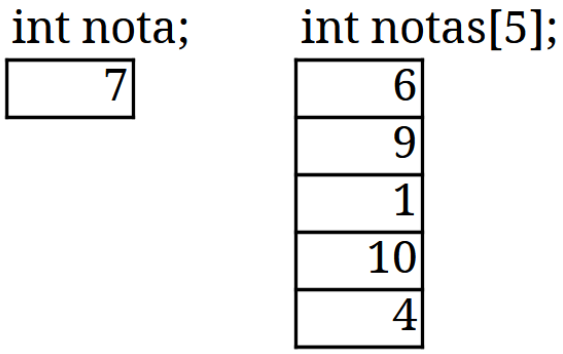
Hasta el momento hemos utilizado estructuras de datos que podrían considerarse básicas pero han sido más que efectivas y suficientes para permitirnos resolver una diversa cantidad de problemas. Sin embargo, a medida que deseemos crear algoritmos cada vez más complejos vamos a necesitar estructuras de datos más complejas que nos permitan almacenar la información de forma más práctica y sencilla.

Los vectores son estructuras de datos que permiten almacenar, en una misma variable, varios elementos a la vez. Anteriormente, lo que conocíamos como variable, y a partir de ahora llamaremos **variable simple,** permitía guardar de a un dato a la vez. Dicho dato podía cambiar, de allí el concepto de variable, pero al modificar el dato en una variable pisábamos el valor anterior. En consecuencia, no podíamos tener una variable simple que, a la vez, tenga los números 5, 7 y 9. Sólo puede tener uno de ellos a la vez.

En cambio, un vector sí permitiría tal situación. Dentro de una misma variable podríamos tener varios elementos y luego acceder individualmente a cada uno de ellos. Podemos visualizar gráficamente a un vector en contraposición a una variable simple de la siguiente manera:



Resulta claro que con la variable *nota* podemos guardar de a una nota a la vez. En cambio con el vector de *notas* tenemos la capacidad de guardar hasta 5 notas. La razón de que sean solamente cinco es el tamaño que utilizamos al declarar el vector.

Declaración de vectores

Si bien C++ dispone de otro tipo de vectores que se verán en Laboratorio II, en Laboratorio I trabajaremos con vectores y matrices de tamaño fijo. Esto significa que en el momento de la declaración del vector debemos determinar a partir de un valor entero y constante el tamaño del vector. Ese tamaño será la cantidad total de elementos que tendrá el vector y, por tanto, la capacidad máxima de elementos a almacenar.

Ejemplos:

```
const int SUCURSALES = 15;
int empleadosxSucursal[SUCURSALES];
float recaudacionxSucursal[SUCURSALES];
float recaudacion2023[12];
string domiciliosSucursales[SUCURSALES];
```

Como se puede observar en el código anterior, se declararon cuatro vectores. Tres de ellos de un tamaño definido a partir de una constante simbólica llamada SUCURSALES que contiene el valor 15. El otro de los vectores se declara con un tamaño 12 a partir de una constante literal.

> Los tamaños de los vectores siempre deben ser un valor constante. Declarar un vector definiendo el tamaño a partir de una variable, aunque cierta configuración del compilador permita hacerlo, es conceptualmente erróneo y puede llevar a errores de funcionamiento.

Inicialización de vectores

Como es sabido, una variable puede declararse sin inicializar asumiendo que dicha variable tendrá basura o puede declararse y a la vez inicializarse pudiendo elegir qué valores tendrá el vector. Por ejemplo, un uso práctico y habitual con vectores numéricos es inicializarlos en cero si luego vamos a utilizar dichos vectores para contar o acumular. La inicialización de un vector al momento de la declaración se realiza de la siguiente manera:

```
const int ALUMNOS = 5;
int legajos[ALUMNOS] = {1000, 2000, 3000, 4000, 5000};
string apellidos[ALUMNOS] = {"Seinfeld", "Costanza", "Benes", "Kramer",
"Newman"};
int materiasAprobadas[ALUMNOS] = {};
```

En el código anterior se puede observar cómo se inicializa el vector de legajos con cinco legajos y también los apellidos de los alumnos con cinco apellidos. El vector de materiasAprobadas también está siendo inicializado. En este caso, al poner un par de llaves sin datos dentro, lo que hace es establecer a cada uno de los elementos el valor cero.
Esto puede hacerse únicamente al momento de la declaración. Luego de declarado el vector ya no es posible asignarle valores de esta manera.

Acceso a los elementos

A esta altura del apunte, ya debe quedar claro que un vector admite varios valores a la vez en una misma variable. Por lo tanto, debe ser necesaria una sintaxis para poder acceder a los elementos individualmente. Los operadores que utilizaremos para acceder a cada elemento del vector son los corchetes [ ].

Veamos algunos ejemplos:

```
string nombresMeses[3];
int diasMeses[3];

nombresMeses[0] = "Enero";
nombresMeses[1] = "Febrero";
nombresMeses[2] = "Marzo";
diasMeses[0] = 31;
diasMeses[1] = 28; // Salvo en años bisiestos que son 29
diasMeses[2] = 31;

cout << nombresMeses[0] << " tiene " << diasMeses[0] << " días." << endl;
```

Salida por pantalla

```
Enero tiene 31 días.
```

Como se puede observar en el ejemplo, declaramos un vector de string de 3 elementos y también un vector de enteros de 3 elementos. El primero de ellos guarda los nombres de los primeros tres meses del año y el segundo guarda la cantidad de días que tiene cada mes. Haciendo uso de los corchetes podemos indicar cuál de los tres elementos del vector queremos acceder.
Por ejemplo, en la instrucción:

```
diasMeses[0] = 31;
```

Se le asigna el valor 31 al primer elemento del vector. Recordemos que el vector diasMeses

fue declarado como entero. Por lo que cada elemento del vector representará un número entero. También en la siguiente instrucción:

```
nombresMeses[2] = "Marzo";
```

Se asigna el valor "Marzo" al vector de nombresMeses. En este caso en el último de los elementos del vector ya que su tamaño es de 3 elementos.

Es importante tener claro que cualquier instrucción que utilizábamos con variables simples para modificar el valor de las mismas es posible utilizar con vectores. Por ejemplo, las siguientes instrucciones son totalmente válidas:

```
nombresMeses[0] = nombresMeses[1];
cin >> nombresMeses[2];
diasMeses[0]++;
diasMeses[1] += 10;
```

Por último, en la instrucción:

```
cout << nombresMeses[0] << " tiene " << diasMeses[0] << " días." << endl;
```

También se accede a los primeros elementos de cada vector. Pero a diferencia de los ejemplos anteriores no estamos modificando los datos sino que estamos obteniéndolos para mostrarlos por pantalla con cout.

Indexación en Base-0

Ya hemos aprendido cómo acceder a los distintos elementos de nuestros vectores. Tanto para modificar sus valores como para accederlos y utilizarlos como salida por pantalla o para asignarlo a otro elemento o variable.

También es evidente que los índices o posiciones que nos permiten acceder a cada uno de los elementos del vector tienen la particularidad de comenzar en cero.

Esto es muy común en la programación y prácticamente todos los lenguajes de programación manejan sus arrays y colecciones de esta manera.

Esto significa que si quisiera un vector para guardar el sueldo de 100 empleados de una empresa y quisiera pedir por teclado el sueldo del primer y último empleado tendría que escribir el siguiente conjunto de instrucciones:

```
float sueldos[100];
cin >> sueldos[0];
cin >> sueldos[99];
```

En consecuencia a lo mencionado anteriormente, el primero de los sueldos del grupo de 100 sueldos que permite guardar nuestro vector se encuentra en la posición 0. Y, como comenzó en cero, el último de los sueldos lo encontramos en la posición 99.
Podemos definir entonces que para un vector *vec* de tamaño *T*:
El primer elemento será → vec [0]
El último elemento será → vec [T - 1]

Esta regla se cumple siempre que un vector sea con indexación en base 0. En C++ siempre lo son.
Supongamos que queremos indicarle al usuario que ingrese el número de empleado al que desea asignar el sueldo y luego el sueldo del mismo. Para ello, deberíamos pedirle al usuario con qué número de empleado desea trabajar y luego poder modificar el sueldo correspondiente.
Para resolver esto, podríamos utilizar un código como el siguiente:

```
float sueldos[100], sueldo;
int numeroEmpleado, indice;
cin >> numeroEmpleado;
cin >> sueldo;
indice = numeroEmpleado - 1;
sueldos[indice] = sueldo;
```

Resumidamente, podríamos expresarlo de la siguiente manera:

```
float sueldos[100];
int numeroEmpleado;
cin >> numeroEmpleado;
cin >> sueldos[numeroEmpleado - 1];
```

En el código podemos observar que tanto en la variante 1 como en la variante 2, se solicita al usuario un número de empleado con el cual trabajar. El usuario no tiene porqué saber que esto se guardará en un vector y que los vectores tienen una indexación en base 0, por lo que ingresará un valor que se encuentre entre el rango 1 y 100.

Luego, nosotros tomamos dicho valor entre 1 y 100 y lo adecuamos a la indexación de nuestro vector. En este caso, disminuyendo en uno el valor ingresado por el usuario podemos guardar sin problemas el valor en nuestro vector.
Si quisiéramos mostrar por pantalla el sueldo de todos los empleados podríamos hacer un ciclo exacto y en cada iteración mostrar por pantalla el valor que contiene el vector en el índice que corresponde a la variable que controla el ciclo:

```
float sueldos[6] = { 40000, 504000, 320000, 594000, 90000, 120000 };
int i;
for (i=0; i<6; i++){
  cout << "El sueldo del empleado #" << i+1 << " es: $" << sueldos[i] << endl;
}
```

Desafío

Pares distintos con diferencia absoluta igual a Z

Hacer un programa que permita, de un vector de 50 números enteros con valores negativos, positivos o cero, encontrar la cantidad de pares de números distintos con una diferencia absoluta igual a Z.
El programa debe establecer los 50 números en la inicialización o bien pidiéndolos por teclado. También debe poder asignar el valor Z en la inicialización o por teclado. Debe mostrar por pantalla la cantidad de pares distintos que tengan una diferencia absoluta igual a Z.

Ejemplo:
{1, 5, 3, 4, 2}
Z = 3

El vector tiene 2 pares con diferencia absoluta igual a Z

En este caso, los pares son {5, 2} y {1, 4}

NOTA: No hace falta mostrar cuáles son los pares encontrados pero es un plus si logran hacerlo.
ACLARACIÓN: Por pares de números distintos se entiende que, si por ejemplo encontramos el par {1, 56}, entonces no tenemos que contar el par {56, 1}

El siguiente ejercicio se recomienda resolver luego de haber realizado la práctica de vectores.