Unidad 1
Introducción a las Estructuras de Datos

1.1 Tipos de datos abstractos (TDA).
1.2 Modularidad.
1.3 Ejemplos de uso de TDAs.
1.4 Manejo de memoria Estática.

1.1 Tipos de Datos Abstractos

Aunque la programación de computadoras es realizada por personas con conocimientos extensos en las ciencias computacionales, **esconder la complejidad natural de los datos** a los diseñadores y programadores de computadoras ha permitido alcanzar el grado de desarrollo que tiene en la actualidad la industria informática.

De eso se trata la **abstracción de datos**.

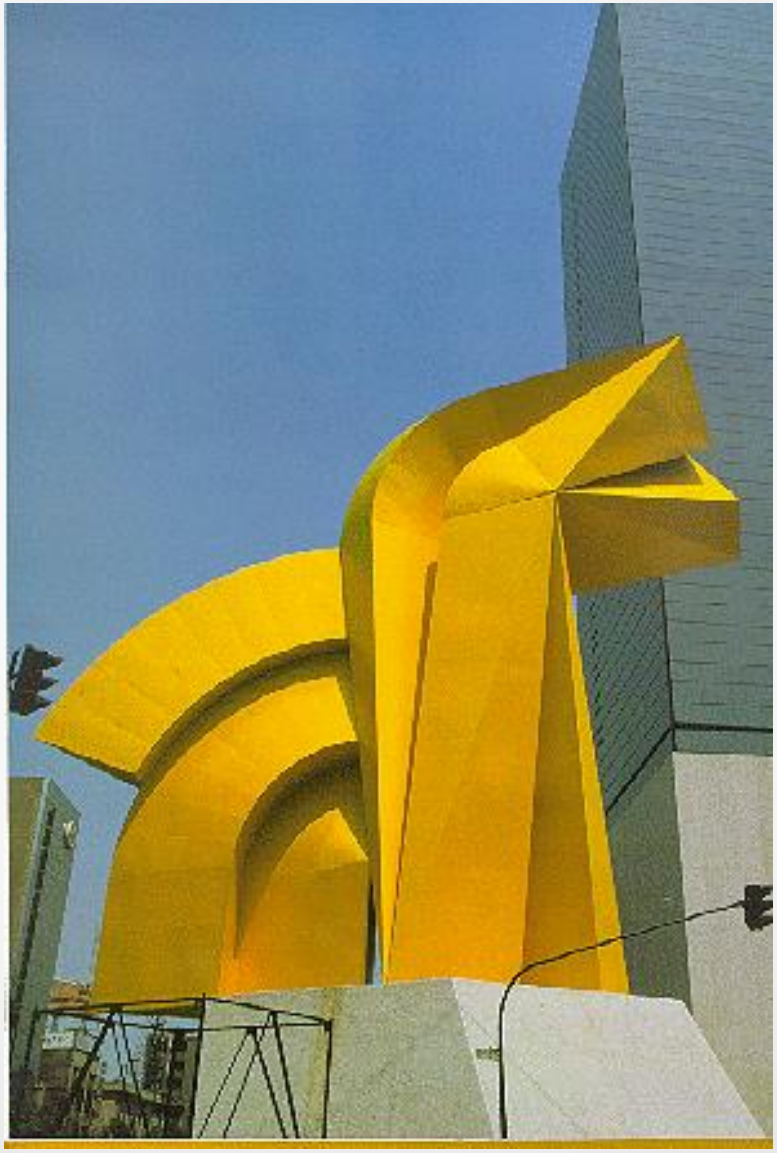
1.2 Tipos de Datos Abstractos

¿Qué es la **abstracción**?

Análisis de una cosa prescindiendo de los detalles de sus componentes.

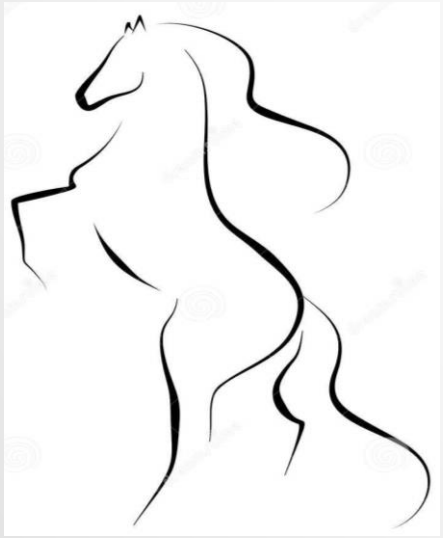
Ejemplo:

El caballito en la Ciudad de México



Sebastián

1.2 Tipos de Datos Abstractos

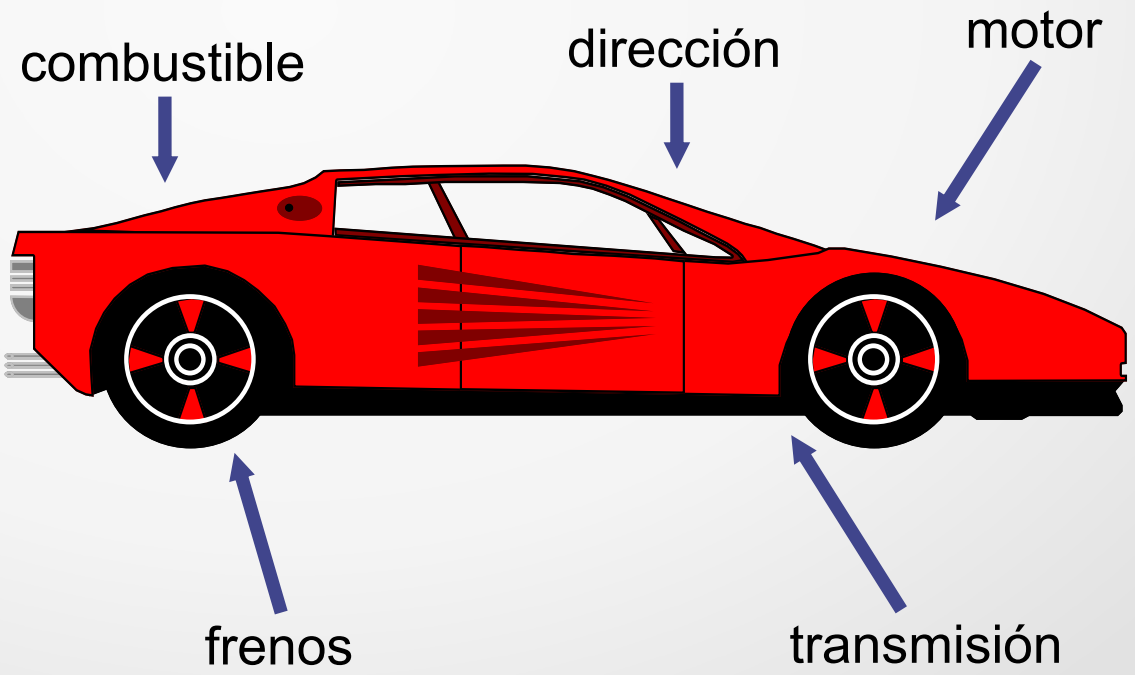


Arte
Abstracto

Arte Figurativo (**Realista**)

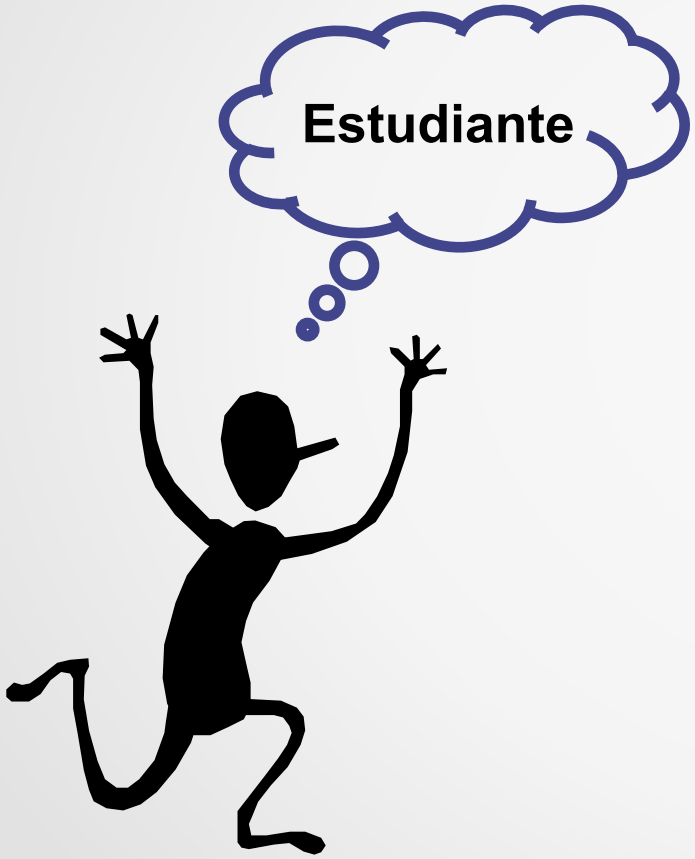
1.2 Tipos de Datos Abstractos

abstracción



1.2 Tipos de Datos Abstractos

abstracción



Número de Matrícula	Nombre	Carrera	Semestre que cursa

1.1 Tipos de Datos Abstractos

Tipo de dato abstracto es un concepto en el que los datos y su **comportamiento** se ven como una sola cosa.

Los TDA se crean por los programadores para usarse por si mismos u otros programadores ocultando detalles que complican la programación.

Los programadores usan los TDA para ignorar el contenido interno y solo se concentraran en su comportamiento (de esta forma **se simplifica la programación**).

1.2 Modularidad

La modularidad es un concepto de **programación** que permite, para efectos de claridad y eficiencia, **dividir** la solución de un problema grande **en módulos más pequeños**.

Para escribir un TDA es indispensable descomponer el problema en procesos más simples para lograr un nivel de abstracción adecuado.

1.2 Modularidad

La Programación Orientada a Objetos (POO), usa los conceptos de TDA y modularidad mediante el uso de **métodos** (procedimientos o funciones pertenecientes a una clase).

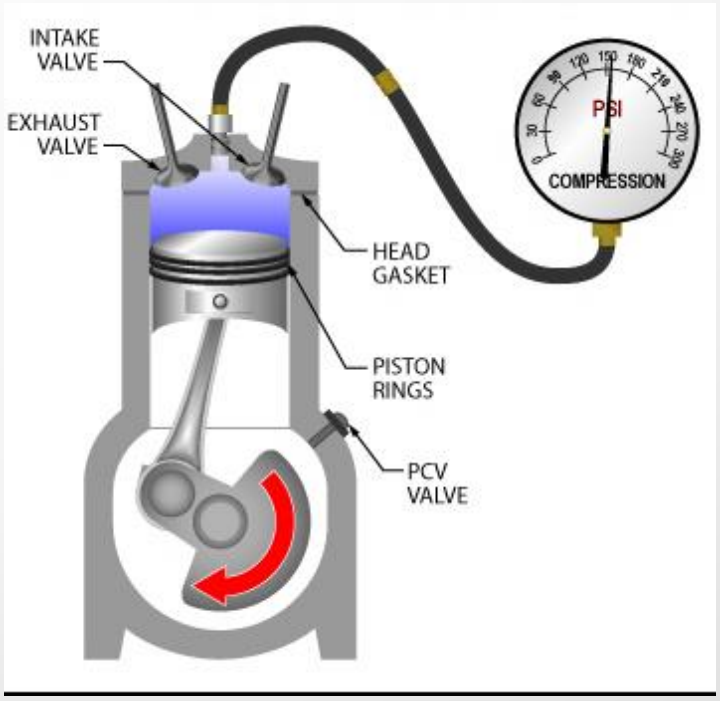
Una ventaja importante de la modularidad es la reutilización de código. Se busca que un método pueda ser invocado en cualquier momento desde otros métodos para evitar escribir las mismas líneas de código redundantemente.

1.3 Ejemplo de Tipos de Datos Abstractos

Ejercicio 1

Escribiremos un programa en C++, que resuelve el siguiente problema:

Un mecánico de automóviles necesita realizar varias pruebas de compresión a los cilindros de un automóvil.



1.3 Ejemplo de Tipos de Datos Abstractos

- A intervalos de pocos segundos, va a realizar lecturas de compresión sucesivas sin establecer de antemano la cantidad.
- El mecánico va a registrar una a una las diferentes lecturas del manómetro.
- La presión se registrará en PSI (libras por pulgada cuadrada, un valor numérico entero).
- Cuanto termine de tomar las lecturas, el mecánico lo indicará a la aplicación escribiendo un valor cero.
- El programa deberá reportar el primer valor y el último registrados, el número de lecturas realizadas, el valor promedio y el número de lecturas mayores al promedio.

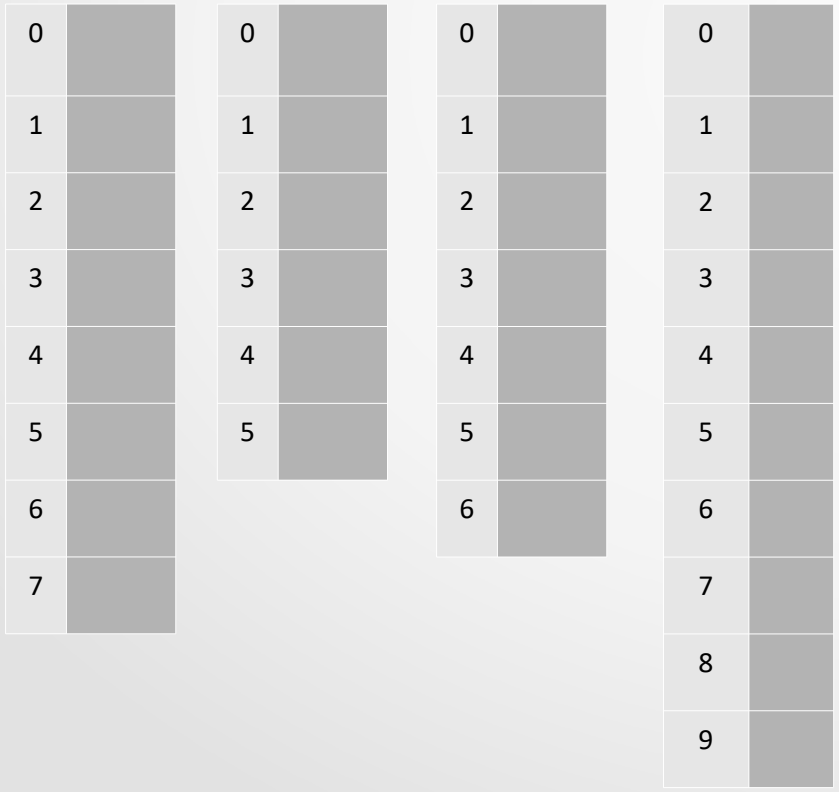
Descargue el compilador de C++ DevCpp de la URL siguiente:
https://www.felipealanis.org/Cursos/Estructuras de Datos/Unidad 1/
y pruebe este programa

```
#include <iostream>
using namespace std;
int main() {
    int datosGuardados[100];
    int lectura;
    int contador=0;
    cout<<"Escribe la lectura de compresion (0->200): ";cin >> lectura;
    while (lectura!=0) {
        datosGuardados[contador]=lectura;
        contador++;
        cout<<"Escribe la lectura de compresion (0->200): ";cin >> lectura;
    }
    cout << "Numero de Datos: " << contador << endl;
    cout << "Primera entrada: " << datosGuardados[0] << endl;
    cout << "Ultima entrada : " << datosGuardados[contador-1] << endl;
    float suma=0;
    for(int i=0;i<contador;i++)
        suma += datosGuardados[i];
    float promedio=suma/contador;
    cout << "Promedio: " << promedio << endl;
    int mayores=0;
    for(int i=0;i<contador;i++)
        if (datosGuardados[i]>promedio)
            mayores++;
    cout << "Numero de datos mayores al Promedio: " << mayores << endl;
    return 0;
}
```

Esta no es una solución usando el concepto de Tipos de Datos Abstractos. Este es un programa que usa un *array* como estructura de datos básica para guardar el conjunto de datos que se requiere.

1.3 Ejemplo de Tipos de Datos Abstractos

Una alternativa para resolver el problema anterior de manera más simple, se podría usar un vector que vaya creciendo de tamaño conforme se requiere, es decir, que se comporte de manera dinámica, como los *ArrayList* de Java (la memoria reservada por los vectores es estática, por eso, en el problema se declaró de tamaño 100 para que no falte espacio).



Memoria estática: Espacio reservado en la memoria durante la compilación; el espacio se libera al terminar la ejecución del programa. Si se necesitara añadir más datos a un vector, tendría que modificarse el programa y declarar un vector de mayor tamaño.

Memoria dinámica: Espacio reservado y liberado durante la ejecución del programa. Se usa la memoria exacta requerida.

Por el momento, no usaremos la capacidad de C++ para hacer un manejo dinámico de la memoria por lo que crearemos una **clase para simular un vector dinámico**.

Usaremos la abstracción (es decir la creación de un TDA) para para simplificar los programas escritos.

La clase se llamará: ***Vec*** y se usará de la manera que se indica en la página siguiente[1].

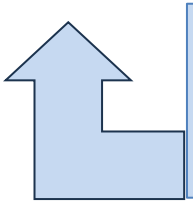
[1]Tampoco usaremos la clase *Vector* de C++ o alguna otra Clase de otro lenguaje. El objetivo es crear una clase propia para comprender el concepto de TDA.

Con la clase *Vec*, podremos declarar una variable llamada **x**:

Vec x;

Y podremos invocar a los métodos de esa clase, por ejemplo:

Métodos	**Propósito**
x.Anadir(50)	Añadir al Vector **x** un 50 al final
x.Insertar(45,3)	Insertar en **x** un 45 en la posición 3
x.Eliminar(5)	Eliminar de **x** el dato que se encuentra en la posición 5
cout << x.Traer(8)	Consultar el dato de la posición 8 guardado en **x**
cout << x.Tamanio()	Consultar el número de datos guardados en el vector **x**



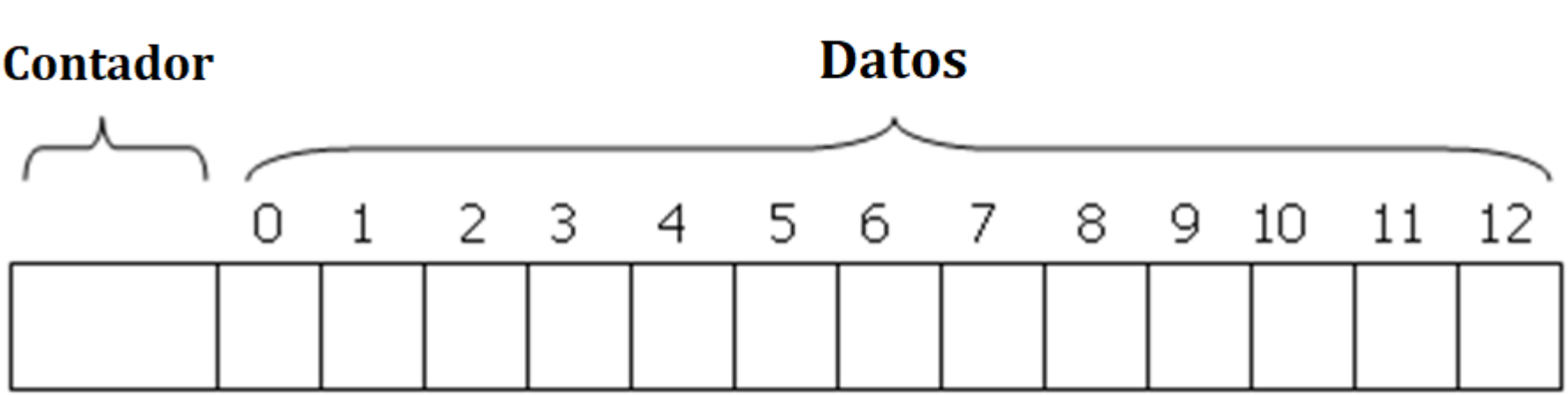
Con estos métodos se crea un nivel de abstracción importante que permite usar fácilmente la clase sin conocer los procesos internos que se llevan a cabo al interior de la clase.

1.3 Ejemplo de Tipos de Datos Abstractos

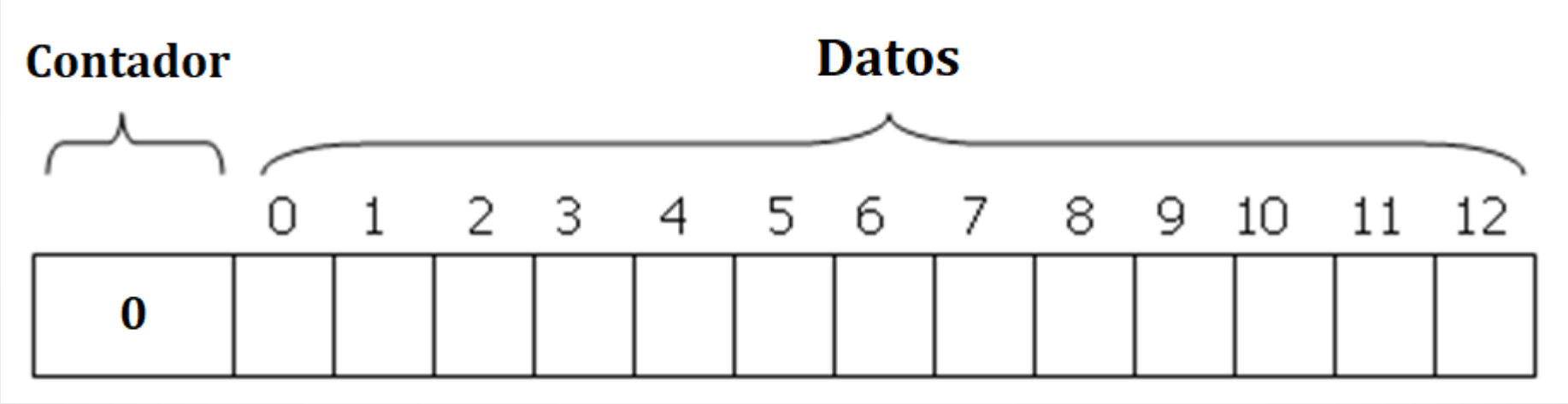
Los *arrays* son estáticos (durante la compilación se reserva en la memoria el espacio declarado) por lo que tendremos que simular que tenemos un array que crece o disminuye de tamaño según se requiera.

La estructura de la **Clase Vec** contendrá dos elementos:

- Un array llamado ***"Datos"*** para guardar los diferentes valores.
- Una variable entera llamada ***"Contador"*** que indicará el numero de lugares realmente usados del array (13 es la capacidad máxima en este ejemplo pero solo se usaran las posiciones indicadas por la variable *Contador*).



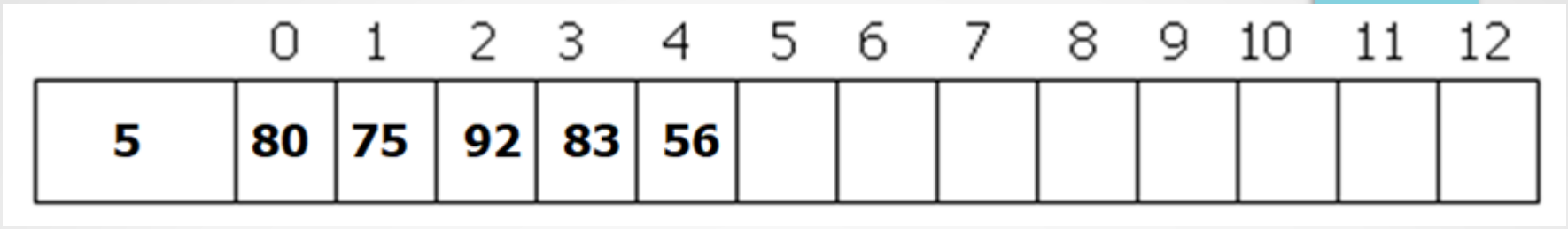
1.3 Ejemplo de Tipos de Datos Abstractos



Vec::Constructor()

Contador ← 0

1.3 Ejemplo de Tipos de Datos Abstractos

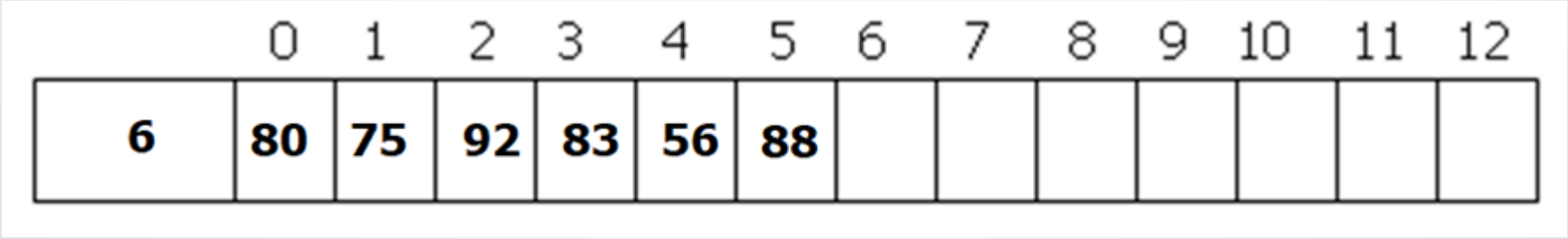


Anadir(88)

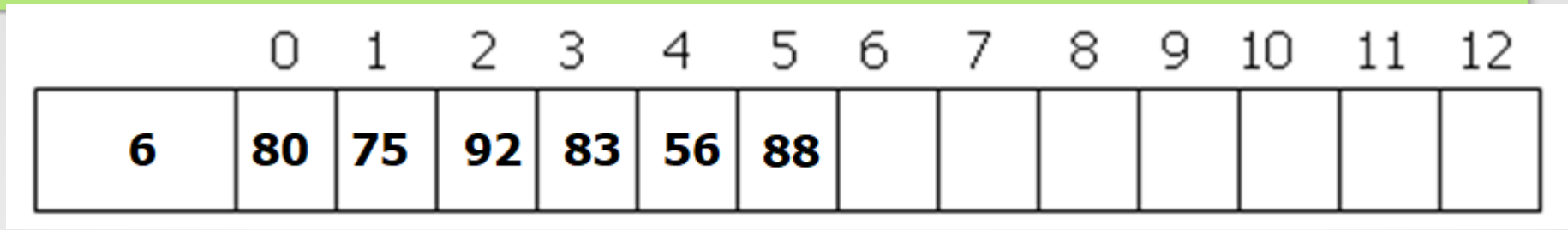
Vec::Anadir(nuevoDato)

Datos[Contador] ← nuevoDato

Contador ← Contador + 1



1.3 Ejemplo de Tipos de Datos Abstractos



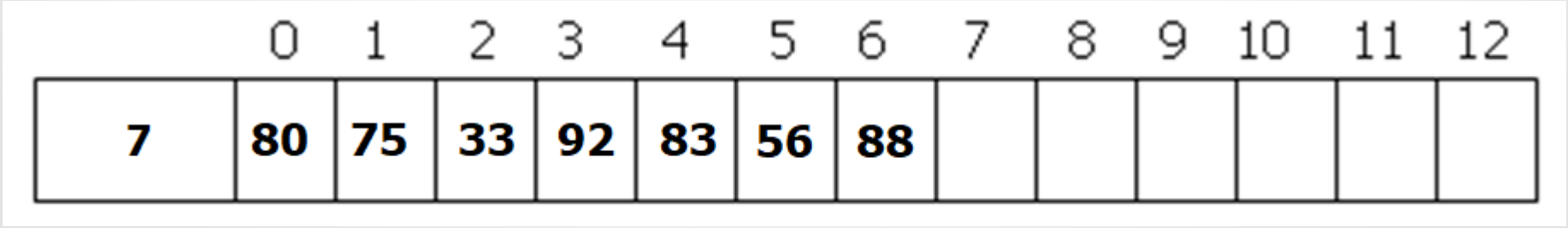
Insertar(33,2)

Vec::Insertar(nuevoDato,posicion)

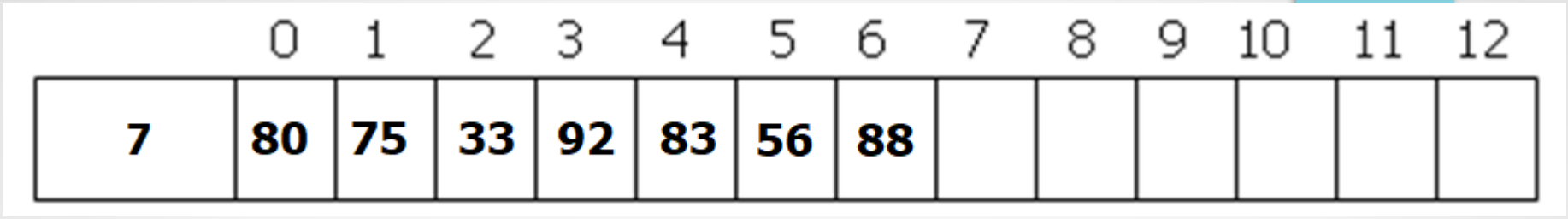
Copiar cada valor, entre las posiciones **Contador-1** y **posicion**, a la siguiente dirección. A este proceso se le llama *corrimiento inverso*.

Datos[posicion] ← nuevoDato

Contador ← Contador + 1



1.3 Ejemplo de Tipos de Datos Abstractos



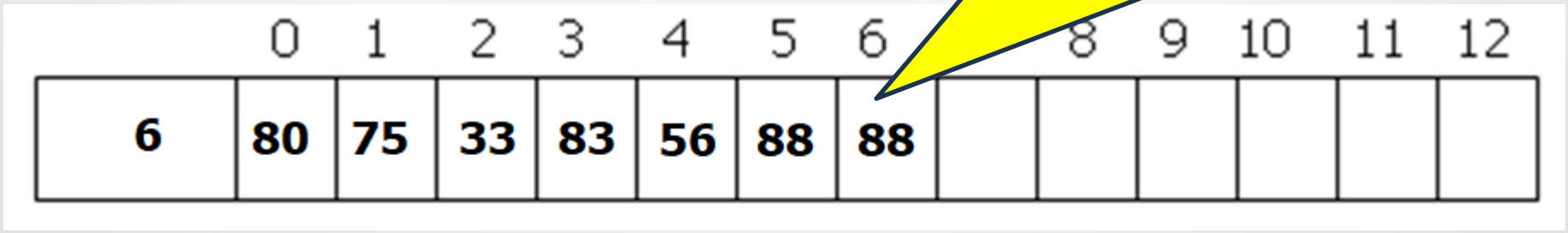
Eliminar(3)

Vec::Eliminar(posicion)

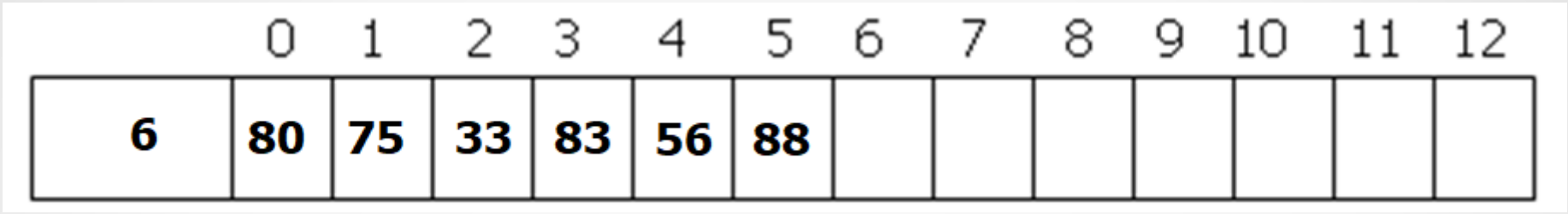
Copiar cada valor, entre las posiciones **posicion+1** y **Contador-1,** a la dirección anterior.

Contador ← Contador - 1

Ese valor no importa, se queda ahí (esto es lo que se llama *basura*), pero será reemplazado cuando ingrese un nuevo dato, observe que el numero de datos es 6 y se encuentran en las posiciones del 0 al 5



1.3 Ejemplo de Tipos de Datos Abstractos



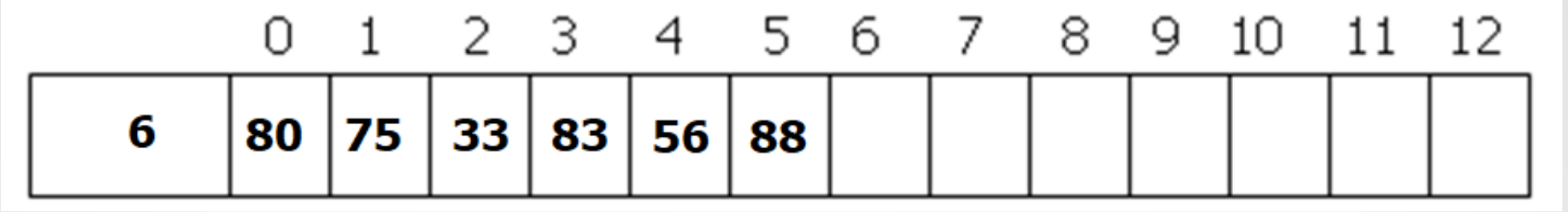
Traer(4)

Vec::Traer(posicion)

Regresa como resultado Datos[posicion]

Resultará el valor 56

1.3 Ejemplo de Tipos de Datos Abstractos



Tamanio()

Vec::Tamanio()

Regresa como resultado el valor de Contador

Resultará el valor 6

1.3 Ejemplo de Tipos de Datos Abstractos

Enseguida se crea la clase ***Vec*** y se usa para resolver el problema planteado anteriormente.

Solución usando una clase, es decir usando el concepto de TDA.

```
#include <iostream>
#include "ClaseVec.h"
using namespace std;
int main() {
    Vec Lecturas;
    int nuevaLectura;
    cout<<"Escribe la lectura de compresion (0->200): ";cin >> nuevaLectura;
    while (nuevaLectura!=0) {
        Lecturas.Anadir(nuevaLectura);
        cout<<"Escribe la lectura de compresion (0->200): ";cin >> nuevaLectura;
    }
    cout << "Primera entrada: " << Lecturas.Traer(0) << endl;
    cout << "Ultima entrada : " << Lecturas.Traer(Lecturas.Tamanio()-1) << endl;
    cout << "Numero de Datos: " << Lecturas.Tamanio() << endl;
    float suma=0;
    for(int i=0;i<Lecturas.Tamanio();i++)
        suma += Lecturas.Traer(i);
    float promedio=suma/Lecturas.Tamanio();
    cout << "Promedio                : " << promedio << endl;
    int mayores=0;
    for(int i=0;i<Lecturas.Tamanio();i++)
        if (Lecturas.Traer(i)>promedio)
            mayores++;
    cout << "Numero de Datos mayores al Promedio: " << mayores << endl;
    return 0;
}
```

Este es un archivo de cabecera que contiene la definición e la clase Vec. En la diapositiva siguiente se muestra su contenido

Archivo de cabecera ClaseVec.h

```
#define MaxVector 100
class Vec{
      private:
        int contador;
        int Datos[MaxVector];
      public:
        Vec();
        void Anadir(int);
        void Insertar(int,int);
        void Eliminar(int);
        int Traer(int);
        int Tamanio();
    };
Vec::Vec(){
   contador=0;
};
void Vec::Anadir(int nuevoDato){
    Datos[contador]=nuevoDato;
    contador++;
}
void Vec::Insertar(int nuevoDato,int posicion){
    for (int i=contador-1;i>=posicion;i--)
           Datos[i+1] = Datos[i];
    Datos[posicion]=nuevoDato;
    contador++;
}
void Vec::Eliminar(int posicion){
    for (int i=posicion+1;i<=contador-1;i++)
           Datos[i-1] = Datos[i];
    contador--;
}
int Vec::Traer(int posicion){
    return Datos[posicion];
}
int Vec::Tamanio(){
    return contador;
}
```

Los métodos Insertar y Eliminar no se usan para la aplicación de las lecturas de compresión pero se incluyen en la *Clase* porque pueden ser útiles para otras aplicaciones.

Ventaja principal de los TDA

La estructura interna de la Clase **se puede modificar** para optimizar su funcionamiento pero **las aplicaciones no requerirán ningún cambio**.

- Por ejemplo, se puede cambiar la Clase para que los datos no se almacenen en un vector de tamaño fijo sino en una estructura dinámica que no se agote rápidamente y sobre todo que tenga reservado un espacio que tal vez no se usará (los vectores ocupan un espacio fijo, se usen o no).
- Una Clase se puede usar en muchas aplicaciones y si se cambia para optimizar su funcionamiento, **solo se cambia la Clase una vez**, si no usamos clases, todas las aplicaciones deberán modificarse cuando se requiera un cambio.

Encapsulamiento

- En la siguiente diapositiva se incorporan modificaciones a la clase *Vec*.
- Esas modificaciones no afectarán al programa de aplicación de las Lecturas de Compresión.
  - Use la nueva Clase y compruebe que no se afecta el programa de aplicación.

Archivo de cabecera ClaseVecVersion2.h

```
#define MaxVector 100
class Vec{
      private:
            int contador;
            int contadorDatosIngresados;
            int contadorDatosEliminados;
            int Datos[MaxVector];
      public:
            Vec();
            void Anadir(int);
            void Insertar(int,int);
            void Eliminar(int);
            int Traer(int);
            int Tamanio();
            int NumeroEliminados();
            int NumeroIngresados();
};

Vec::Vec(){
            contador=0;
            contadorDatosEliminados=0;
            contadorDatosIngresados=0;
};

void Vec::Anadir(int nuevoDato){
            Datos[contador]=nuevoDato;
            contador++;
            contadorDatosIngresados++;
}

void Vec::Insertar(int nuevoDato,int posicion){
            for (int i=contador-1;i>=posicion;i--)
                        Datos[i+1] = Datos[i];
            Datos[posicion]=nuevoDato;
            contador++;
            contadorDatosIngresados++;
}
```

Observe los cambios a la Clase

```
void Vec::Eliminar(int posicion){
            for (int i=posicion+1;i<=contador-1;i++)
                        Datos[i-1] = Datos[i];
            contador--;
            contadorDatosEliminados++;
}

int Vec::Traer(int posicion){
            return Datos[posicion];
}

int Vec::Tamanio(){
            return contador;
}

int Vec::NumeroEliminados(){
            return contadorDatosEliminados;
}

int Vec::NumeroIngresados(){
            return contadorDatosIngresados;
}
```

Modificación básica al programa de aplicación

```
#include <iostream>
#include "ClaseVecVersion2.h"
using namespace std;
int main() {
    Vec Lecturas;
    int nuevaLectura;
    cout<<"Escribe la lectura de compresion (0->200): ";cin >> nuevaLectura;
    while (nuevaLectura!=0) {
        Lecturas.Anadir(nuevaLectura);
        cout<<"Escribe la lectura de compresion (0->200): ";cin >> nuevaLectura;
    }
    cout << "Primera entrada: " << Lecturas.Traer(0) << endl;
    cout << "Ultima entrada : " << Lecturas.Traer(Lecturas.Tamanio()-1) << endl;
    cout << "Numero de Datos: " << Lecturas.Tamanio() << endl;
    float suma=0;
    for(int i=0;i<Lecturas.Tamanio();i++)
        suma += Lecturas.Traer(i);
    float promedio=suma/Lecturas.Tamanio();
    cout << "Promedio                    : " << promedio << endl;
    int mayores=0;
    for(int i=0;i<Lecturas.Tamanio();i++)
        if (Lecturas.Traer(i)>promedio)
            mayores++;
    cout << "Numero de Datos mayores al Promedio: " << mayores << endl;
    return 0;
}
```

Modificaciones opcionales al programa de aplicación para que cuente con la funcionalidad adicional de eliminar el último dato registrado

```
#include <iostream>
#include "ClaseVecVersion2.h"
using namespace std;
int main() {
    Vec Lecturas;
    int nuevaLectura;
    cout<<"Registrar lectura (0=Terminar -1=Borrar Ultimo): ";cin >> nuevaLectura;
    while (nuevaLectura!=0) {
        if (nuevaLectura!=-1)
                Lecturas.Anadir(nuevaLectura);
        else
                Lecturas.Eliminar(Lecturas.Tamanio()-1);
        cout<<"Registrar lectura (0=Terminar -1=Borrar Ultimo): ";cin >> nuevaLectura;
    }
    cout << "Primer Valor registrado : " << Lecturas.Traer(0) << endl;
    cout << "Ultimo Valor registrado : " << Lecturas.Traer( Lecturas.Tamanio()-1 ) << endl;
    cout << "Numero de Datos         : " << Lecturas.Tamanio() << endl;
    float suma=0;
    for(int i=0;i<Lecturas.Tamanio();i++)
        suma += Lecturas.Traer(i);
    float promedio=suma/Lecturas.Tamanio();
    cout << "Promedio                : " << promedio << endl;
    int mayores=0;
    for(int i=0;i<Lecturas.Tamanio();i++)
        if (Lecturas.Traer(i)>promedio)
                mayores++;
    cout << "Numero de Datos mayores al Promedio: " << mayores << endl;
    cout << "Numero de Datos ingresados: " << Lecturas.NumeroIngresados() << endl;
    cout << "Numero de Datos eliminados: " << Lecturas.NumeroEliminados() << endl;
    return 0;
}
```

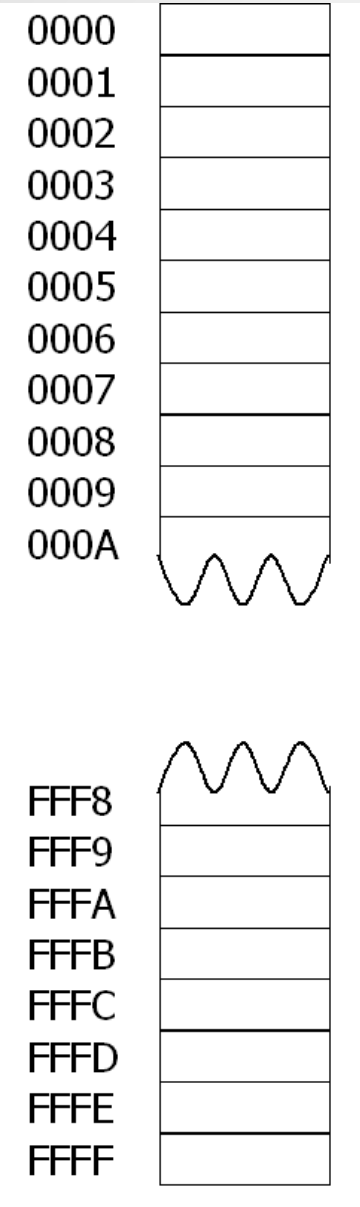
1.4 Manejo de Memoria Estática

```
// Pruebe con este programa C++
#include <iostream>
using namespace std;
int main()
{
    short x,y;
    char a[6];
    int j;
    cout << "Direcciones\n";
    cout << "de Memoria\n";
    cout << "x " << &x << en
    cout << "y " << &y <<  dl;
    cout << "a " << &a  << endl;
    cout << "j " << &j << endl;
    system("pause");
    return 0;
}
```

Declaración de variables

El *ampersand* (&) junto al nombre de la variable instruye al programa para obtener la dirección de memoria que le asignó el compilador (en vez del valor guardado en esa dirección)

1.4 Manejo de Memoria Estática



- La memoria se compone de lugares consecutivos.
- Para fines didácticos asumiremos que cada lugar es de un byte de capacidad.
- Cada byte tiene una dirección.
- Se acostumbra escribir la dirección en hexadecimal.

1.4 Manejo de Memoria Estática
Variables y arreglos

¿Cómo se distribuye el espacio en la memoria con una declaración como la del programa que acabamos de ejecutar?

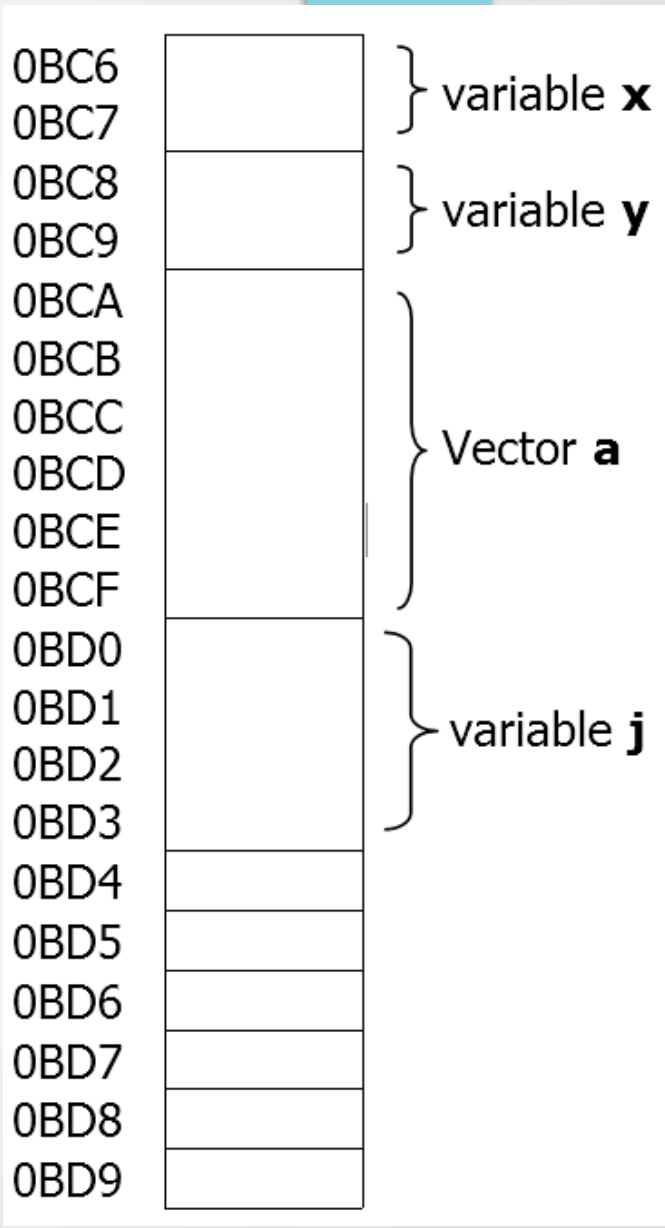
```
short x,y;
char a[6];
int j;
```

Respuesta:

Durante la compilación se reserva el espacio necesario para cada variable de acuerdo a su tipo.

Se asigna el espacio de acuerdo al orden de las variables dentro de la declaración.

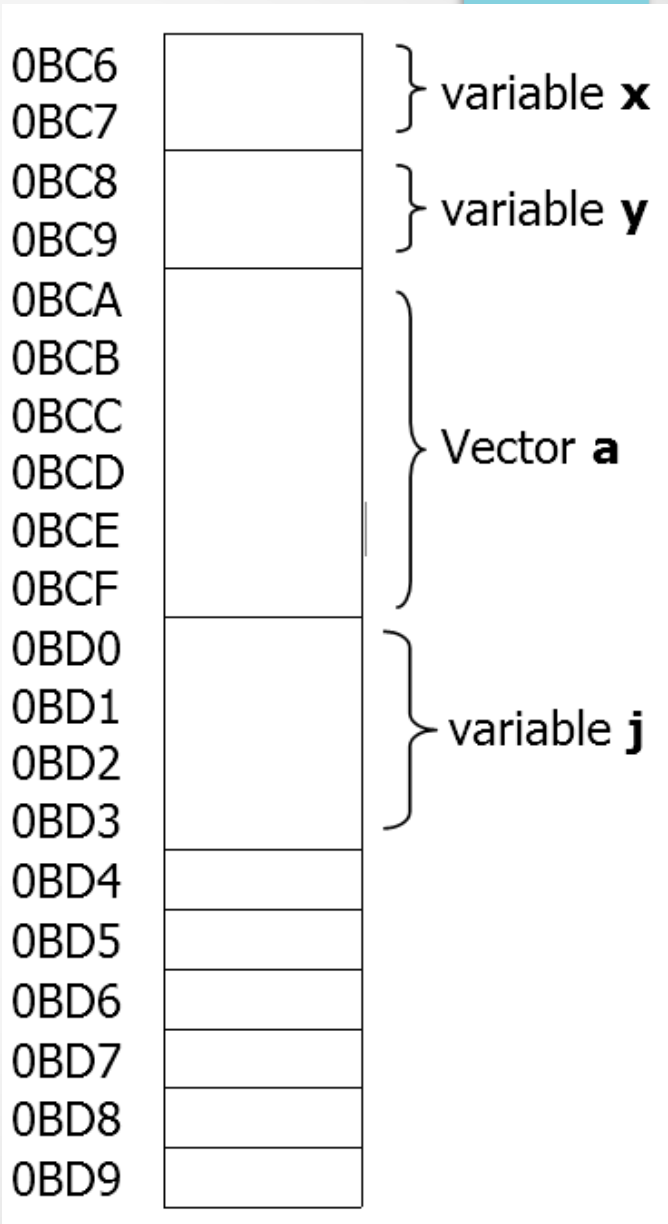
Observe que ***char***, aunque **ocupa 2 bytes** (UNICODE), en nuestro ejemplo para simplificar, lo consideramos como 1 byte (ASCII).



1.4 Manejo de Memoria Estática
Variables y arreglos

Resultados de la ejecución del programa:

```
Direcciones
de Memoria
x 0x6ffe1e
y 0x6ffe1c
a 0x6ffe10
j 0x6ffe0c
```



Observe que se asignan las direcciones de memoria a la inversa por eficiencia del código del compilador (se usa una estructura llamada **PILA**, que estudiaremos posteriormente)

1.4 Manejo de Memoria Estática

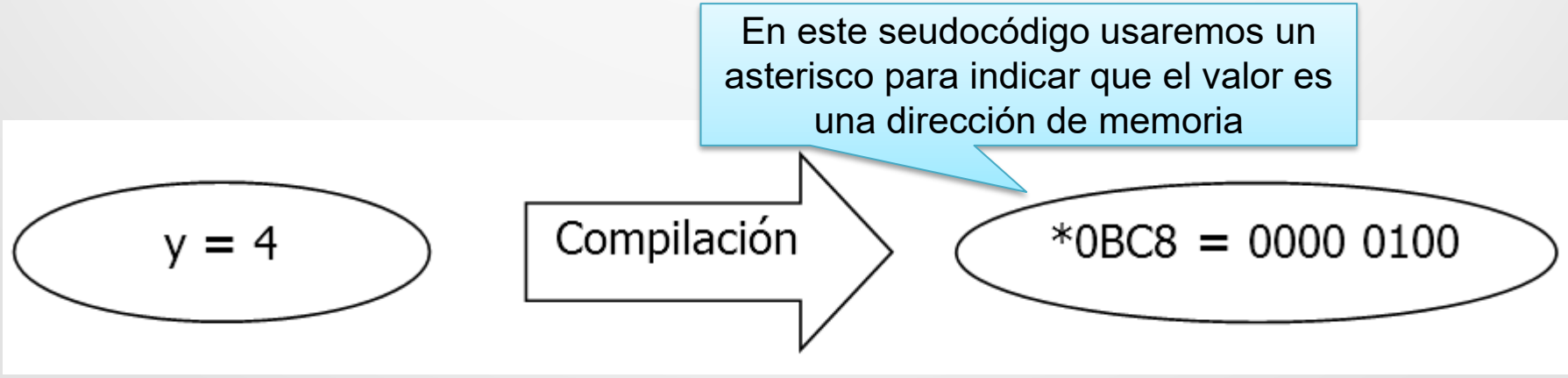
Durante el proceso de producción del código máquina, el compilador crea una tabla, de uso generalmente temporal para sustituir los nombres de variables y otros elementos por direcciones de memoria. Esta tabla se llama ***tabla de símbolos***.

Nombre de la variable	Tipo	Dirección inicial
x	short	0BC6
y	short	0BC8
a	char[0..5]	0BCA
j	int	0BD0

1.4 Manejo de Memoria Estática

Nombre de la variable	Tipo	Dirección inicial
x	short	0BC6
y	short	0BC8
a	char[0..5]	0BCA
j	int	0BD0

De acuerdo a la tabla de símbolos, ¿cómo quedaría en código objeto (al reemplazarse las variables por las direcciones de memoria) la instrucción **y = 4;**?



1.4 Manejo de Memoria Estática

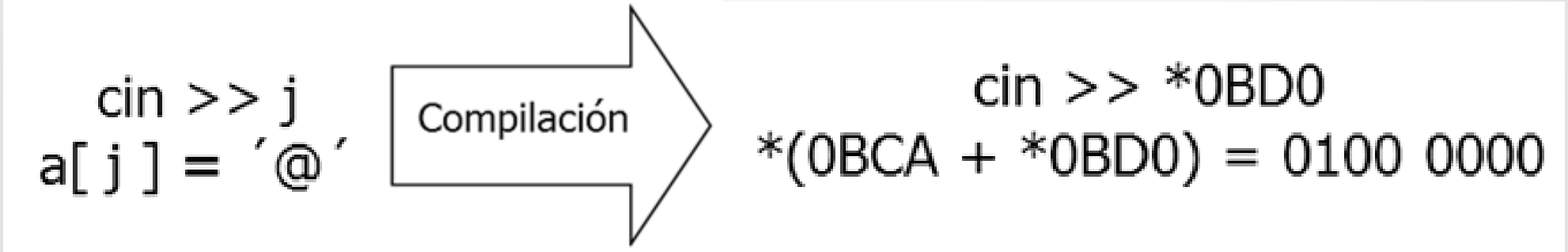
Nombre de la variable	Tipo	Dirección inicial
x	short	0BC6
y	short	0BC8
a	char[0..5]	0BCA
j	int	0BD0

¿Cómo quedará el código compilado si se asigna un "*" en la posición 2 del vector ***a***?

1.4 Manejo de Memoria Estática

Nombre de la variable	Tipo	Dirección inicial
x	short	0BC6
y	short	0BC8
a	char[0..5]	0BCA
j	int	0BD0

¿Cómo quedará si no se conoce, DURANTE LA COMPILACIÓN, en que posición del vector irá un dato sino hasta que se ejecute el programa?



1.4 Manejo de Memoria Estática

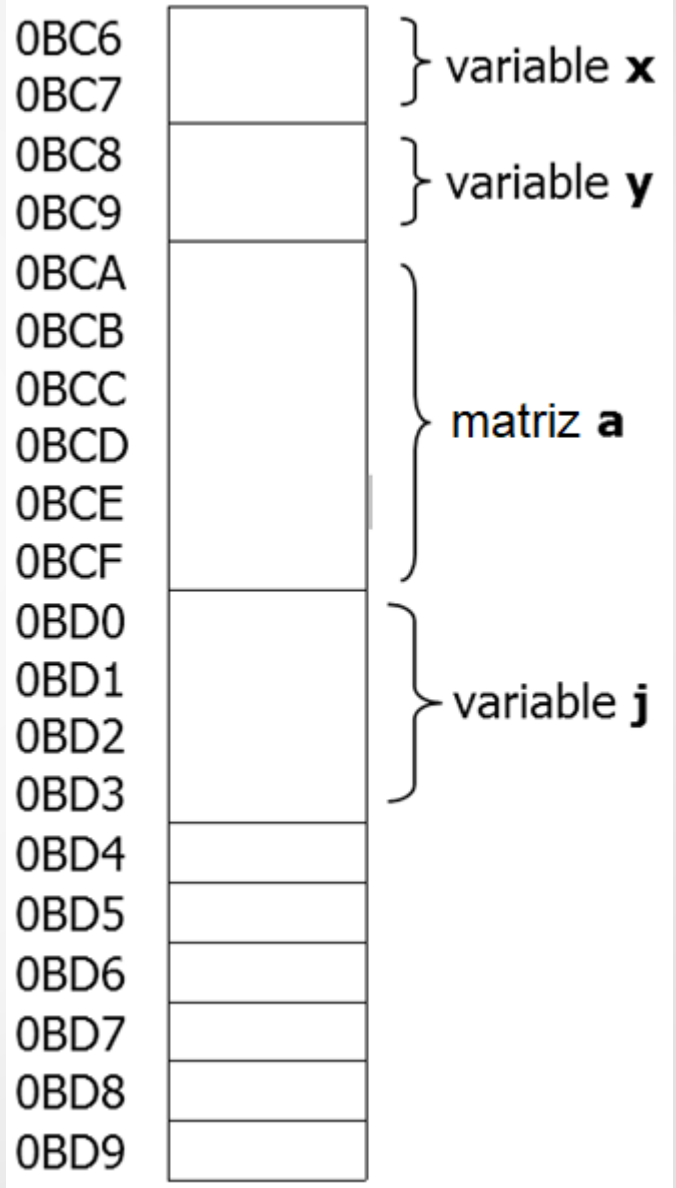
Considerar ahora que la declaración de variables es como la siguiente:

```
short x,y;
char a[3][2];
int j;
```

¿Como se guardan los elementos de la matriz a en la memoria?

```
cin>>i   cin>>j
 a[ i ][ j ] = '*'
```

Compilación



1.4 Manejo de Memoria Estática

Antes de responder a la pregunta anterior, hay que escribir un programa que use una **matriz de 4x5**. El programa debe contener un menú como el siguiente:

1.- Guardar un valor en la matriz.
2.- Consultar cierta posición de la matriz.
0.- Terminar.

Guardar preguntará al usuario el valor a guardar y el renglón y columna donde se guardará.
Luego regresa al menú de nuevo.

Consultar preguntará el renglón y columna que se desea consultar e informará el valor contenido en ese lugar.
Luego regresa al menú de nuevo.

1.4 Manejo de Memoria Estática

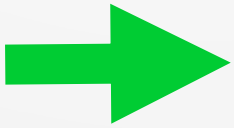
El objetivo del programa anterior es iniciar el razonamiento sobre la forma en la que el compilador esconde la complejidad del proceso al calcular la dirección equivalente unidimensional a partir de una bidimensional.

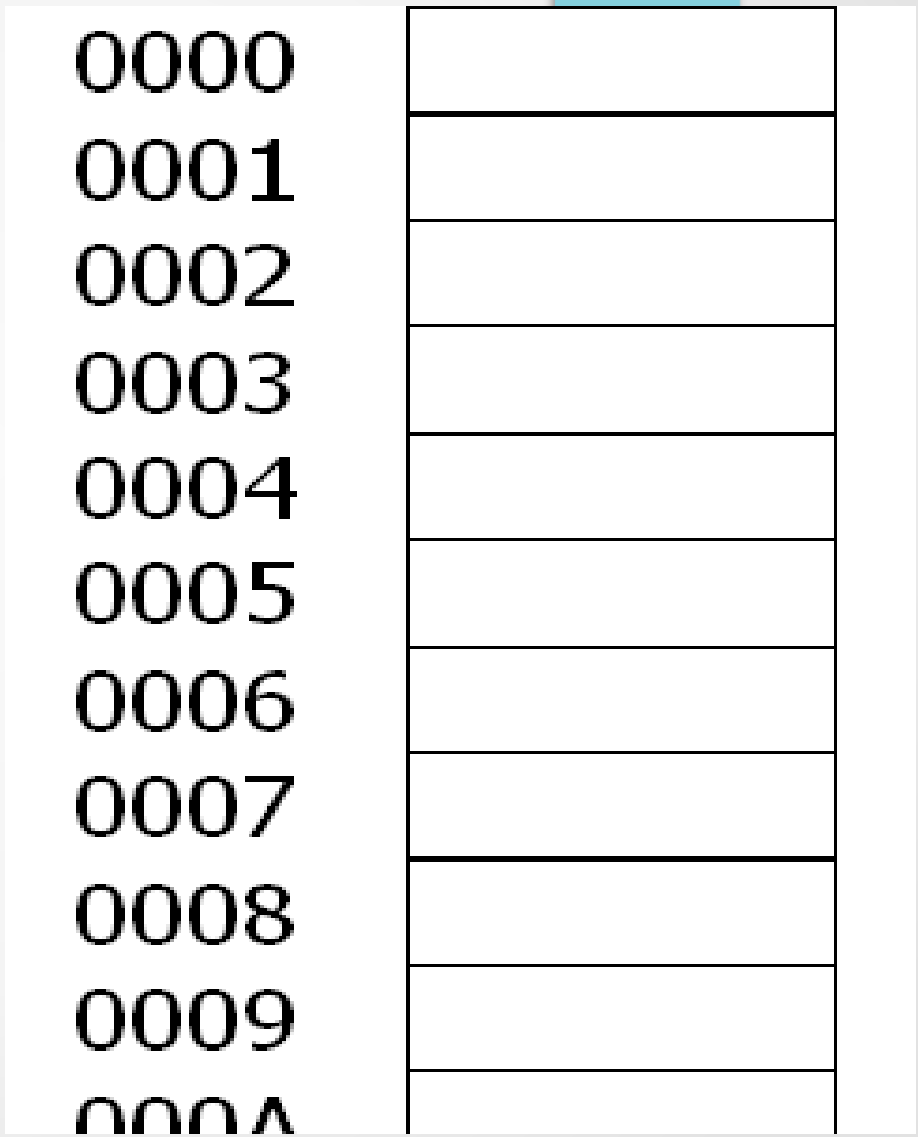
El compilador lo hace por los programadores, ahora nos toca simular ese proceso.

1.4 Manejo de Memoria Estática

La memoria es unidimensional (como un vector). Si en un lenguaje de programación se declara una matriz, el compilador contribuye aparentando que los datos se guardan en una matriz cuando en realidad se guardan en un vector (la memoria).

	0	1	2	3
0	A_{00}	A_{01}	A_{02}	A_{03}
1	A_{10}	A_{11}	A_{12}	A_{13}
2	A_{20}	A_{21}	A_{22}	A_{23}





1.4 Manejo de Memoria Estática

Ya que la memoria es unidimensional, haremos el razonamiento que resuelva el problema planteado.

Definiremos una metodología que sirva de base para escribir un programa para **guardar los datos de lo que aparentemente es una matriz en un vector**.

1.4 Manejo de Memoria Estática

	0	1	2	3
0	A_{00}	A_{01}	A_{02}	A_{03}
1	A_{10}	A_{11}	A_{12}	A_{13}
2	A_{20}	A_{21}	A_{22}	A_{23}

0	1	2	3	4	5	6	7	8	9	10	11
A_{00}	A_{01}	A_{02}	A_{03}	A_{10}	A_{11}	A_{12}	A_{13}	A_{20}	A_{21}	A_{22}	A_{23}

1er renglón 2º renglón 3er renglón

1.4 Manejo de Memoria Estática

Nomenclatura:

i, j Dirección bidimensional de cualquiera de los elementos de la matriz.

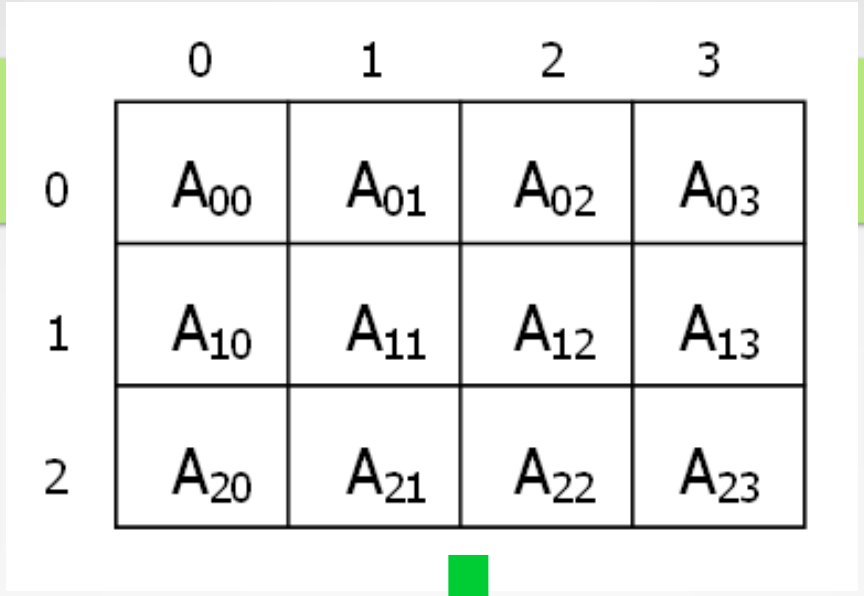
m Número total de renglones de la matriz.

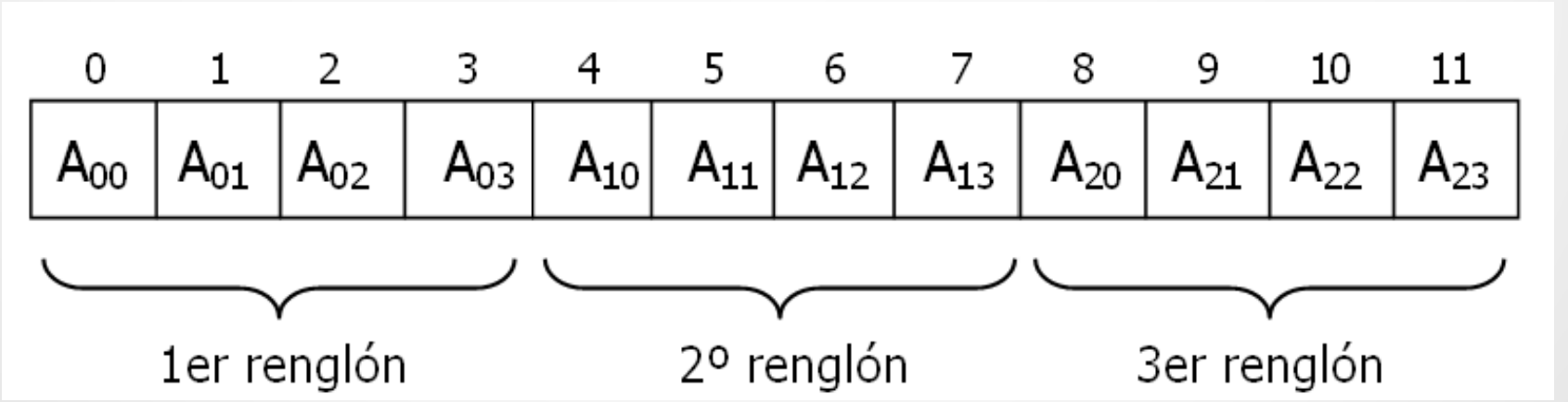
n Número total de columnas de la matriz.

p Dirección en el vector equivalente que le corresponderá a uno de los elementos de la matriz.

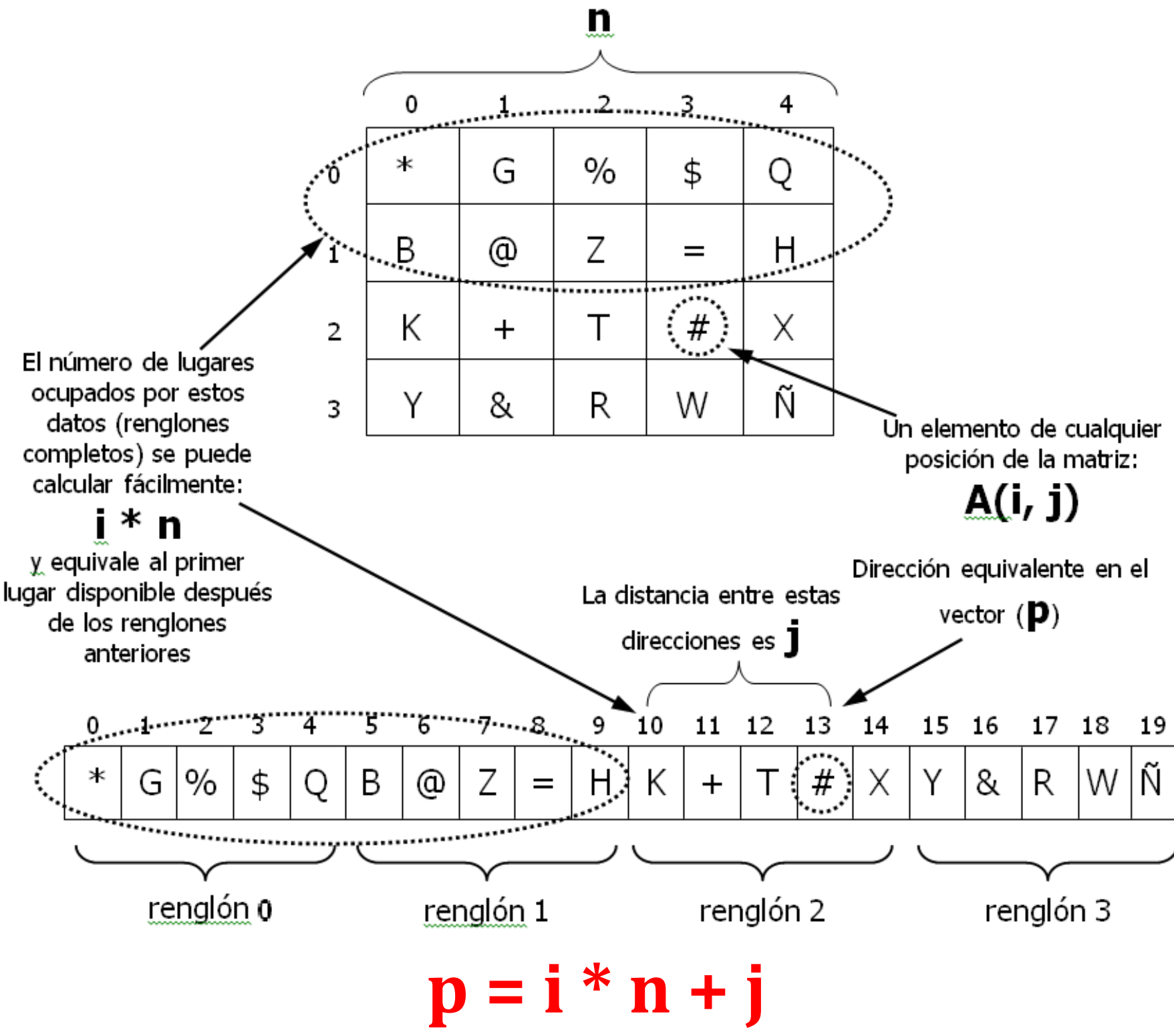
Se debe encontrar una fórmula para **p**, en función del estado de cualquier elemento de la matriz

$$\mathbf{P} = f(\mathbf{i, j, m, n})$$


0 1 2 3
0 A00 A01 A02 A03
1 A10 A11 A12 A13
2 A20 A21 A22 A23


0 1 2 3 4 5 6 7 8 9 10 11
A00 A01 A02 A03 A10 A11 A12 A13 A20 A21 A22 A23
1er renglón
2º renglón
3er renglón

$$p = i * n + j$$


n
0 1 2 3 4
0 * G % $ Q
1 B @ Z = H
2 K + T # X
3 Y & R W Ñ
El número de lugares ocupados por estos datos (renglones completos) se puede calcular fácilmente:
i * n
y equivale al primer lugar disponible después de los renglones anteriores
Un elemento de cualquier posición de la matriz:
A(i, j)
La distancia entre estas direcciones es j
Dirección equivalente en el vector (p)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
* G % $ Q B @ Z = H K + T # X Y & R W Ñ
renglón 0
renglón 1
renglón 2
renglón 3
p = i * n + j

1.4 Manejo de Memoria Estática

Debe modificar el programa anterior para declarar **un vector de 20 posiciones en vez de una matriz de 4x5** … **pero el funcionamiento del programa debe ser idéntico al anterior**:

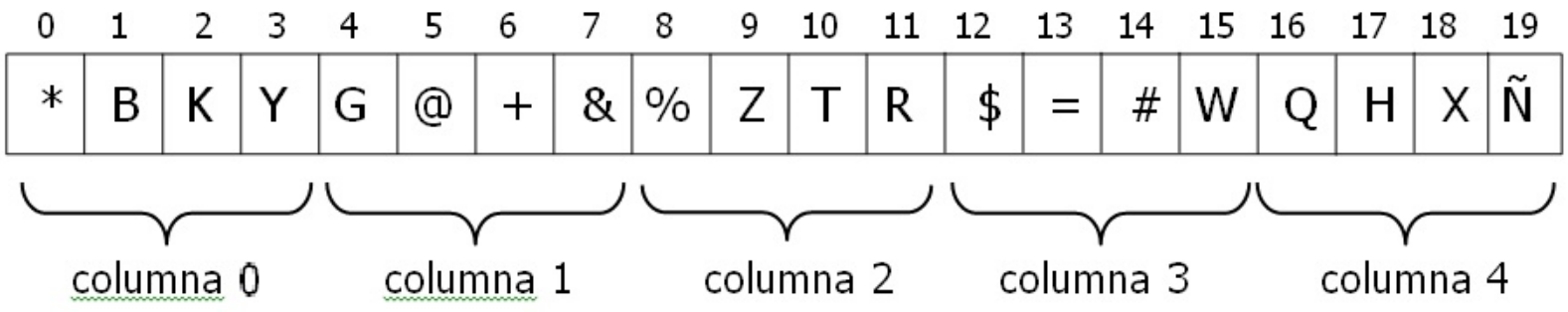
1.- Guardar un valor en la matriz.
2.- Consultar cierta posición de la matriz.
0.- Terminar.

Guardar debe preguntar la posición (renglón y columna) del lugar donde se va a guardar el valor y el valor mismo. *Consultar debe* preguntar el renglón y columna que se desea consultar e informa el valor que está guardado en ese lugar.

1.4 Manejo de Memoria Estática

Como ejercicio debe usted encontrar una fórmula para **p** siguiendo un razonamiento diferente (almacenando los elementos de la matriz **por columnas**)

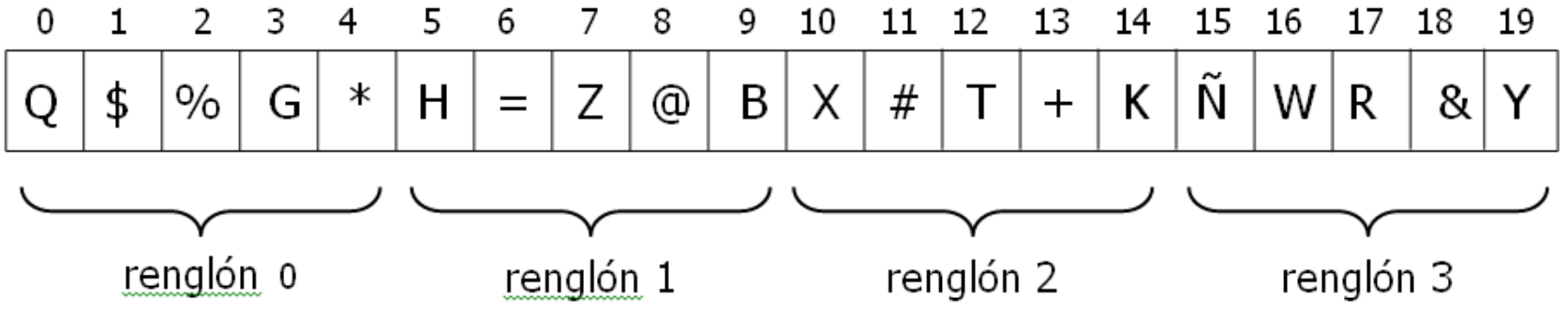
	0	1	2	3	4
0	*	G	%	$	Q
1	B	@	Z	=	H
2	K	+	T	#	X
3	Y	&	R	W	Ñ



1.4 Manejo de Memoria Estática

Encontrar una fórmula para **p** con base en este otro razonamiento: **por renglones *versión 2***)

	0	1	2	3	4
0	*	G	%	$	Q
1	B	@	Z	=	H
2	K	+	T	#	X
3	Y	&	R	W	Ñ



1.4 Manejo de Memoria Estática

Encontrar una fórmula para **p** con base en este otro razonamiento: **por renglones *version 3***)

	0	1	2	3	4
0	*	G	%	$	Q
1	B	@	Z	=	H
2	K	+	T	#	X
3	Y	&	R	W	Ñ

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Y	&	R	W	Ñ	K	+	T	#	X	B	@	Z	=	H	*	G	%	$	Q

renglón 3 renglón 2 renglón 1 renglón 0

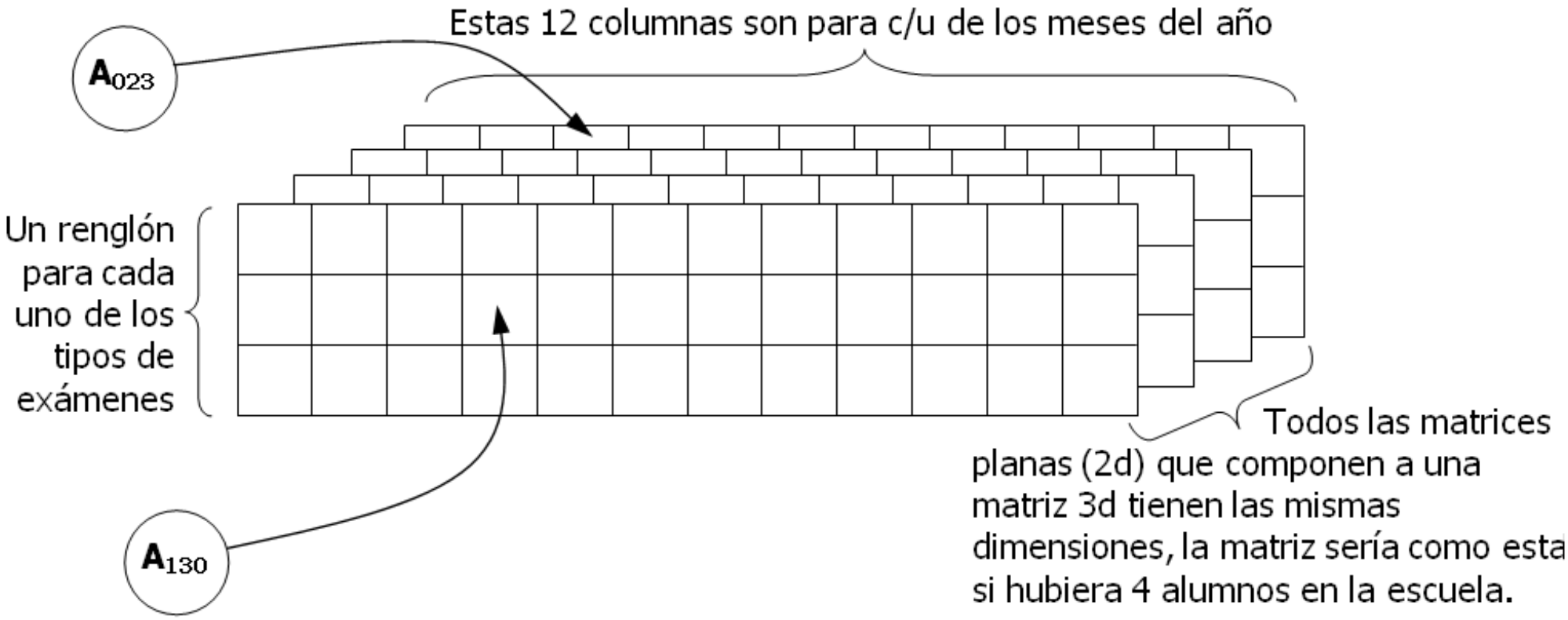
1.4 Manejo de Memoria Estática

Arreglos de tres o más dimensiones

Algunos problemas se resuelven más fácilmente usando arreglos tridimensionales, por ejemplo:

Se requiere guardar en una matriz los resultados **mensuales** de **todos los alumnos** de una escuela secundaria, obtenidos en **tres** **diferentes exámenes** (conocimientos, sicométrico y cultura general).

1.4 Manejo de Memoria Estática



La declaración sería:

char examenes[3][12][4];

1.4 Manejo de Memoria Estática

Fórmula para el Indice Equivalente

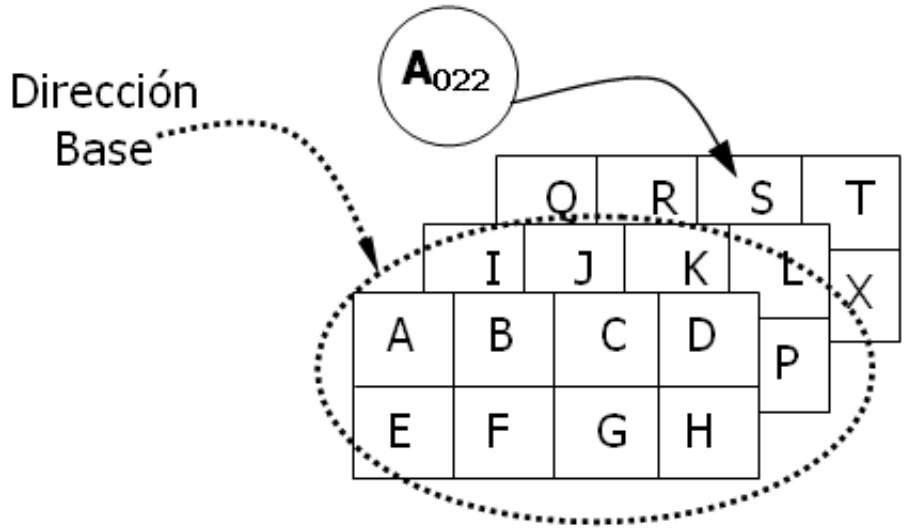
Se puede ver a los **arreglos tridimensionales** como una **proyección** de arreglos de dos dimensiones para facilitar la deducción de la fórmula.

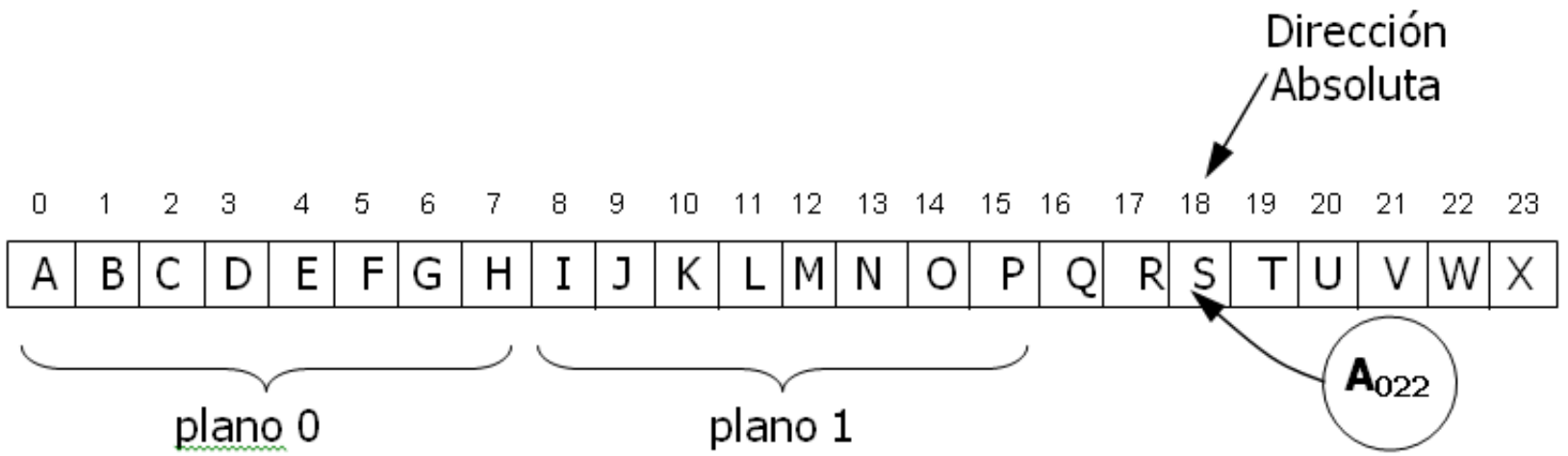
Cada elemento se identifica con una expresión de subíndices con tres componentes: $\mathbf{A}_{\mathbf{i\,j\,k}}$

i renglón donde se encuentra un elemento cualquiera.
j columna donde se encuentra un elemento cualquiera.
k plano (profundidad) donde se encuentra un elemento cualquiera.
m Número de renglones en total de cada plano.
n Número de columnas en total de cada plano.
o Número de planos del arreglo.
P Dirección en el vector equivalente

1.4

Apoyémonos usando un **arreglo 3d** más pequeño

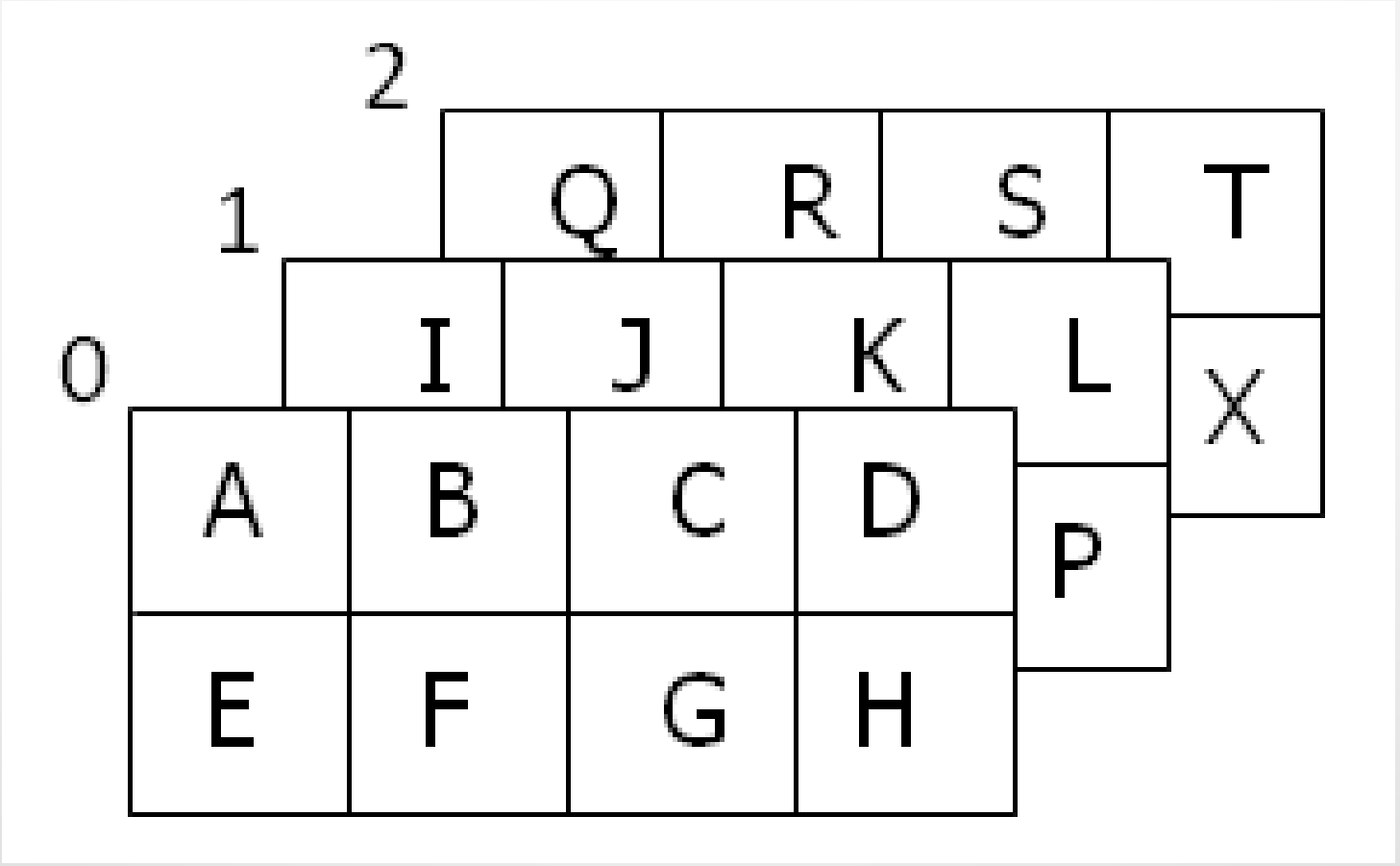




$$p = k\ m\ n + i\ n + j$$

1.4 Manejo de Memoria Estática

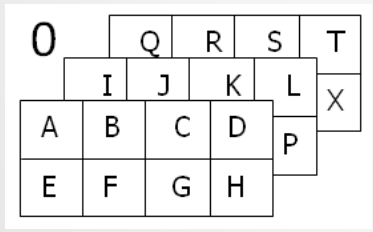
Matriz de tres dimensiones == Vector de matrices de dos dimensiones.

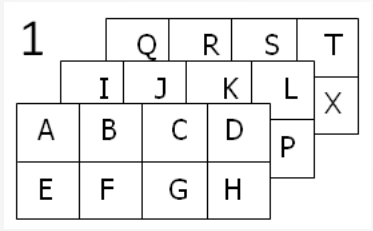


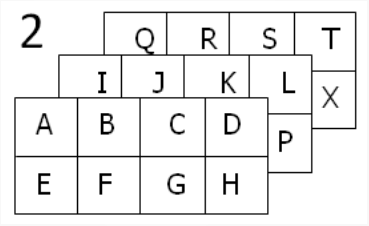
1.4 Manejo de Memoria Estática

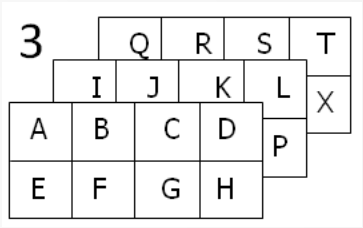
Arreglos de más de Tres Dimensiones

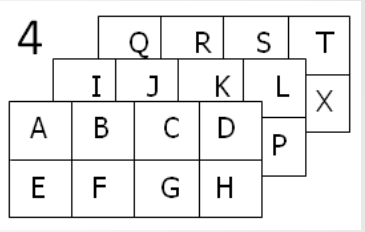
Matriz de **4** dimensiones = Vector de matrices de **3** dimensiones.











p = l*m*n*o + k*m*n + i*n + j

Siendo "l" la posición de cualquier elemento en la 4ª dimensión A(i,j,k,l).

Las fórmulas para los arreglos de más dimensiones son obtenidas por la serie correspondiente.

1.4 Manejo de Memoria Estática

Matrices poco densas regulares

10 personas participan en un juego de lanzamiento de dados en el cual se asigna un **número del 1 al 10** a cada una de ellas.

Cada jugador lanzará un **par de dados** tantas veces como indique el número que le haya tocado (el jugador **10 lanzará diez veces** y el **1 solo una vez**). Los resultados de cada lanzamiento deberán conservarse, para fines estadísticos, en una matriz en la cual los renglones representan a los jugadores y las columnas a cada lanzamiento particular.

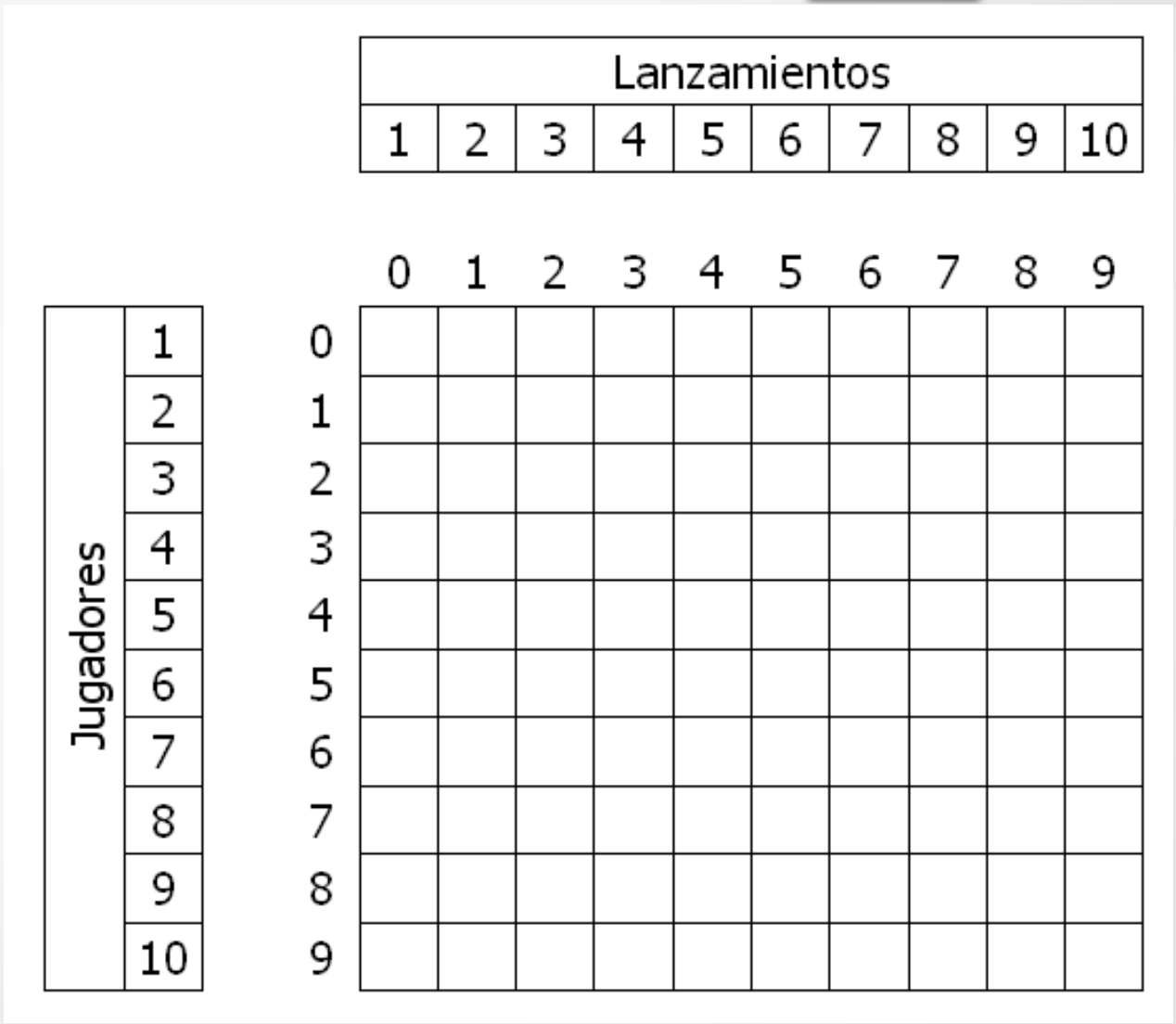
El jugador que obtenga el promedio más alto es el que ganará.

1.4 Manejo de Memoria Estática

Ejercicio 1

Escriba un programa que haga una simulación de éste juego imprimiendo, al menos, el resultado promedio de cada jugador.

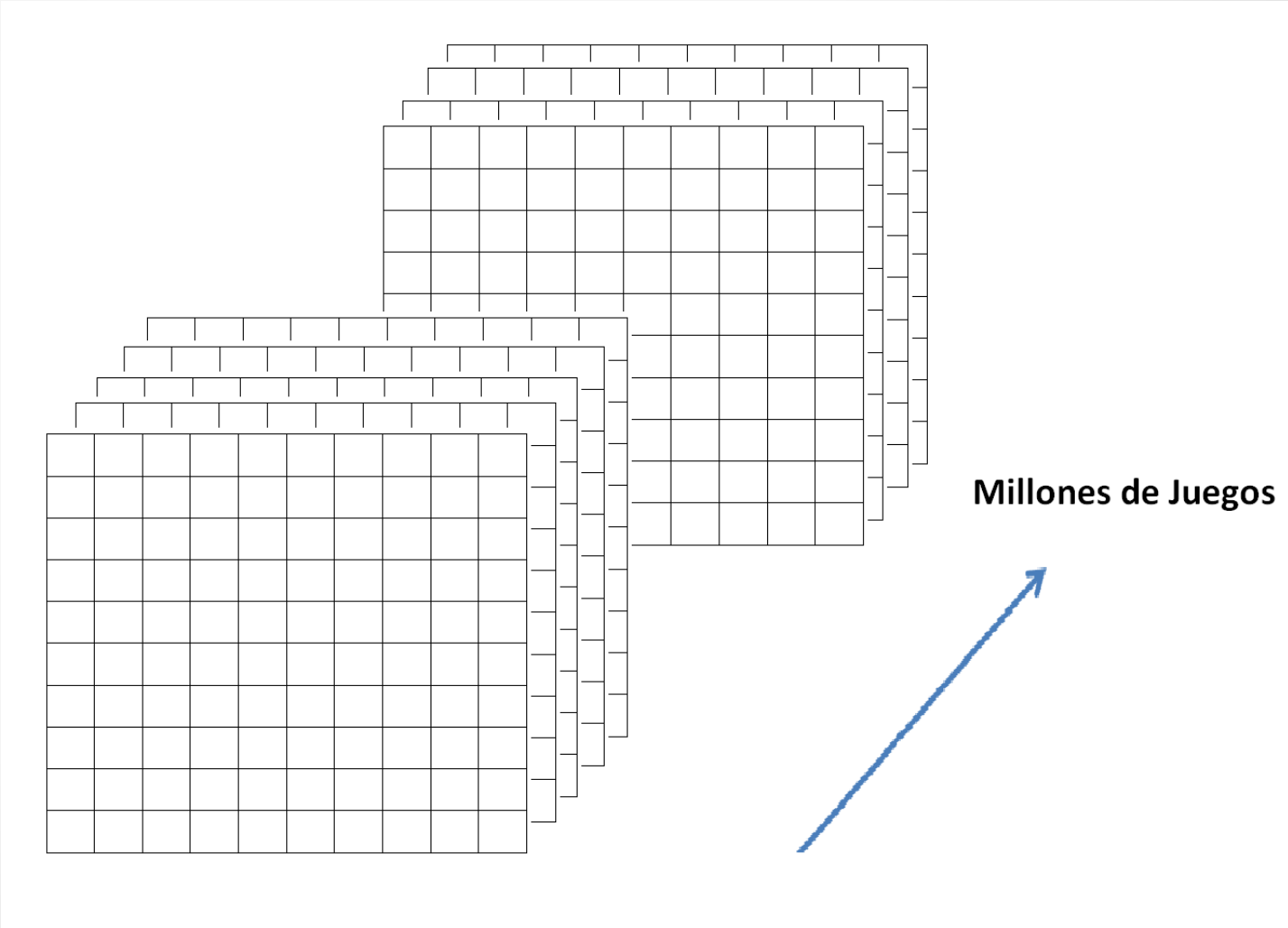
Utilice una matriz completa en la que los renglones representen a los jugadores y las columnas a los lanzamientos de los dados.



1.4 Manejo de Memoria Estática

Ejercicio 2

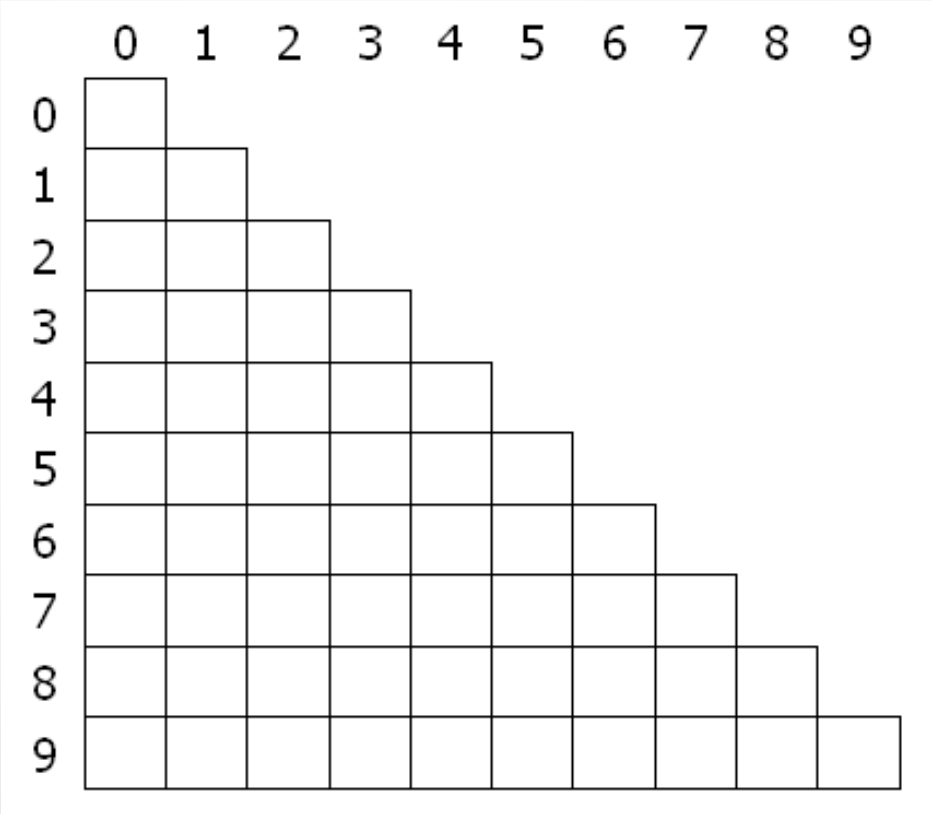
Modifique el programa para que se haga una simulación de cualquier número de juegos (hasta millones de ellos) y se conserven los resultados de todos en una matriz tridimensional.



1.4 Manejo de Memoria Estática

Matrices Triangulares Inferiores

- Son matrices cuadradas.
- Solo los datos de las direcciones **i >= j** son significativos.
- La matriz que se requiere para el ejercicio 1 es una **MTI** como la siguiente:



1.4 Manejo de Memoria Estática

Matrices Triangulares Inferiores

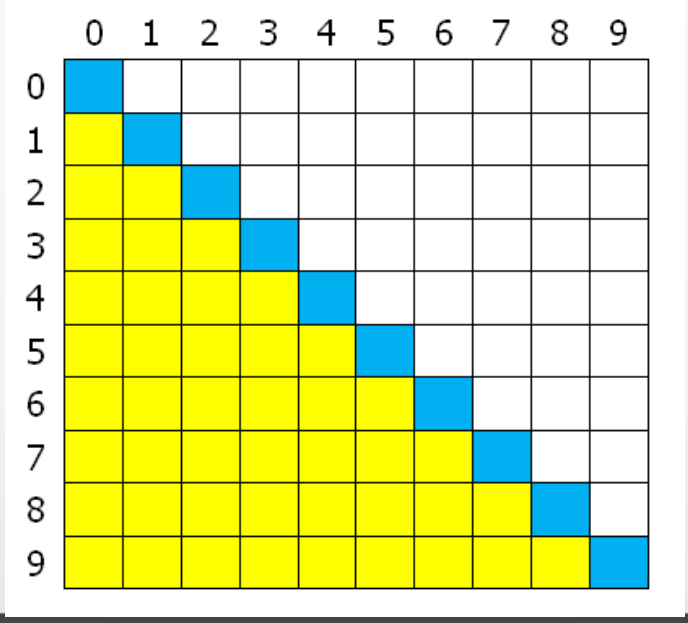
- Una **MTI** es **matriz poco densa** porque el número de elementos que contiene es pequeño con respecto a la matriz completa.
- Una **MTI** es una **matriz regular** porque se puede predecir su forma sin importar el tamaño.
- Si se requiere una **MTI de gran tamaño**, conviene guardar los datos en un **vector** para optimizar el uso de la memoria.

1.4 Manejo de Memoria Estática

Determinación del Tamaño del Arreglo Unidimensional
(para declarar un vector del tamaño justo)

Número de elementos en la diagonal principal es: **n**
Total de elementos en una matriz completa de orden n es: **n^2**

Número de lugares bajo la diagonal principal es: $\frac{n^2 - n}{2}$

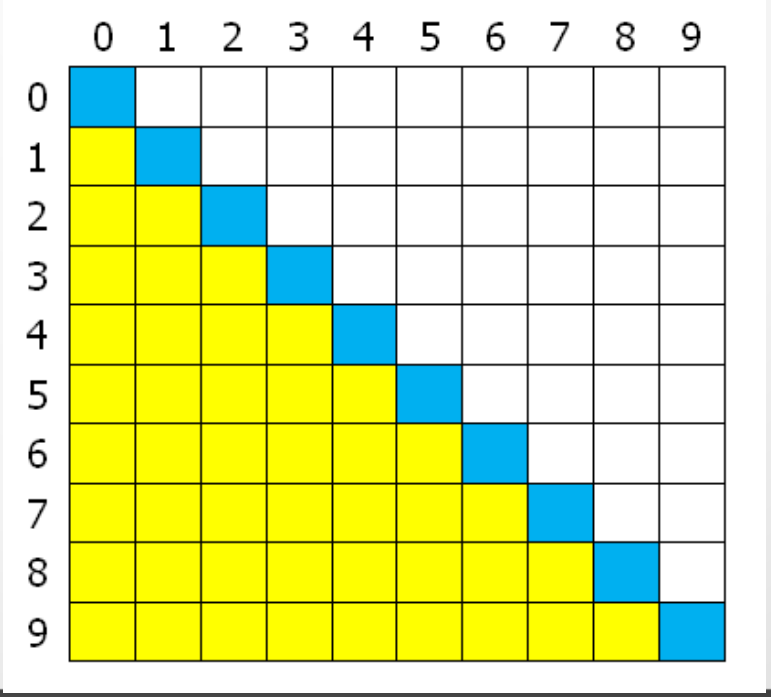


1.4 Manejo de Memoria Estática

Determinación del Tamaño del Arreglo Unidimensional

Número total de elementos en una **MT** es:

$$NTE = \frac{n^2 - n}{2} + n = \frac{n(n+1)}{2}$$



1.4 Manejo de Memoria Estática

Encontrar fórmula para el Indice Equivalente de una Matriz Triangular Inferior en un vector

	0	1	2	3	4
0	Z				
1	X	A			
2	C	S	F		
3	V	D	G	Q	
4	B	E	H	W	E

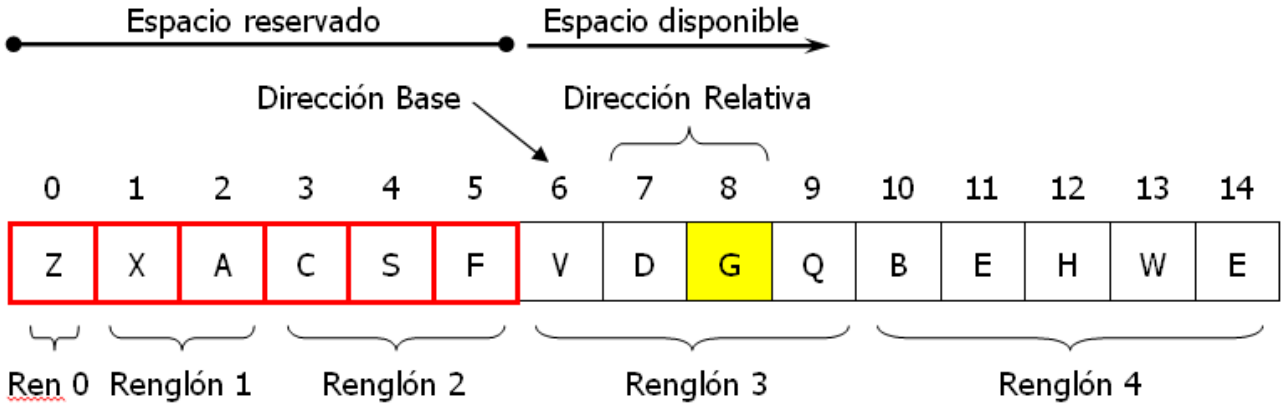
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Z	X	A	C	S	F	V	D	G	Q	B	E	H	W	E

Ren 0 Renglón 1 Renglón 2 Renglón 3 Renglón 4

1.4 Manejo de M

El elemento marcado con amarillo está en el renglón 3, y como se puede ver, encima de ese renglón hay una MTI de tamaño 3, por lo tanto, para cada elemento de la posición **i**, arriba hay una MTI de tamaño **i**, y como el número de elementos en una MTI es n(n+1)/2:

	0	1	2	3	4
0	Z				
1	X	A			
2	C	S	F		
3	V	D	G	Q	
4	B	E	H	W	E



Dirección Base, primer lugar disponible para usarse

$$DB = \frac{i\,(i+1)}{2}$$

La *Dirección Relativa* (distancia del lugar asignado desde la dirección base)

$$DR = j$$

$$p = \frac{i\,(i+1)}{2} + j$$

1.4 Manejo de Memoria Estática

Algunas aplicaciones requieren el uso de Matrices Triangulares Superiores.

Ejercicio 3

Encuentre dos razonamientos para el índice equivalente en el vector para una Matriz Triangular Superior.