en cuenta que un *float* ocupa 4 bytes, la siguiente tabla muestra un esquema de la memoria:

Contenido de puntero a puntero fila	**Contenido de puntero a fila**	**Direcciones del array bidimensional float A[4][4]**		
*A = A[0]	A[0] = 100	A[0][0] = 100	A[0][1] = 104	A[0][2] = 108
*(A+1) = A[1]	A[1] = 112	A[1][0] = 112	A[1][1] = 116	A[1][2] = 120
*(A+2) = A[2]	A[2] = 124	A[2][0] = 124	A[2][1] = 128	A[2][2] = 132
*(A+3) = A[3]	A[3] = 136	A[3][0] = 136	A[3][1] = 140	A[1][2] = 144
*(A+4) = A[4]	A[4] = 148	A[4][0] = 148	A[4][1] = 152	A[4][2] = 156

Ejemplo. Matriz de reales y vector de punteros a reales. Diferencias que pueden encontrarse entre las declaraciones: float A[10][10]; float *V[10];. ¿Pueden realizarse las siguientes asignaciones?: A = V; V[1] = A[1];

V[2] = &mt[2][0];.

A es una matriz de reales y V es un vector de punteros a reales, por lo que las declaraciones son esencialmente diferentes.
En cuanto a la primea asignación A = V, ambas variables A y V son punteros que apuntan a punteros a reales (aunque ligeramente diferente) por lo que en principio podría realizarse la asignación, pero debido a que ambas variables son punteros constantes (al ser nombres de variables de arrays), no puede realizarse, ya que no puede modificarse su contenido.
En la segunda asignación V[1] = A[1] , el compilador interpreta la expresión A[1] como conteniendo la dirección de la segunda fila de la matriz, que es un puntero a un vector de 10 reales, por lo que es un puntero a reales. Por otro lado, para el compilador V[1], es un puntero a reales y, por tanto, su valor es del mismo tipo que A[1]. Todo lo anteriormente indicado permite realizar la asignación indicada.
La expresión de la derecha &mt[2][0] , de la tercera asignación V[2] = &mt[2][0] , proporciona la dirección del primer elemento de la tercera fila de la matriz, por consiguiente es también de tipo puntero a real al igual que el lado izquierdo de la asignación V[2] , por lo que la asignación es correcta.

Ejemplo. Direcciones ocupadas por punteros asociados a una matriz.
El programa muestra las direcciones ocupadas por todos los elementos de una matriz de reales dobles de 5 filas y 4 columnas, así como las direcciones de los primeros elementos de cada una de las filas, accedidos por un puntero a fila.
Observe que la salida se produce en hexadecimal. La dirección de un elemento de la matriz se obtiene del anterior sumándole 8 en hexadecimal.

```
#include <cstdlib>
#include <iostream>
using namespace std;
#define N 5
#define M 4
double A[N][M];
int main( int argc, char *argv[])
{
    int i,j;
    cout <<" direciones de todos lo elementos de la matriz\n\n";
    for (i = 0; i < N; i++)
        {
            for (j = 0; j < M; j++)
                cout << " &A[" << i << "][" << j << "]=" << &A[i][j];
            cout << "\n";
        }
    cout << " direcciones de comienzo de las filas de la matriz\n\n";
    for (i = 0; i < N; i++)
        cout << " A["<<i<<"] = " << A[i]
        << " contiene direccion de &A[" << i << "][" << 0 << "]" << endl;
    cin.get();
    return EXIT_SUCCESS;
}
```

Ejemplo. Lectura y escritura de matrices mediante punteros. Escribir un programa que lea y escriba matrices genéricas mediante punteros y funciones.
Para poder tratar la lectura y escritura de matrices mediante funciones que reciban punteros como parámetros, basta con trasmitir un puntero a puntero. Además, se debe informar a cada una de las funciones el número de columnas y filas que tiene (aunque sólo es necesario el número de columnas), por lo que las funciones pueden ser declaradas de la siguiente forma: *void escribir_matriz(int ∗∗ A, int f, int c)* y *void*

leer_matriz(int ∗∗ A, int f, int c) donde f y c son respectivamente el número de filas y el número de columnas de la matriz. Para tratar posteriormente la lectura y escritura de datos en cada una de las funciones hay que usar $*(*A + c * i + j)$. Las llamadas a ambas funciones, deben ser con un tipo de dato compatible tal y como se hace en el programa principal. En el programa, además se declaran el número de filas F, y el número de columnas C como macros constantes.

```
#include <cstdlib>
#include <iostream>
#define F 3
#define C 2
using namespace std;
int A[F][C];
void leer_matriz( int ** A, int f, int c)
{
    int i, j;
    for (i = 0; i < f; i++)
        {
            for(j = 0; j < c ; j++)
                cout << " " << *(*A + c*i+j);
            cout << endl;
        }
}

void escribir_matriz( int** A, int f, int c)
{
    int i, j;
    for (i = 0; i < f; i++)
    {
        for(j = 0; j < c; j++)
        {
            cout << "ingrese la fila " << i << " columna " << j << endl;
            cin >> *(*A + c*i+j);
        }

    }

}

int main( int argc, char *argv[])
{
    int * a = &A[0][0];
    escribir_matriz(&a,F,C);
    leer_matriz(&a,F,C);

    cin.get();
    return EXIT_SUCCESS;
}
```

8. Punteros como argumentos de funciones

Cuando se pasa una variable a una función (paso por valor) no se puede cambiar el contenido de esa variable. Sin embargo, si se pasa un puntero a una función (paso por dirección) se puede cambiar el contenido de la variable a la que el puntero apunte. El paso de un nombre de array a una función es lo mismo

que pasar un puntero al array . Se pueden cambiar cualquiera de los elementos del array. Cuando se pasa un elemento a una función, sin embargo, el elemento se pasa por valor.

> Los parámetros dirección son más comunes en C, dado que en C++ e xisten los parámetros por referencia que resuelven mejor la modificación de los parámetros dentro de funciones.

Ejemplo. Paso por referencia en C y en C++.

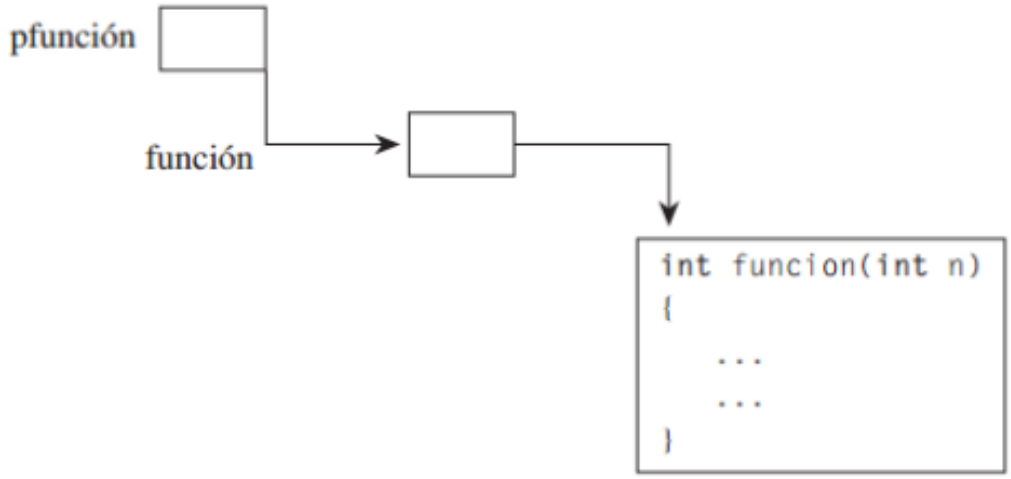
```
#include <cstdlib>
#include <iostream>
using namespace std;
struct datos
{
   float mayor, menor;
};

void leer_registrodatosCmasmas(datos &t) // parámetro por referecia en C++
{
   float actual;
   cin >> actual;
   if (actual > t.mayor)
   t.mayor = actual;
   else if (actual < t.menor)
   t.menor = actual;
}

void leer_registrodatosC(datos *t) // parámetro dirección C, referencia
{
   float dat;
   cin >> dat;
   if (dat > t -> mayor)
   t -> mayor = dat;
   else if (dat < t -> menor)
   t -> menor = dat;
}

int main( int argc, char *argv[])
{
   datos dat;
   .....
   leer_registrodatosCmasmas(dat); // llamada por referencia en C++
   leer_registrodatosC(&dat); // llamada por dirección en C
   cin.get();
   return EXIT_SUCCESS;
}
```

9. Punteros a funciones

Un puntero a una función es simplemente un puntero cuyo valor es la dirección del nombre de la función. Dado que el nombre es, en sí mismo un puntero, un puntero a una función es un puntero a un puntero constante. Mediante un puntero cuyo valor sea igual a la dirección inicial de una función se puede llamar a una función de una forma indirecta. La sintaxis general para la declaración de un puntero a una función es:

Tipo_de _retorno (PunteroFuncion) (<lista de parámetros>);

Este formato indica al compilador que PunteroFuncion es un puntero a una función que de vuelve el tipo Tipo_de_retorno y tiene una lista de parámetros.



La función asignada a un puntero a función debe tener el mismo tipo de retorno y lista de parámetros que el puntero a función; en caso contrario, se producirá un error de compilación. La sintaxis general de incialización de un puntero a función y el de llamada son respectivamente:

```
PunteroFuncion = una funcion
PunteroFuncion(lista de parametros)
```

Recuerde

- *func*, nombre de un elemento.
- *func[]* es un array.
- *(*func[])* es un array de punteros.
- *(*func[])( )* es un array de punteros a funciones.
- *int (*func[])( )* es un array de punteros a funciones que de vuelven valores int.

Ejemplo. Array de punteros a funciones. Se quiere evaluar las funciones $f_1(x)$, $f_2(x)$ y $f_3(x)$ par a todos los valores de x en el intervalo $0,3 \leq x \geq 4,1$ con incremento de $0,5$. Escribir un programa que evalúe dichas funciones utilizando un array de punteros a función. Las funciones son las siguientes:

$$f_1(x) = 3 * \sin(x) + 2{,}5 * \cos(x) \qquad f2(x) = -x * \sin(x) + x * x; \qquad f_3(x) = x^2 - x + 1$$

Se trata, como se hace en la codificación, de definir las tres funciones f_1, f_2 y f_3 al estilo de C++ y después definir un array de punteros a las funciones, a los que se asignan cada una de las funciones préviamente definidas. El acceso a dichas funciones para su ejecución es similar al acceso que se realiza con cualquier otro tipo de dato cuando se accede con punteros.

```
#include <cstdlib>
#include <iostream>
#include <math.h>
#define maxf 3
#define minx 0.3
#define maxx 4.1
#define incremento 0.5
```

```
using namespace std;
float f1 ( float x)
{
    return (3 * sin(x)+ 2.5*cos(x));
}

float f2( float x)
{
    return (-x * sin (x) + x*x);
}

float f3( float x)
{
    return ( x * x - x + 1);
}

int main(int argc, char *argv[])
{
    float (*Array_de_Funciones[maxf]) ( float);
    // array de punteros a funciones que retornan reales
    Array_de_Funciones [0] = f1;
    Array_de_Funciones [1] = f2;
    Array_de_Funciones [2] = f3;
    for ( int i = 0; i < maxf; i++)
        {
            cout << " funcion " << i +1 << endl;
            for ( float x = minx; x < maxx; x += incremento)
                cout << " x = "<< x << " f = " << Array_de_Funciones [i](x)
                 << endl;
        }
    cin.get();
    return EXIT_SUCCESS;
}
```

Ejemplo. Paso de funciones como parámetros a otras funciones. Cálculo de la suma de los n primeros términos de una serie genérica.

Se trata de sumar los n primeros términos de una serie genérica $suma = \sum_{i=1}^{n} t_i$. La función *sumaterminos*, recibe como parámetro el número de términos n a sumar y una función *fun* que calcula el término genérico. Para probar la función se usan los términos de dos series

$$serie_1 = \sum_{i=1}^{n} \frac{3}{i*i} \qquad serie_2 = \sum_{i=1}^{n} \frac{1}{i}$$

Las funciones *terminoserie1* y *terminoserie2*, calculan los términos de cada una de las series.

```
#include <cstdlib>
#include <iostream>
using namespace std;
double Sumaterminos( int n, double (*fun) ( int ))
{
   double acu = 0;
   for ( int i = 1; i <= n; i++)
      acu += fun(i);
   return acu;
```

```
}
double terminoserie1( int i)
{
   return (double)3.0 / (i * i);
}
double terminoserie2( int i)
{
   return (double)1.0 / i;
}
int main( int argc, char *argv[])
{
   cout << "Suma de cinco elemento de serie1: "
        << Sumaterminos(5, terminoserie1) << endl;
   cout << "Suma de cuatro terminos de serie2: "
        << Sumaterminos(3, terminoserie2) << endl;
   cin.get();
   return EXIT_SUCCESS;
}
```

10. Punteros a estructuras

Cuando se referencia un miembro de una estructura usando el nombre de la estructura, se emplea el operador punto (.) . En cambio, cuando se referencia una estructura utilizando el puntero estructura, se emplea el operador flecha (->) para acceder a un miembro de ella.

Ejemplo. Acceso a miembros de estructuras. Dadas las siguientes declaraciones de estructuras, se trata de escribir cómo acceder a los atributos *dat, dia* y *mes* de la variable estructura *est* de tipo *datos.* ¿Qué problemas habría en la siguiente sentencia? *cin.getline(est.nombre,40).*

```
struct fechas
{
int dia, mes, anyo;
float dat;
};

struct datos
{
char* nombre;
fechas * fec;
} est;
```

La variable *est* es una estructura que tiene dos punteros, para acceder a sus miembros se debe utilizar el operador ·

```
est.nombre;
est.fec;
```

Tanto *nombre* como *fec* son punteros, este último a la estructura *fechas,* accediendo a los datos con el operador flecha − >

```
est.fec -> dia;
est.fec -> mes;
est.fec -> anyo;
est.fec -> dat;
```

El puntero *nombre* es de tipo *char* pero no apunta a ningún lado, la instrucción *cin.getline(est.nombre, 40)* va a leer la línea pero no va a saber donde colocarla, para ello se debe reservar memoria. En C utilizamos el comando *malloc*, en C++ la reserva de memoria se realiza mediante el comando *new*, procediendo de la siguiente forma.

```
est.nombre = new char[41] ; // reserva 41 lugares de tamaño char
cin.getline(est.nombre, 40);
```

Ejemplo. Lectura de una estructura con parámetro de tipo puntero a estructura, y visualización con parámetro de tipo estructura.

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

struct articulo
{
    char nombre[81];
    int codigo;
};

void leerarticulo( articulo * particulo)
{
    cout << "nombre del articulo :";
    cin.getline(particulo->nombre, 80);
    cout << "Codigo del articulo :";
    cin >> particulo->codigo;
}

void mostrararticulo ( articulo articul)
{
    cout << "nombre del articulo :" << articul.nombre << endl;
    cout << "Codigo asociado al articulo : " << articul.codigo << endl;
}

int main( int argc, char *argv[])
{
    articulo a;
    leerarticulo(&a);  // paso la dirección de la estructura a (puntero)
    mostrararticulo(a); // paso los valores de la estructura a
    cin.get();
    return EXIT_SUCCESS;
}
```

11. Problemas

1. Encontrar los errores de las siguientes declaraciones de punteros:

```
int x, *p, &y;
char* b= "Cadena larga";
char* c= 'C';
float x;
void* r = &x;
```

2. Un array unidimensional se puede indexar con la aritmética de punteros. ¿Qué tipo de puntero habría que definir para indexar un array bidimensional?

3. El código siguiente accede a los elementos de una matriz. Acceder a los mismos elementos con aritmética de punteros.

```
#define N 4
#define M 5
int f,c;
double mt[N][M];
. . .
for (f = 0; f < N; f++)
{
for (c = 0; c < M; c++)
cout << mt[f][c];
cout << "\n";
}
```

4. Escribir una función con un argumento de tipo puntero a double y otro argumento de tipo int. El primer argumento se debe de corresponder con un array y el segundo con el número de elementos del array. La función ha de ser de tipo puntero a double para devolver la dirección del elemento menor.

5. Dada las siguientes definiciones y la función gorta:

```
double W[15], x, z;
void *r;
double* gorta( double* v, int m, double k)
{
int j;
for (j = 0; j < m; j++)
if (*v == k)
   return v;
return 0,
}
```

¿Hay errores en la codificación? ¿De qué tipo?
¿Es correcta la siguiente llamada a la función?:

```
r = gorta(W,10,12.3);
```

¿Y estas otras llamadas?:

```
cout << (*gorta(W,15,10.5));
z = gorta(w,15,12.3);
```

6. ¿Qué diferencias se pueden encontrar entre un puntero a constante y una constante puntero?

7. En física se presentan a menudo magnitudes que son el resultado del producto vectorial, por ejemplo:

Torca: $\vec{\tau} = \vec{r} \times \vec{F}$
Aceleración tangencial: $\vec{a}_t = \vec{\alpha} \times \vec{r}$
Fuerza generada por un campo magnético sobre una carga móvil: $\vec{F}_B = q\vec{v} \times \vec{B}$

Para obtener el producto vectorial se puede realizar colocando el primer vector con sus componentes y el segundo vector como una matriz antisimétrica como se muestra a continuación:

$$a = [a_1 \quad a_2 \quad a_3]$$

$$b = [b_1 \quad b_2 \quad b_3]$$

$$\vec{a} \times \vec{b} = [a_1 \quad a_2 \quad a_3] \begin{pmatrix} 0 & -b_3 & b_2 \\ b_3 & 0 & -b_1 \\ -b_2 & b_1 & 0 \end{pmatrix}$$

Realice un programa que calcule el producto vectorial:

a) Pasando los parámetros por valor.

b) Utilizando punteros a funciones.

c) Utilizando un arreglo de funciones.

d) Utilizando una estructura que cargue los datos de cada vector.

e) Utilizando una estructura que cargue la dirección de la estructura con los datos de cada vector y un parámetro multiplicador del producto vectorial.

f) Utilizando un arreglo de punteros a funciones.

g) Utilizando sobrecarga de funciones para los items anteriores.

8. Escribir un programa que decida si una matriz de números reales es simétrica. Utilizar

   a) una función de tipo bool que reciba como entrada una matriz de reales, así como el número de filas y de columnas,y decida si la matriz es simétrica

   b) otra función que genere la matriz de 10 filas y 10 columnas de números aleatorios de 1 a 100

   c) un programa principal que realice llamadas a las dos funciones. Nota: usar la transmisión de matrices como parámetros punteros y la aritmética de punteros para la codificación.

9. Codificar funciones que realicen las siguientes operaciones: multiplicar una matriz por un número, y rellenar de ceros una matriz. Nota: usar la aritmética de punteros.

10. Escribir un programa que permita calcular el área de diversas figuras: un triángulo rectángulo, un triángulo isósceles, un cuadrado, un trapecio y un círculo. Nota: utilizar un array de punteros de funciones, siendo las funciones las que permiten calcular el área.