Universidad Nacional del Litoral
Facultad de Ingeniería y Ciencias Hídricas
Departamento de Informática

Ingeniería Informática

PROGRAMACIÓN ORIENTADA A OBJETOS

UNIDAD 1
Punteros

Ing. Horacio Loyarte

Facultad de Ingeniería y Ciencias Hídricas
FICH
UNL

Unidad 1

Punteros

Resumen de Conceptos

Introducción

Un puntero es una variable que representa un valor numérico correspondiente a la ubicación física de un elemento determinado del programa. Ese valor numérico asignado a un puntero también es conocido como *dirección de memoria* del elemento. Dicho elemento puede constituir en C++:

- Un dato simple
- Una estructura de datos
- Una función
- Una variable de cualquier tipo

Es decir que podemos emplear punteros para referenciar datos o estructuras más complejas y también administrar bloques de memoria asignados dinámicamente. Su empleo hace más eficiente la administración de recursos del programa y su ejecución.

Muchas funciones predefinidas de C++ emplean punteros como argumentos e inclusive devuelven punteros. Para operar con punteros C++ dispone de los operadores **`&`** y **`*.`**

Una de las ventajas más interesantes del uso de punteros es la posibilidad de crear variables dinámicas; esto es, variables que pueden crearse y destruirse dentro de su ámbito, lo cual permite optimizar el uso de recursos disponibles en un programa.

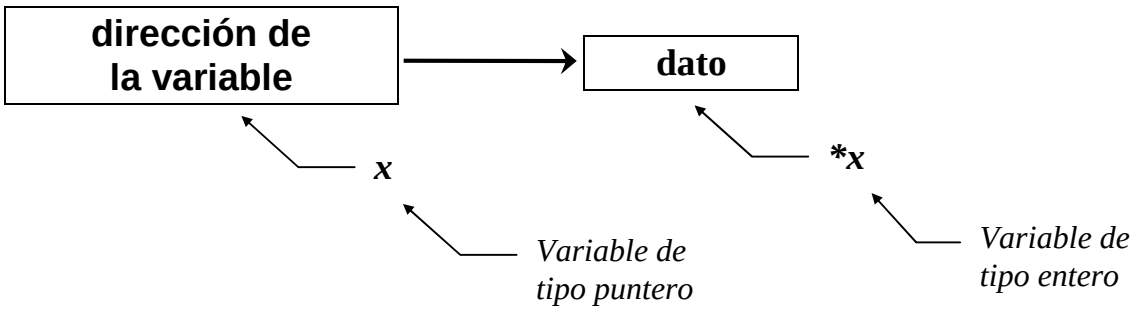
Definición de Punteros

Para declarar un puntero, C++ emplea el operador **`*`** anteponiéndolo a la variable puntero. Además se debe indicar el tipo de dato al que apuntará esta variable.

```
tipo  *variable_puntero;
```

Ejemplo:

```
int *x; // se declara a x como variable puntero a un entero
```



En el ejemplo anterior, la variable puntero **x** contiene la dirección donde se almacenará un dato de tipo entero, pero ***x** es una expresión nemotécnica que corresponde a un entero; por lo tanto podemos emplear ***x** en cualquier proposición o expresión de C++ que requiera la presencia de un entero.
En otras palabras: ***x** es una expresión que representa un valor entero almacenado en la dirección de memoria contenida en **x**.

Relación entre los operadores de referencia & y de desreferencia *

Como vimos en el epígrafe anterior, es posible declarar una variable puntero a través del operador * y asignarle la dirección de memoria de algún elemento del programa.

Pero si declaramos una variable simple cualquiera: ¿Cómo determinamos su dirección?. Como ya estudiamos, en C++ es posible conocer la dirección de cualquier variable a través del *operador dirección* que representamos con: **&**. Ese dato solo podremos asignarlo a una variable que admita direcciones, es decir a una variable puntero:

```
int  z=8;  // declaramos e inicializamos con 8 una variable entera z
int *p;  // declaramos un puntero p a un entero
p= &z;  // asignamos la dirección de z al puntero p
cout<< *p <<endl;  // informamos 8, el dato apuntado por p
cout<< p;  // informamos la dirección de memoria contenida en p
```

Nota: la asignación de la dirección de **z** a **p** fue posible pues **z** es entero y **p** fue declarada como puntero a un entero.

En resumen: el operador **&** seguido de un operando permite obtener la dirección de dicho operando; por el contrario, el operador * permite obtener el dato ubicado en la dirección asignada a una variable puntero. De allí que se denomine a estos operadores:

&: operador dirección o referencia
***:** operador indirección o desreferencia;

La constante NULL

Si una variable puntero no ha sido inicializada apunta a una dirección aleatoria. Si la lógica del programa requiere que una variable puntero no apunte a ningún dato ni elemento del programa podemos asignarle la constante **NULL.**

```
float *q = NULL;
cout << q;   //devuelve la dirección nula 00000000
```

El objeto de definir con **NULL** a una variable puntero tiene que ver con controles que suelen ser útiles en algoritmos que emplean estructuras de datos con punteros.

Nota: Los compiladores C++ devuelven un entero en base hexadecimal de 8 o 16 cifras para definir una dirección de memoria.

Variables dinámicas: new y delete

Al declarar una variable puntero a un float escribimos: **`float *q;`**
Esto significa que el compilador de C++ reserva memoria para el puntero **q**, donde este almacenará la dirección de inicio de un dato flotante. Pero esa dirección aún no ha sido alocada con un dato y podría suceder que esté siendo usada por alguna otra variable. Si intentamos acceder a la dirección que este apunta se producirá un error.

```
float *q=3.14159;//ERROR:no storage has been allocated for *q
(//ERROR: No hay almacenamiento reservado para ubicar a *q)
```

Vimos que una forma adecuada de evitar tal error es la siguiente:

```
float x=3.14159;  // x almacena el valor 3.14159
float *q=&x  ;    // q contiene la dirección de x
cout<<*q ;  // OK: *q ha sido convenientemente ubicado
```

En este caso no habrá inconveniente para acceder a ***q** porque previamente se creó la variable **x**, y luego se le dio a **q** la dirección de **x**.
Otra forma de resolver la asignación de un espacio de memoria para definir un puntero es destinar la memoria necesaria para almacenar el dato que será apuntado por el puntero en cuestión empleando el operador **new:**

```
float *q;    //se declara el puntero a flotante q
q= new float; //reserva un espacio de almacenamiento para *q
*q=3.14159; // OK: q y *q se han inicializado sin error
```

Se pueden combinar las líneas anteriores en una sola:

```
float *q=new float(3.14159);
```

Esto permite ubicar un dato en memoria en tiempo de ejecución. También en tiempo de ejecución se puede efectuar la operación **delete** inversa a **new,** desalocando la memoria apuntada y dejándola disponible para que el programa la emplee para almacenar otro dato.

```
delete q; //libera el espacio de memoria apuntada por q
```

Operaciones con Punteros

Las variables puntero pueden ser operadas aritméticamente; esto nos permite referenciar una nueva dirección de memoria. Esa nueva dirección dependerá del tipo de dato al que apunta la variable puntero. Por ejemplo:

```
int *p;
p+=3;  /*La nueva dirección de p supera a la anterior en 12 bytes, pues al sumar 3
estamos desplazándonos la cantidad de bytes correspondientes a 3 enteros (12
bytes) */
double *r;
```

r+=2; /*La nueva dirección de **r** supera a la anterior en 16 bytes pues al incrementar en 2 su dirección, la estamos desplazando la cantidad de bytes correspondientes a 2 datos de tipo double (16 bytes)*/

Ampliando estos conceptos podemos resumir las siguientes operaciones como válidas para las variables de tipo puntero:

a) Se puede asignar a una variable puntero la dirección de una variable no puntero.

```
float x, *p;
.....
p=&x;
```

b) A una variable puntero puede asignarse el contenido de otra variable puntero si ambas variables son compatibles (ambos punteros apuntan al mismo tipo de dato).

```
int *u, *v;
.....
u=v;
```

c) A un puntero es posible asignarle el valor NULL (el puntero no apunta a dirección de memoria alguna).

```
int *p;
p=NULL;   //Dirección nula: 00000000
```

d) Es posible sumar o restar una cantidad entera **n** a una variable puntero. La nueva dirección de memoria obtenida difiere en una cantidad de bytes dada por: **n** por el tamaño del tipo apuntado por el puntero.

```
int *p;
.....
p+=4;  //la dirección original de p se ha incrementado 16
bytes
p-=1;  //La dirección anterior de p se decrementó en 4
bytes.
```

e) Es posible comparar dos variables puntero.

```
u<v       u>=v       u==v          u!=v          u==NULL
```

f) Una variable puntero puede ser asignada con la dirección de una variable creada dinámicamente a través del operador **new**.

```
float *q; // se declara q como puntero a un float
q= new float; /* se asigna a q la dirección de una nueva
variable */
*q=4.1513; //se almacena un float en la dirección de q
```

Paso de punteros a una función

El objeto de pasar punteros a una función es poder modificar los parámetros de llamada, es decir, permite efectuar un pasaje por referencia.

Al pasar un puntero como parámetro por argumento de una función se pasa realmente la dirección del argumento; cualquier cambio que se efectúe en los datos ubicados en esa dirección, se reflejarán en el bloque de llamada y en la propia función.

Esto es radicalmente diferente al pasaje de parámetros por valor donde las direcciones de los argumentos de llamada son diferentes a las direcciones de los argumentos formales.
Para observar la diferencia entre los diferentes pasajes de parámetros, analicemos primeramente un ejemplo donde apliquemos el pasaje de parámetros por valor:

*Llamada a la función **f_porvalor** donde los parámetros actuales son pasados por valor.*

```
/*** ejemplo ***/
#include <iostream.h>
#include<conio.h>

void f_porvalor(int x,int y);

void main( )
{
  int dato1=5,dato2=10;
  cout<<"Datos iniciales: dato1="<<dato1<<" dato2="<<dato2<<"\n";
  cout<<"\n Pasaje por valor \n";
  f_porvalor(dato1, dato2);
  cout<<"Despues de llamar a la funcion
f_porvalor(dato1,dato2):\n";
  cout<<"Datos iniciales: dato1="<<dato1<<" dato2="<<dato2<<"\n";
  getch();
}
void f_porvalor(int x,int y)
{ x=x*100; y=y*100;
  cout << "Dentro de la funcion: x="<<x<<" y="<<y<<"\n";
}
```

La salida de este programa será

```
Datos iniciales: dato1=5  dato2=10
Pasaje por valor
Dentro de la función: x=500  y=1000
Después de llamar a la función f_porvalor(int x, int y)
Datos iniciales: dato1= 5  dato2=10
```

Obsérvese que los datos iniciales no se ven alterados en main().
En el ejemplo siguiente se plantea un caso similar, pero pasando parámetros a través de punteros:

*Llamada a la función **f_porpunteros** donde los parámetros actuales son pasados por referencia.*

```
/*** ejemplo ***/
#include <iostream.h>
#include <conio.h>

void f_porpunteros(int *p,int *q);

void main( )
{ int dato1=5,dato2=10;
  cout<<"Datos iniciales: dato1="<<dato1<<" dato2="<<dato2<<"\n";
  cout<<"\n Pasaje por referencia \n";
  f_porpunteros(&dato1, &dato2);
  cout<<"Despues de llamar a la funcion f_porpunteros(int *p,int*q);
  cout<<"dato1="<< dato1<<" dato2="<<dato2<<"\n";
  getch();
}
void f_porpunteros(int *p,int *q)
```

```
{ *p+=45; *q+=60;
   cout << "Dentro de la funcion: *p="<<*p<<" *q="<<*q<<"\n";
}
```

```
Datos iniciales: dato1=5  dato2=10
Pasaje por referencia
Dentro de la función: *p=50  *q=70
Después de llamar a la función f_porpunteros(int *p, int *q)
Datos iniciales: dato1= 50  dato2= 70
```

En este caso los parámetros actuales dato1 y dato2 se han modificado pues en la función se ha trabajado con sus direcciones.

En el caso de pasaje de un parámetro de tipo array a una función debemos considerar que el nombre de un arreglo representa a la dirección del primer elemento del arreglo. Por esto, no es necesario emplear el símbolo & al llamar a una función con un parámetro actual de tipo arreglo.

```
int   x[6]={5,9,12,45,41,11};
func(x); /*Llamada a una función empleando como parámetro la
dirección de inicio del arreglo x, es decir: &x[0] */
```

Punteros a arreglos lineales

En C++ los punteros y el tipo arreglo están íntimamente relacionados. Las declaraciones de arreglos que hemos estudiado pueden plantearse a través de punteros logrando mayor eficiencia en la ejecución del programa.

Como dijimos, el nombre de un arreglo representa la dirección del primer elemento del arreglo, es decir, el nombre es un puntero al inicio de esa estructura.

```
int   x[6]={5,9,12,45,41,11};
```

Esto significa que en el arreglo lineal **x** del ejemplo, la dirección de su primer componente puede ser referenciada con el propio identificador del arreglo **x**, o también con **&x[0]**.

Para referenciar al segundo elemento del arreglo podemos hacerlo de 2 formas equivalentes: **&x[1]** y **x+1**.

```
cout << x+1 << endl;    /* obtenemos como salida la dirección
del elemento 1 del arreglo. Por ejemplo: 0000FFEE   */
cout << &x[1] << endl;  /* otra forma de obtener como salida
la dirección del elemento 1 del arreglo: 0000FFEE   */
```

O sea que, la dirección del elemento i-ésimo de un arreglo **x** será **x+i-1** o bien **&x[i-1]**.

Para expresar el contenido del elemento i-ésimo de un arreglo, podemos emplear también dos formas: **x[i-1]** y ***(x+i-1)**

Observemos el siguiente programa C++ donde se obtienen y muestran en la salida las direcciones de memoria de cada componente del arreglo **x** :

```
#include <iostream.h>
void main()
{
 int x[]={12,13,14,15,16};
 for (int i=0; i<5; i++)
    cout<<&x[i]<<endl;
}
```

La salida que se obtiene es similar a la siguiente:

```
0012FF78
0012FF7C
0012FF80
0012FF84
0012FF88
```

Obsérvese que las direcciones de memoria de los 6 componentes del arreglo **x** (en base hexadecimal) saltan de 4 en 4. Esto es debido a que cada elemento del arreglo es de tipo **int** y requiere 4 bytes de memoria. Si el arreglo fuera de elementos de tipo **float** las direcciones también saltarán de 4 en 4 (cada número de tipo **float** ocupa 4 bytes). Para el tipo **double** el salto sería de 8 bytes.

Obsérvese a continuación 4 maneras distintas de asignar el sexto y último elemento del arreglo **x** al segundo elemento de dicho arreglo:

```
x[1] = x[5] ;
x[1] = *(x+5) ;
*(x+1) = x[5]  ;
*(x+1) = *(x+5) ;
```

Entonces, de acuerdo con el concepto de que el nombre de un arreglo representa a la dirección de su primer elemento, podemos declarar al arreglo lineal **x** de la manera siguiente:

```
int *x;
```

La diferencia con la declaración **`int x[6]={5,9,12,45,41,11}`**; se basa en que **`*x`** no reserva un espacio de memoria para todos los elementos del arreglo. Es necesario definir un bloque de memoria para alojar la estructura. Esto es posible utilizando el operador **new**:

```
x = new int[6];
```

La expresión anterior asigna a **x** un puntero a un bloque de memoria con cantidad de bytes requerida para almacenar el arreglo de 6 elementos enteros.

Para liberar la memoria reservada para un arreglo se utiliza el operador **delete[]** (agregando corchetes vacios antes del puntero). Ejemplo:

```
delete []x;
```

Importante: si en la declaración de un arreglo se incluye la incialización entre llaves de los elementos del mismo, se lo tiene que declarar como arreglo estático y no se puede emplear el operador *.

```
/* Ejemplo: generar aleatoriamente un arreglo de 20 números natura-
les menores que 1000. Luego insertar en la posición 5 (sexto elemen-
to) el valor  -45. El programa debe mostrar el nuevo arreglo de 21
elementos */
#include <stdlib.h>
#include <alloc.h>
#include <iostream>
#include <iomanip.h>
using namespace std;

void main()
{
int *x,i;
x=new int[21];

for(i=0; i<20; i++)
 { *(x+i)=rand()%(1000);
   cout<<setw(8)<< *(x+i); }

for (i=20; i>5; i--)
   *(x+i)=*(x+i-1);

*(x+5)=-45;

cout<<endl<<endl;
for( i=0; i<21; i++)
   cout<<setw(8)<< *(x+i);

delete []x;

} // fin del programa
```

*Reserva de memoria para 21 elementos enteros del arreglo **x***

En el for se genera y muestra el arreglo de enteros aleatorios

Se desplazan los elementos para poder usar la posición 5 e insertar el nuevo dato.

Se inserta el dato -45 en la posición 5 del array.

Se muestra el nuevo arreglo de 21 componentes.

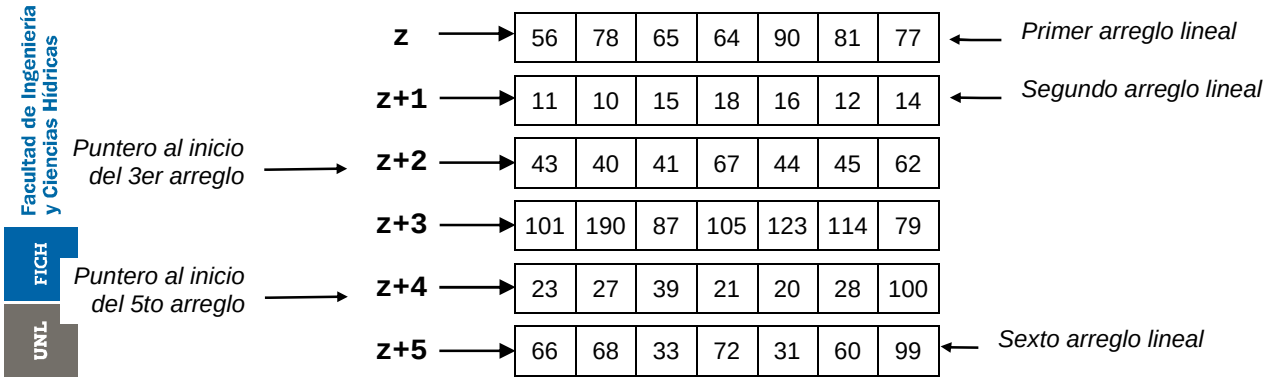
Se libera la memoria previamente reservada

Punteros a Arreglos Multidimensionales

Consideremos el caso de un arreglo bi-dimensional o matriz. Podemos considerarlo como una lista de arreglos unidimensionales (lista de filas por ejemplo). Si tenemos en cuenta que un arreglo uni-dimensional se define con un puntero a su primer elemento, una lista de arreglos unidimensionales para plantear una matriz puede definirse empleando punteros.

Supongamos que deseamos operar con una matriz de 6 filas por 7 columnas. Para ello declararemos dicha estructura empleando un puntero a una colección de 6 arreglos lineales contiguos de 7 elementos cada uno:

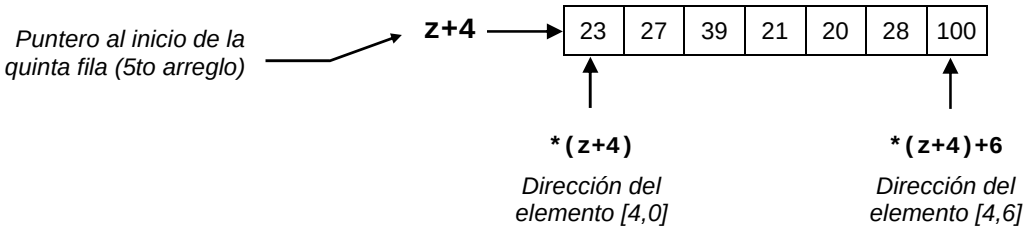
```
int (*z)[7];  //en lugar de declarar z[6][7]
```



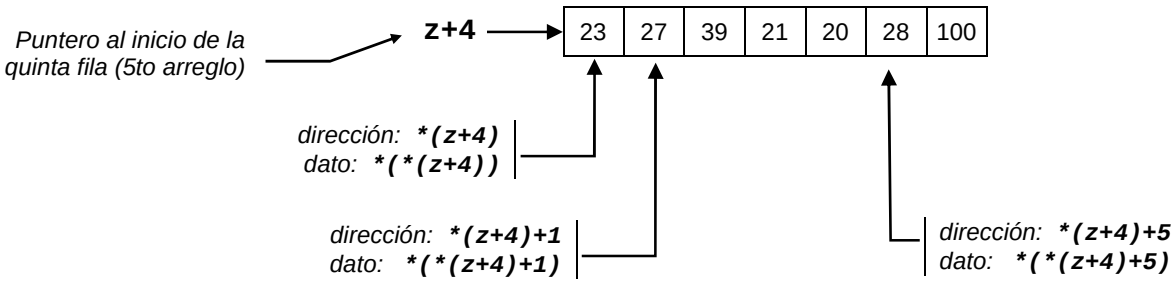
Es muy importante la presencia del paréntesis **(*z)**, pues de otro modo, estaríamos definiendo un arreglo de punteros. Esto está de acuerdo con los operadores asterisco y corchetes que se evalúan de derecha a izquierda.

Estudiemos cómo acceder a los elementos de esta matriz, definida a través de una colección de arreglos lineales contiguos y empleando un puntero **z** al primer elemento del primer arreglo.

Tomemos el caso de la fila 4 (quinta fila)



Como **z+4** es un puntero a la quinta fila ***(z+4)** será el objeto apuntado. Pero el objeto apuntado resulta ser un arreglo lineal de 7 elementos y por tanto ***(z+4)** hace referencia a toda la fila; por ello para identificar a los elementos de esta fila y acceder a los datos debemos desreferenciar las direcciones de estos punteros:



Recordemos que el nombre de un arreglo es un puntero a su primer elemento. Por eso ***(z+4)** es un puntero al inicio de la quinta fila, pero a su vez es el nombre de un arreglo lineal: la quinta fila. Por ello necesitamos desrefreneciar la dirección con el operador ***** para obtener el dato.

En general, para operar con un arreglo bidimensional definido a través de un puntero **z** al inicio de una serie de arreglos lineales podemos decir:

Para indicar el puntero al inicio de la fila i:	***(z+i)**
Para identificar un puntero al elemento [i][j]:	***(z+i)+j**
Para identificar el dato ubicado en la fila i, columna j:	***(*z+i)+j)**

```
//Ejemplo del uso de punteros para acceso a arreglos;

 #include <iostream>

 using namespace std;

 int main(){
 const int filas=6;
 const int columnas=7;
 int a[filas][columnas];
 int i,j;

 for (i=0;i<filas;i++)
  for (j=0;j<columnas;j++){
    a[i][j]=random()%100;
    cout << a[i][j] << " ";
  }

 cout << endl << endl;
 int (*x)[columnas];

 x=&a[0];
 for (i=0;i<filas;i++)
  for (j=0;j<columnas;j++)
    cout << *((x[i])+j) << " ";
 return 0;
}
```

Punteros y *const*

Empleando la palabra reservada **const** podemos declarar como constante a:
i) una variable un puntero, o bien a: ii) el objeto apuntado.

Declaración del puntero p como constante

```
int dato=25;
int  *const  p=&dato;
```

Para definir una constante apuntado por un puntero q

```
float  valor=1.89;
const  float *q=&valor
```

Es posible que tanto el puntero como el objeto apuntado sean constantes:

```
const char *const r="Universidad";
```

Arreglos de Punteros

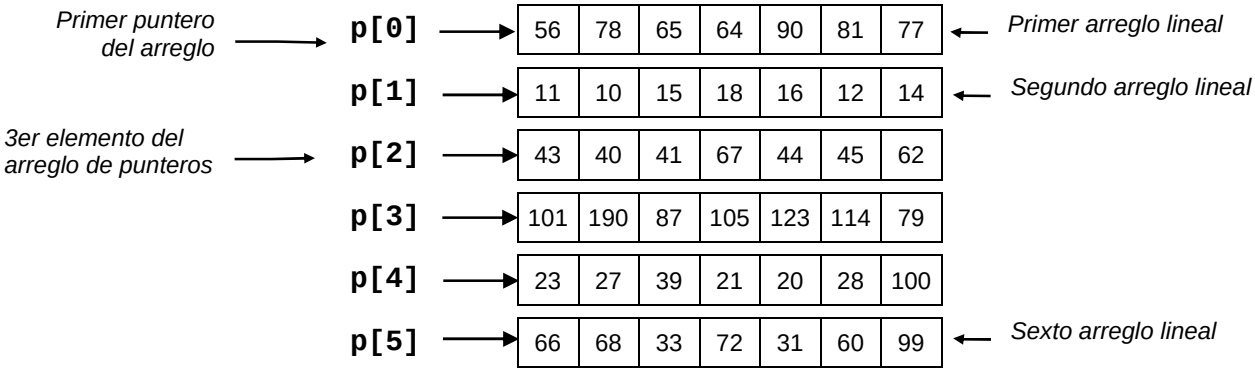
Es posible definir un arreglo en C++ cuyos elementos contengan direcciones de memoria en lugar de datos. Un arreglo de este tipo constituye un array de punteros. La sintaxis que debemos emplear en C++ es la siguiente:

```
int *p[20];
```

En este caso p[0] es el primer puntero y apunta a una entero; p[1] es el siguiente puntero que apunta a otro entero, hasta p[19] que constituye el último puntero.

Esta declaración puede ser útil para representar una matriz de 2 dimensiones. Pero en este caso debemos reservar memoria para cada una de las filas. Por ejemplo si deseamos representar la matriz de enteros de 6 filas por 7 columnas (que usamos antes) empleando un arreglo de punteros, tal reserva de espacio debe indicarse como sigue:

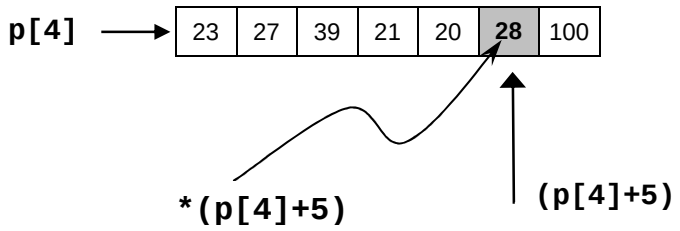
```
int *p[6]; //cantidad de punteros (filas)
for (int fila=0; fila<6; fila++)
   p[fila]=new int[7];
   // reservamos espacio para 7 datos enteros por cada fila
```



En este caso, para mostrar el elemento de la fila 4 columna 5 podemos hacer:

```
cout << *(p[4]+5);
```

Donde **p[4]** es un puntero al primer elemento de la fila 4. Al sumar 5 a ese puntero **(p[4]+5)** estamos haciendo referencia a la dirección (puntero) del sexto elemento de esa fila. El dato almacenado ene esa dirección se obtiene a través del operador de indirección: ***(p[4]+5)**



La ventaja de utilizar un arreglo de punteros radica en que el tamaño de ese primer arreglo también puede definirse dinámicamente como se muetra en el siguiente ejemplo:

```
//Arreglo de punteros

#include <iostream>

using namespace std;

int main(){
const int filas=6;
const int columnas=7;
int **p=new int*[filas];
int i,j;

for(i=0;i<filas;i++)
  p[i]=new int[columnas];

for (i=0;i<filas;i++)
  for (j=0;j<columnas;j++)
    *(p[i]+j)=i*10+j; // tambien se puede escribir p[i][j]=...


for (i=0;i<filas;i++)
  for (j=0;j<columnas;j++)
    cout << *(p[i]+j) << "  ";

for (i=0;i<filas;i++)
   delete [] p[i];
delete [] p;

return 0;
}
```

Se debe notar que por cada new debe existir un delete para liberar correctamente todos los bloques de memoria reservados.

Punteros y Funciones

Como en el caso de arreglos el nombre de una función representa la dirección de memoria donde se localiza dicha función. Por lo tanto ese puntero (a la función) puede emplearse y operarse como cualquier puntero a datos simples o a estructuras.

Ya estudiamos que un puntero puede pasarse como argumento de una función, lo que en este caso será pasar una función como argumento de otra función.

Al plantear un identificador de función como parámetro de otra función, debe interpretarse como un parámetro de tipo puntero (puntero a la función argumento) que de hecho es usado como si fuera una variable. La expresión: **`int (*f)();`**

define a **`f`** como un puntero a una función que retorna un entero. Nótese la presencia de paréntesis, obligatorios que rodean a **`*f`**. Sin ellos, la expresión **`int *f();`** estaría definiendo una función que retorna un puntero a un entero.

Luego de dicha declaración podemos decir que ***f** es la función y **f** un puntero a dicha función. Esto es muy similar al caso de arreglos, donde el nombre del arreglo era un puntero al inicio de dicha estructura.

```
#include <iostream.h>
#include <stdio.h>
#include <math.h>
#include<conio.h>

void complejas(void)       //función argumento
{  puts("Raices complejas");}

void resolvente( int x,int y,int z, void (*p)(void) )
  {  float dis,r1,r2;
     dis= y*y-4*x*z;
     if (dis <0)
      p();
     else
    { r1=(-y+sqrt(dis))/(2*x);
      r2=(-y-sqrt(dis))/(2*x);
      cout<<"r1="<<r1<<endl;
      cout<<"r2="<<r2; }
  }

void main(void)
{ int a,b,c;
  cout<<"Ecuacion de 2do grado\n";
  cout<<"a=";cin>>a;
  cout<<"b=";cin>>b;
  cout<<"c=";cin>>c;
  resolvente(a,b,c,complejas);//Llamada a función resolvente
}
```

Puntero a la función argumento

Llamada a la función argumento (complejas)

Cuál es la ventaja de emplear punteros a funciones?
Una utilidad muy interesante de emplear punteros a funciones es el hecho de poder invocar a una función empleando como parámetro funciones diferentes en cada llamada. Veamos esto en un ejemplo.

La función **sum** tiene 2 parámetros: el primero es un valor entero que será acumulado tantas veces como indique el siguiente parámetro. El primer argumento lo representaremos mediante una función que devuelve un entero y acepta como argumento otro entero.

```
int sum(int (*pf)(int a),int n); //función que suma n veces un entero
```

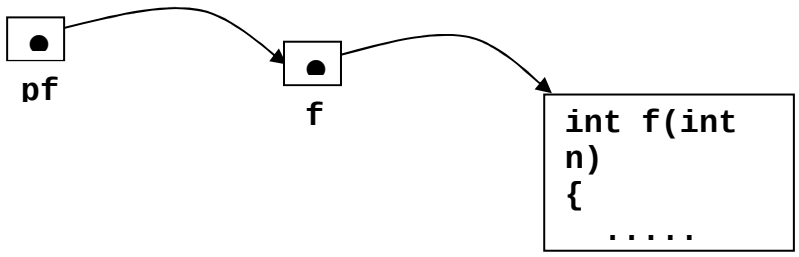
El siguiente programa C++ emplea **sum** para informar la suma de los cuadrados (1+4+9+16) y de los cubos (1+8+27+64) de los primeros 4 números naturales

```
int sum(int (*pf)(int a),int n);
{int acum=0;
 for (int i=1;i<=n;i++)
      acum+=(*pf)[i]
 return acum;  }

 int cuad(int a) {return a*a;}

int cubo(int a) {return a*a*a;}
```

```
void main(void)
{ cout<<sum(cuad,4)<<endl;
  cout<<sum(cubo,4)<<endl;
}
```

En conclusión podemos afirmar que un puntero a una función es un puntero que representa la dirección del nombre de la función- Pero como el nombre de una función es a su vez un puntero, decimos que un puntero a una función es un puntero a una constante puntero.

```
int f(int x);    //declaración de la función f
int (*pf)(int x);//declaración del puntero pf a función
pf=&f;           //asignamos la dirección de f al puntero pf
```



Arreglos dinámicos

Estudiamos que el nombre de una arreglo es un puntero alocado en tiempo de compilación,

```
float x[50];            // arreglo estático tradicional
```

La declaración siguiente define un arreglo dinámico creado en tiempo de ejecución.

```
float *q = new float[50];   // arreglo dinámico
```

En C++ disponemos del operador **`delete`** para liberar la memoria asignada por **`new.`**

```
delete [] q;  // libera la memoria apuntada por q
```

Funciones malloc y free

Para alocar y desalocar memoria dinámicamente también se pueden utilizar las funciones malloc (que recibe el tamaño del bloque en bytes y devuelve un puntero void que se debe convertir mediante cast al tipo adecuado) y free (que recibe el puntero). Estas funciones han sido heredadas de C y también están presentes en C++. Sin embargo, en muchos casos es conveniente utilizar new y delete, no solo por tener una sintaxis más clara, sino porque cuando se trabaja con objetos new y delete construyen y destruyen (conceptos que se estudiaran en la siguiente unidad) los objetos además de reservar el bloque de memoria.

```
int *x=(int*)malloc(sizeof(int)*10); // reserve para 10 int
...
free(x); // libera la memoria reservada
```