1. Definición

Un arreglo (*array*) es un conjunto de datos **del mismo tipo** almacenados en una única variable a los cuales se puede acceder a través de uno o más índices, dependiendo de la dimensión del mismo. El arreglo tendrá tantos índices como dimensiones posea. Es una manera de tener indexados y/u ordenados los datos, agrupándolos por practicidad o necesidad, dependiendo el algoritmo a modelar. Los índices o posiciones del arreglo en `C++` siempre empiezan por el 0 (cero). El espacio que ocupan en la memoria es la suma de todos sus datos. Es decir, lo que ocupa el dato (`sizeof`) multiplicado la cantidad de datos indexados por el arreglo.

2. Unidimensionales (vectores)

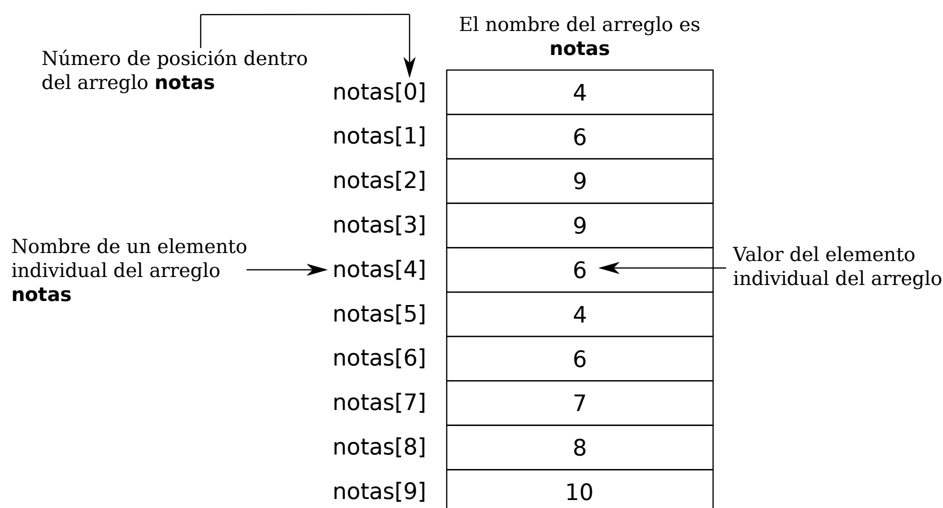
2.1. Declaración y acceso

Los arreglos más sencillos son los que poseen un único índice o dimensión. A estos suele denominárselos **vectores**. Por ejemplo, un arreglo de 10 enteros se lo declara de la siguiente manera:

```
int notas[10]; // notas posee 10 valores enteros
```

Como se observa, la cantidad de datos es especificada a la derecha del nombre de la variable entre *corchetes* `[]`. Si se deseara acceder a alguno de los valores indexados por la variable `notas` se debe utilizar el índice correspondiente (del 0 al 9):

```
notas[0] = 4;  // se almacena un 4 en la primera posición
notas[1] = 6;  // se almacena un 6 en la segunda posición
notas[2] = 9;  // se almacena un 9 en la tercera posición
// ...
notas[9] = 10; // se almacena un 10 en la última posición
```



Esto es tanto para almacenar un valor como para leerlo, por ejemplo, si se deseara mostrar alguno por pantalla se debe hacer:

Matías S. Ávalos© CC BY NC SA 4.0

```
cout << "La octaba nota es: " << notas[7] << endl;
```

2.1.1. Indexación por medio de variables

No es necesario que el número sea escrito literal por el programador, el arreglo puede estar indexado por una variable. Por ejemplo, si se deseara mostrar todas las `notas`:

```
for(int i=0; i < 10; i++) {
  cout << "La nota " << i+1 << " es: " << notas[i] << endl;
}
```

Es importante destacar que `i` **debe comenzar en 0** y terminar en el *valor correcto*, es decir, el último elemento del arreglo (en este caso 9). Ya que, de lo contrario, estaremos accediendo a una posición de memoria con contenido desconocido o que el sistema operativo rechace esa lectura terminando nuestro programa abruptamente con un error conocido como **"Violación de segmento"** (`SIGSEGV`[1]). La salida producida por este fragmento de código es la siguiente:

```
La nota 1 es: 4
La nota 2 es: 6
La nota 3 es: 9
...
La nota 10 es: 10
```

2.1.2. Referencias a arreglos

Podemos referirnos a un solo elemento del vector utilizando el operador `&` como ya se ha visto anteriormente:

```
int &nota2 = notas[2];
nota2 = 7;
cout << notas[2] << endl;
```

En el ejemplo anterior se sobrescribe el elemento 2 del arreglo `notas[2]` a través de la referencia `nota2`. Pero si se quisiera hacer una referencia al arreglo en general deberá agregarse unos paréntesis a la sintaxis como se muestra a continuación:

```
int (&ref_notas)[] = notas;
for(int i=0; i < 10; i++) {
  // Notas aleatorias entre 1 y 10
  ref_notas[i] = rand() % 10 + 1;
}
```

[1]https://es.wikipedia.org/wiki/SIGSEGV

Matías S. Ávalos© 4.0

2.1.3. foreach o for *mejorado*

Cabe destacar que en C++ existe un formato especial del *loop* for para recorrer vectores conocido en otros lenguajes como *foreach* o *enhanced for loop* y tiene la siguiente sintaxis:

```
for(int n : notas) { // para cada 'n' en notas:
  cout << "[" << n << "] ";
}
cout << endl;
```

En la variable n se irán copiando consecutivamente los valores almacenados en notas, se observa que no hace falta especificar el largo del arreglo, eso es deducido automáticamente. Una posible salida de este bucle es:

```
[4] [6] [9] [9] [6] [4] [6] [7] [8] [10]
```

La ventaja del *foreach* es que permite en una sintaxis muy abreviada recorrer todos los valores del vector, pero no es útil si se desea trabajar con los índices, ya que no tenemos la variable de control 'i' para saber en qué posición nos encontramos. Los algoritmos que requieran utilizar el índice del arreglo, como por ejemplo, los de ordenamiento que serán explicados más adelante requerirán del for *clásico*.

En este tipo de bucles, si en lugar de usar una simple variable, se utiliza una referencia se puede modificar el contenido del arreglo.

```
for(int &n : notas) { // para cada 'n' en 'notas':
  n = rand() % 10 + 1;
}
```

En la porción de código anterior la referencia 'n' apunta a una posición el arreglo 'notas' consecutivamente cada vez asignándole un número aleatorio entre 1 y 10 a cada una hasta finalizar el recorrido. Siempre usaremos referencias en los *foreach*, ya que es más eficiente usar una referencia a copiar los valores en una variable.

2.2. Inicialización

Los arreglos, al igual que las variables sencillas, pueden tener valores iniciales, pero debemos especificar los valores para cada posición. Esto se logra encerrando entre llaves {} y enumerando los valores para cada posición:

```
string materias[10] = {"Literatura", "Matemática","Ed. Físca","Fí
    sica", "Historia", "Geografía", "Programación","Inglés", "Alem
    án", "Biología"};
```

Es posible no inicializar todos los elementos de un arreglo. Por ejemplo, en un vector de enteros si se inicializa sólo algunas posiciones, el resto serán inicializadas en 0, es decir:

```
int notas[10] = {10, 4, 7};
for(int &nota : notas) {
  cout << "[" << nota << "]";
}
cout << endl;
```


Matías S. Ávalos© 4.0

Producirá la siguiente salida:

```
[10] [4] [7] [0] [0] [0] [0] [0] [0] [0]
```

Es posible recorrer más de un vector al mismo tiempo, en especial si estos están relacionados, como se observa en el Programa 1. Donde se vinculan los nombres de las materias con sus respectivas notas.

Programa 1: Boletín escolar

```
#include <iostream>

using namespace std;

/****************  Declaración de funciones  ****************/
int pedir_entero(string mensaje);
int pedir_entero_entre(string mensaje, int min, int max);

int main() {
  const int CANT_MATERIAS = 10;

  string materias[CANT_MATERIAS] = {"Literatura","Matemática",
                                    "Ed. Físca","Física","Historia",
                                    "Geografía","Programación",
                                    "Inglés","Alemán","Biología"};
  int notas[CANT_MATERIAS];

  // Se ingresan las notas:
  for(int i = 0; i < CANT_MATERIAS; i++) {
    cout << "\n* Materia: " << materias[i] << " *\n";
    notas[i]=pedir_entero_entre("Nota (del 1 al 10):", 1, 10);
  }
  // Se calcula el promedio:
  double promedio = 0;
  for (int &nota : notas) { // Para cada 'nota' en "notas"
    promedio += nota;       // Se acumulan las notas
  }
  promedio /= CANT_MATERIAS;
  cout << "\n\nEl promedio general es: " << promedio << endl;
}

/************* Implementación de las funciones *************/
int pedir_entero(string mensaje) {
  int result;
  string entrada;
  do {
    cout << mensaje;
    getline(cin, entrada);
    result = atoi(entrada.c_str());
  } while(result == 0 and entrada != "0");
  return result;
}
```


Matías S. Ávalos© 4.0

```
43
44 int pedir_entero_entre(string mensaje, int min, int max) {
45   int result;
46   if(min > max) {
47     int aux = max;
48     max = min;
49     min = aux;
50   }
51   do {
52     result = pedir_entero(mensaje);
53   } while(result < min or result > max);
54   return result;
55 }
```

2.2.1. Tamaño implícito de vectores

Si el vector es inicializado es posible no especificar la cantidad de elementos del mismo, este es inferido por la cantidad de valores presentes al momento de la inicialización:

```
// vector de 7 posiciones declarado implícitamente:
string dias_de_la_semana[] = {"lunes", "martes", "miércoles",
                              "jueves", "viernes",
                              "sábado", "domingo"};
```

En este caso, el compilador asignará automáticamente a la variable `diasDeLaSemana` el tamaño de 7 elementos.

2.2.2. Copiar vectores

No es posible copiar un arreglo en otro, por lo tanto si se desea una copia es necesario recorrer el vector original asignando elemento a elemento los valores:

```
int vec[10] = {1,2,3,4,5,6,7,8,9,10};
int cpv[10] = vec;   // ERROR!! Esto no compilará

for(int i=0; i < 10; i++) {
  cpv[i] = vec[i];   // Correcto
}
```

Esto ocurre porque '`vec`' y '`cpv`' son arreglos, y lo que almacenan son las direcciones de memorias del primer elemento, que luego accedemos a través de los '`[]`'. Por lo tanto, al declararlos esto serán diferentes e inmutables.

2.3. Arreglos como parámetros de una función

Para que una función reciba un arreglo debemos declararlo como tal, es una buena práctica no hacer un algoritmo para una cantidad específica de elementos, sino que estos también

Matías S. Ávalos© CC BY NC SA 4.0

sean pasados como argumento en otro parámetro de la función para que esta sea más "genérica".

Programa 2: Arreglo como parámetro

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

using namespace std;

/* Declaración de promediar que recibe un arreglo */
double promediar(int valores[], int tamanyo);

int main() {
  const int NOTA_MAXIMA = 10; // Nota más alta posible
  const int CANT_NOTAS  = 10; // cantidad de notas
  // Se generan 'CANT_NOTAS' aleatorias para el vector 'notas'
  random_device rand;
  int notas[CANT_NOTAS];
  for(int &n : notas) { // notas entre 1 y NOTA_MAXIMA
    n = rand() % NOTA_MAXIMA + 1;
  }
  // Se muestran las notas generadas y se promedian:
  for(int &n : notas) {
    cout << "[" << n << "] ";
  }
  cout << "\n\nEl promedio es: "
       << promediar(notas, CANT_NOTAS) << endl;
}
/* Implementación de la función promediar */
double promediar(int valores[], int tamanyo) {
  double resultado = 0;
  // ¡¡Aquí no se puede utilizar el foreach!!
  for(int i=0; i < tamanyo; i++) {
    resultado += valores[i];
  }
  return resultado/tamanyo;
}
```

Debemos tener en cuenta que cuando se pasa un arreglo como argumento, este podrá ser modificado dentro de la función, ya que el arreglo (sin los corchetes) tiene almacenada la dirección de memoria. Es decir que siempre son referencias cuando se trata de parámetros en una función.

2.4. Strings como arreglo de caracteres

Los `strings` pueden ser considerados como arreglos de caracteres ('`char str[]`'), eso quiere decir que cada posición será un `char` en un vector. Esto podemos evidenciarlo utilizando los '`[]`' para acceder a cada uno de ellos por separado. Aunque en rigor no son lo mismo, esto será analizado a continuación.

Matías S. Ávalos© CC BY-NC-SA 4.0

```
string texto = "Hola!";
for(char &c : texto) {
  cout << "[" << c << "]";
}
cout << endl;
// Alternativa con índice:
for(int i=0; i < 5; i++) {
  cout << "[" << texto[i] << "]";
}
cout << endl;
```

```
[H][o][l][a][!]
```

¡Cuidado!

Se debe tener en cuenta que la **codificación** es importante, un **char** solo puede almacenar 1 byte (8 bits), por lo tanto si utilizamos un texto codificado en `UTF-8` veríamos los bytes por separados produciendo una salida no deseada:

```
string texto = "Programación"; // ó tiene 2 bytes
for(char &c : texto) {
  cout << "[" << c << "]";
}
cout << endl;
```

```
[P][r][o][g][r][a][m][a][c][i][?][?][n]
```

En el lenguaje `C` no existe el tipo de dato **string**, por lo tanto deben expresarse como arreglos de **char**, también denominados como *"cadena de caracteres"*:

```
char str[] = "Hola, este es un string de C";
cout << str << endl;
```

2.4.1. El carácter de finalización de un string

Como se observó con anterioridad al utilizar **atoi()** o **atof()** es necesario invocar **c_str()** una función interna del **string** de `C++` para poder llamar a dichas funciones de `C`. Ya que, como se mencionó con anterioridad, no son iguales, un **string** de `C++` no es exactamente un vector de **char**. Esto es evidenciado corriendo el siguiente fragmento de código:

```
char str_c[] = "Hola!";
string str_cpp = "Hola!";
cout <<"Tamaño del string de C: "<< sizeof(str_c) <<endl;
cout <<"Tamaño del string de C++: "<< sizeof(str_cpp) <<endl;
```


Matías S. Ávalos© CC BY-NC-SA 4.0

Cuyo resultado será algo parecido a lo siguiente:

```
Tamaño del string de C: 6
Tamaño del string de C++: 32
```

En ambos casos los resultados son a primera vista *enigmáticos*. En el caso de C++ se explica que el **string** almacena mucho más que simplemente el texto, ya que tiene funcionalidades internas como **c_str()** o **length()** entre otras herramientas de utilidad. Estos temas serán abordados en otra materia, ya que están relacionados con la Programación Orientada a Objetos (POO).

Pero, ¿cómo se explica que un vector de **char** que aparenta tener 5 elementos tenga tamaño 6? Cuando se almacena un texto en una variable del tipo vector de caracteres el sistema debe reconocer el final de la misma. El vector es una variable y ocupa un determinado espacio en la memoria, el sistema no tiene cómo delimitar donde termina esta información al menos que pongamos un **indicador de finalización** conocido como el carácter **'\0'**. De lo contrario se imprimirá todo lo que se encuentre en la memoria hasta que se tope con un carácter de terminación (al azar) o se intente acceder a un sector restringido por el sistema operativo y ocurra una violación de segmento. Cuando se escribe un *string* con comillas dobles el carácter de finalización está implícito, no se muestra, pero allí está.

```cpp
char str1[] = "Hola"; // caracter de terminación implícito
char str2[] = {'H','o','l','a'}; // vector de 4 posiciones
cout << str1 << endl; // bien
cout << str2 << endl; // MAL!!

// caracter de terminación explícito:
char str3[] = {'H','o','l','a', '\0'};
cout << str3 << endl; // bien, equivalente al str1
```

Por lo tanto, si deseara mostrar un texto *carácter a carácter* del cual desconocemos su longitud se podría recurrir a un algoritmo como el siguiente:

```cpp
// str es un arreglo de caracteres de longitud desconocida
for(int i=0; str[i] != '\0'; i++) {
  cout << "[" << str[i] << "]";
}
cout << endl;
```

Es recomendable utilizar el **string** que nos ofrece C++, al menos que sea imperioso para la aplicación desarrollada usar la menor cantidad de memoria posible, en cuyo caso, como se observó en la página 8 el arreglo de **char** es más eficiente.

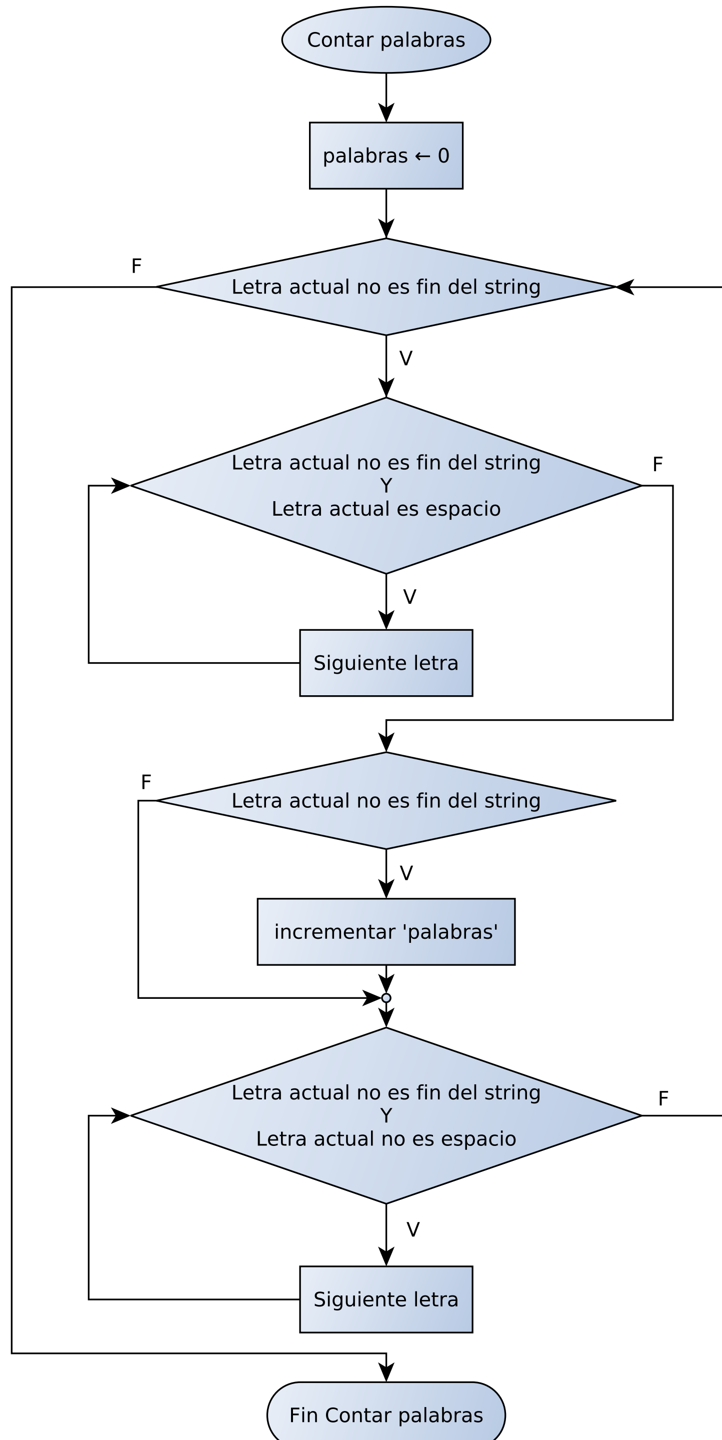
2.4.2. Algoritmos con strings

El manejo de *strings* es un tema muy habitual en los sistemas, porque gran parte de la información está en este formato o, al menos, al momento de representar la informa-

Matías S. Ávalos© CC BY NC SA 4.0

ción para los usuarios es imperativo. El usuario siempre interacturá con una interfaz que muestra y lee cadenas de texto.

Se analizarán algunos ejemplos de algoritmos con strings. Para empezar se desarrollará un algoritmo para contar palabras en un `string`. El mismo consiste en leer letra a letra el texto filtrando los espacios hasta encontrar una carácter que no sea espacio (es decir, una palabra), luego se recorren todas las letras de la palabra hasta encontrar un nuevo espacio y se repite hasta alcanzar el final del texto.



Matías S. Ávalos© CC BY-NC-SA 4.0

Programa 3: Contar palabras en un string

```
#include <iostream>
#include <cctype> // isspace()
using namespace std;

int contar_palabras(string texto);

int main() {
  string str = "  Esto   es  un   texto   de   prueba  ";
  cout << "El texto es \"" << str << "\"\n"
       << "Y tiene " << contar_palabras(str) <<" palabras.\n";
}

int contar_palabras(string texto) {
  int l = 0;
  int palabras = 0;
  while(texto[l] != '\0') {
    // se filtran todos los espacios:
    while(texto[l] != '\0' and isspace(texto[l])) {
      l++;
    }
    // Si no es el final del string, es una palabra nueva
    if(texto[l] != '\0') {
      palabras++;
    }
    // Se busca el próximo espacio (fin de la palabra actual)
    while(texto[l] != '\0' and not isspace(texto[l])) {
      l++;
    }
  }
  return palabras;
}
```

Salida del programa:

```
El texto es "  Esto   es  un   texto   de   prueba  "
Y tiene 6 palabras.
```

En este ejemplo, el índice `l` dentro de la función `contar_palabras()` es incrementado mientras haya espacios dentro del *string* consulandolo a través de la función estándar `isspace()` o se haya alcanzado el final del mismo (líneas 18 y 19). Si no se alcanzó el final, eso quiere decir que hay una palabra nueva y se incrementa el contador de palabras (líneas 22 y 23). Finalmente se busca el siguiente espacio, es decir, el final de la palabra encontrada incrementando `l` (líneas 26 y 27), para luego seguir recorriendo el *string* repitiendo las acciones hasta el final del mismo.

A continuación se desarrolla un algoritmo que invierte un texto. Esto se logra recorriendo letra a letra el texto de atrás hacia adelante almacenándolo en otro `string` llamado '`inv`' el cual debe tener el mismo "largo" que el texto original. Es importante conocer la *longitud* de la cadena para poder implementar este algoritmo.

Matías S. Ávalos© 4.0

Programa 4: Invertir un string

```
#include <iostream>

using namespace std;

string invertir(string texto);

int main() {
  string str = "Hola";
  cout << "El original es: \"" << str << "\"\n"
          "Invertido: \"" << invertir(str) << "\"\n";
}

string invertir(string texto) {
  // Asegura que 'inv' tenga la misma cantidad de chars que
  // 'texto'
  string inv = texto;
  int fin = 0;
  while(texto[fin] != '\0') {
    fin++;
  }
  for(int i=0; i < fin; i++) {
//  inicio <= final
    inv[i] = texto[fin-1-i];
  }
  return inv;
}
```

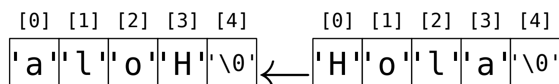
Salida del programa:

```
El original es: "Hola"
Invertido: "aloH"
```

En este algoritmo se inicializa la variable **inv** con el contenido en **texto**, de esta manera, es seguro que la cantidad de letras en ambos serán las mismas (línea 16). Luego se cuentan los caracteres dentro de texto para hallar el final del mismo y almacenarlo en **fin** (líneas 18 y 19). Por último, se recorren ambos *strings* almacenando en '**inv**' desde la última posición (**fin-1**) el carácter de texto hasta su primera posición **texto[0]**, decrementando el índice al restarle **i** y contrariamente aumentando el de **inv** con la misma variable (línea 23). Se analizará el comportamiento en la siguiente tabla de secuencia:

texto←"Hola"		fin←4		
i	fin-1-i	inv[i]	texto[fin-1-i]	char
0	3	inv[0]	← texto[3]	'a'
1	2	inv[1]	← texto[2]	'l'
2	1	inv[2]	← texto[1]	'o'
3	0	inv[3]	← texto[0]	'H'

Matías S. Ávalos© CC BY NC SA 4.0



Por último, se analiza un algoritmo que cuenta vocales en un texto dado. No se consideran las vocales acentuadas, ya que éstas dependerán del sistema operativo y la codificación que utilice, por ejemplo, si utilizara un sistema *Tipo-Unix* que codifica en `UTF-8` se requerirían 2 `char` para poder comparar las letras acentuadas, pero en un sistema que utiliza `ISO-8859-X` sí se podría lograr la comparación, ya que requieren un solo byte[2]. Además, excedería la complejidad que para el análisis del algoritmo. Aquí se recorrerá la cadena comparando cada una con las 5 vocales, incrementando la cantidad a medida que haya coincidencias.

Programa 5: Contar las vocales en un texto

```cpp
#include <iostream>
#include <cctype> // toupper()
using namespace std;

int contar_vocales(string texto);

int main() {
  string str = "  Esto   es  un   texto   de   prueba  ";
  cout << "El texto es \"" << str << "\"\n"
       << "Y tiene " << contar_vocales(str) << " vocales.\n";
}

int contar_vocales(string texto) {
  string vocales = "AEIOU";
  int cant_vocales = 0, i = 0;
  while(texto[i] != '\0') {
    for(int j=0; j < 5; j++) {
      // Se compara el carácter en mayúscula
      if(toupper(texto[i]) == vocales[j]) {
        cant_vocales++;
      }
    }
    i++;
  }
  return cant_vocales;
}
```

Salida del programa:

```
El texto es "  Esto   es  un   texto   de   prueba  "
Y tiene 10 vocales.
```

En este caso se utiliza una función estándar `toupper()` que recibe un `char` y devuelve siempre su mayúscula, para luego ser comparada con cada vocal en el arreglo `vocales` (línea 19). En caso de coincidir se incrementa la cuenta de vocales llevada en `cant_vocales` (línea 20) y, finalmente, se pasa a la siguiente letra (línea 23) hasta llegar al final del texto.

[2]Para más información consulte los apuntes de codificación de texto de la materia **EFSI I**.

Matías S. Ávalos© CC BY NC SA 4.0

2.4.3. Funciones internas de `string` en C++

El *tipo de dato* `string` contiene algunas funciones internas que son de gran utilidad.

1. `operador+` y `append()`:

   La función *apéndice* concatena *strings*, es decir que recibe como argumento un `string` y se lo anexa:

   ```
   string str = "Hola, ";
   str.append("cómo estás?");
   cout << str << endl; // se muestra "Hola, cómo estás?"
   ```

   Por otra parte, el operador '+' tiene un comportamiento similar, pero a diferencia de `append()` que solo puede operar con *strings* y se anexan al existente, este puede operar también con `char`, pero generando un nuevo *string* que debe ser asignado.

   ```
   string str = "Hola, ";
   str = str + "cómo estás" + '?';
   cout << str << endl; // se muestra "Hola, cómo estás?"
   ```

2. `c_str()`:

   Esta función devuelve una copia del `string` como un vector de `char` y es útil para utilizar funciones estándar de `C` como ya se ha demostrado.

3. `length()`:

   Esta función devuelve la longitud del `string`, por lo tanto, esta podría utilizarse como reemplazo al bucle que incrementa un contador hasta encontrar el carácter '`\0`'.

   ```
   string str = "Esto es un texto de prueba";
   int tamanyo = str.length();
   cout <<"\""<< str <<"\" tiene "<<tamanyo<<" chars.\n";
   /* Es equivalente a: */
   int tamanyo = 0;
   while(str[tamanyo] != '\0') {
     tamanyo++;
   }
   /*********************/
   ```

4. `substr()`:

   Obtener un *sub-string* o un texto parcial sobre uno de base y tiene 2 alternativas:

   a) 1 sólo parámetro: `substr(posición_inicial)`, se genera a partir de la posición pasada como argumento hasta el final del *string*.

   ```
   string str = "Esto es un texto";
   //                 ^--------->
   cout << str.substr(5) << endl; // "es un texto"
   ```


Matías S. Ávalos© CC BY NC SA 4.0

b) Con 2 parámetros: **substring(posición, cantidad)**, genera un *string* desde el comienzo hasta la cantidad de caracteres indicados.

```
string str = "Esto es un texto";
//                 ^---^
cout << str.substr(5, 5) << endl; // "es un"
```

Además, existe una función llamada **to_string()** que transforma números a un **string**, lo cual es de mucha utilidad si se desea anexar números a un texto con el **operador+** o la función **append()**.

```
string str = "Esto es una cadena, " + to_string(1)
           + " es un int, " + to_string(2.3) + " es un double"
             " y por último '" + 'a' + "' es un carácter.";
cout << str << endl;
```

Salida del programa:

```
Esto es una cadena, 1 es un int, 2.300000 es un double y por  ú
   ltimo 'a' es un carácter.
```

A continuación se muestra una alternativa de *invertir* utilizando estas herramientas:

```
1 string invertir(string texto) {
2   string inv = "";           // se inicializa vacío
3   int fin = texto.length(); // cantidad de chars en texto
4   for(int i=0; i < fin; i++) {
5     inv += texto[fin-1-i];  // incio <= final
6   }
7   return inv;
8 }
```

3. Multidimensionales (matrices)

La primera aproximación que podemos hacer a los arreglos con más de una dimensión es entenderlos como un vector de vectores. Supongamos entonces:

```
// vector de 3 posiciones
// cuyos elementos son vectores de 5 posiciones:
int matriz[3][5] = {{ 2, 3, 6,-1, 6},
                    { 7,-2,-7, 9,10},
                    {-9,-3, 0,22,-5}};
```

Se observa además, que al inicializar el arreglo de 2 dimensiones, cada uno de sus elementos son arreglos que también deben estar entre {}. Si se deseara acceder a un elemento del arreglo debemos indicar ambos índices.

Matías S. Ávalos© CC BY NC SA 4.0

Representación del arreglo
matriz

[0]

[0]	[1]	[2]	[3]	[4]
2	3	6	-1	6

[1]

[0]	[1]	[2]	[3]	[4]
7	-2	-7	9	10

[2]

[0]	[1]	[2]	[3]	[4]
-9	-3	0	22	-5

Por lo tanto si se quisiera acceder a una de las posiciones se debe indicar de la siguiente manera:

```
cout << "La posición (2, 4) de la matriz es: "<< matriz[1][3]
    << endl;
```

```
La posición (2, 4) de la matriz es: 9
```

3.1. Recorrer un arreglo multidimensional

Para poder recorrer un arreglo usualmente es necesario utilizar tantos bucles como dimensiones tenga. Por lo tanto, para un arreglo de dos dimensiones, serán necesarios dos bucles y dos índices:

```
// Para ingresar datos:
int matriz[3][5];
for(int fila=0; fila < 3; fila++) {
  for(int col=0; col < 5; col++) {
    matriz[fila][col] = pedirEntero("Ingrese posición ("
                    + to_string(fila+1) + ", "
                    + to_string(col+1) + "): ");
  }
}
// Para mostrar los datos:
for(int fila=0; fila < 3; fila++) {
  cout << "[ ";
  for(int col=0; col < 5; col++) {
    cout << matriz[fila][col] << " ";
  }
  cout << "]\n";
}
```

Alternativamente se puede usar un *foreach*:

Matías S. Ávalos© CC BY-NC-SA 4.0

```
// para cada 'fila' de 5 elementos en 'matriz':
for(int (&fila)[5] : matriz) {
  cout << "[ ";
  // para cada elemento de la fila...
  for(int &elem : fila) {
    cout << elem << " ";
  }
  cout << "]\n";
}
```

A continuación se expone un ejemplo concreto de utilización de arreglos multidimensionales, extendiendo el Programa 1 incluyendo notas trimestrales para cada materia y utilizando las funciones anteriormente expuestas para ingresar las notas y calcular el promedio de cada materia.

Programa 6: Boletin anual trimestral

```
1  #include <iostream>
2  using namespace std;
3
4  int pedir_entero(string mensaje);
5  int pedir_entero_entre(string mensaje, int min, int max);
6  double promediar(int valores[], int tamanyo);
7
8  int main() {
9    const int CANT_MATERIAS = 10;
10   const int CANT_NOTAS    = 3;
11
12   string materias[CANT_MATERIAS]={"Literatura","Matemática",
13                                   "Ed. Físca","Física","Historia",
14                                   "Geografía","Programación",
15                                   "Inglés","Alemán","Biología"};
16   int boletin[CANT_MATERIAS][CANT_NOTAS];
17           //  |               |
18           //  |               *-- Trimestre
19           //  *------------------ Materia
20   for (int i = 0; i < CANT_MATERIAS; i++) {
21     string texto = "Ingrese la nota de "+materias[i]+'\n';
22     for (int j = 0; j < CANT_NOTAS; j++) {
23       boletin[i][j] = pedir_entero_entre(
24            texto+"Del trimestre "+to_string(j+1)+": ", 1, 10);
25     }
26   }
27   cout << '\n';
28   for (int i = 0; i < CANT_MATERIAS; i++) {
29     cout << "Promedio de " << materias[i] << ": "
30          << promediar(boletin[i], CANT_NOTAS) << "\n";
31   }
32 }
33 /* Implementación de las funciones (no se muestra) */
```

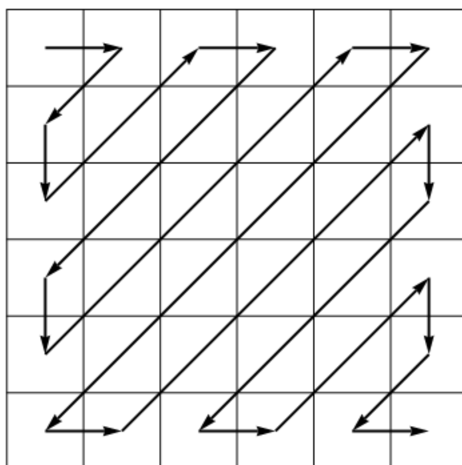

Matías S. Ávalos© CC BY-NC-SA 4.0

Si se deseara podría agregarse a los estudiantes, así se haría presente una nueva dimensión en el arreglo:

```
string estudiantes[CANT_ESTUDIANTES]={"María","Pedro","Juan",
                                      "Sofía","Malena"};
// ...
int boletines[CANT_ESTUDIANTES][CANT_MATERIAS][CANT_NOTAS];
        //    |                 |              |
        //    |                 |              *-- Trimestre
        //    |                 *----------------- Materia
        //    *----------------------------------- Estudiante
```

Esto se interpreta como que hay 5 estudiantes y cada uno tienen 10 materias, y cada materia 3 notas cuyo formato es un entero (`int`). Siempre se está hablando de una nota que es la información almacenada, pero está estructurada a través de los índices y relacionada con otros datos (`materias[]` y `estudiantes[]`).

Como aclaración final, aquí se recorrieron los arreglos de forma lineal, pero existen motivos algorítmicos para desear recorrer las matrices de forma parcial o completa en formas *no-lineales*, por ejemplo en *zig-zag*, por las diagonales, en espiral, etc. Esto es aplicado en diferentes algoritmos de compresión de datos y otros cálculos más avanzados que exceden el contenido de este curso.



Recorrido de matiz en Zig-Zag

4. Ejercitación

Codifique los siguientes programas utilizando arreglos y funciones en `C++`.

1. Pida ingresar 10 números reales y almacénelos en un arreglo. Luego sume los elementos que se encuentran en posiciones pares, reste los que se encuentran en posiciones impares, muestre ambos valores.

2. Pida ingresar 10 números reales y calcule el promedio de los mismos, luego muestre cuales están por encima de ese promedio.

Matías S. Ávalos© CC BY NC SA 4.0

3. Desarrolle un software que pida 10 enteros distintos, en caso de ingresar un valor repetido debe volver a pedir otro hasta completar los 10. Luego, muestre los números ingresados en pantalla.
4. Genere 10 números aleatorios entre -10 y 10 y guárdelos en un arreglo. Muéstrelos y determine cual es el mayor. La función debe estar definida para enteros.
5. Pida ingresar 10 valores numéricos y almacénelos en un arreglo. Busque dentro del arreglo el mayor y el menor valor e intercambie sus posiciones. Por último muestre el arreglo.
6. Desarrolle un software que pida ingresar el nombre completo de 5 estudiantes y sus promedios. Luego **muestre el nombre** del estudiante con mejor promedio.
7. Desarrolle un software que permita tirar 5 dados de 6 caras (al azar) y permita cambiar el valor de hasta 3 de ellos a elección indicando la posición de los dados que se desean cambiar (del 1 al 5).
8. Desarrolle un software donde pueda ingresar el nombre de un restaurante y 10 valoraciones entre 1 y 5 para cada uno. Muestre el nombre del restaurante y su puntuación promedio redondeado a 1 decimal.
9. Extienda el Programa 6 de la pág. 16, agregando una dimensión más al arreglo para 5 estudiantes, los mismos deben ser ingresados por la terminal y luego de mostrar los promedios trimestrales de cada materia de cada estudiante indique el nombre del que tenga mejor promedio general (promedio de los promedios).
10. Desarrolle un software que genere una matriz de enteros de 9x9 donde los valores se generen aleatoriamente del 1 al 9 y no puedan repetirse en ninguna fila o columna.

Matías S. Ávalos© CC BY NC SA 4.0