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

FUNDAMENTOS DE PROGRAMACIÓN

Asignatura correspondiente al plan de estudios de la carrera de Ingeniería Informática

UNIDAD 9

VECTOR y STRUCT

2016

UNIDAD 9

Vector y Struct

Introducción

En esta unidad aprenderemos a emplear dos importantes estructuras de datos en C++. Aplicaremos primero los conceptos relacionados al uso de arreglos estudiados previamente con pseudocódigo, empleando ahora la sintaxis de C++. Veremos cómo definir y utilizar arreglos dinámicos de 1 y 2 dimensiones en C++, y analizaremos sus ventajas y limitaciones. Además estudiaremos otra estructura de datos importante, denomina en C/C++ *struct ,* y analizaremos el por qué de su presencia y las ventajas de su empleo. Veremos que es posible combinar estructuras de datos de acuerdo a las necesidades del caso a resolver.

Definición de arreglo

Definimos como array a la estructura de datos formada por una secuencia de elementos homogéneos (de igual tipo), donde cada elemento tiene una posición relativa dentro de la secuencia, posición que será establecida por uno o más índices

Arreglos lineales

Existen en C++ múltiples formas de obtener y utilizar un bloque de memoria para un arreglo lineal de datos. En esta guía vamos a utilizar el tipo de dato denominado `vector`. Para poder compilar un ejemplo que utiliza el tipo `vector`, se debe incluir la biblioteca vector mediante la directiva `#include<vector>`. Si se omite este #include el compilador arrojará un error indicando que “vector” no ha sido declarado.

Un `vector` representa un arreglo, cuyos elementos a su vez pueden ser de cualquier tipo de dato. Es decir, los elementos que contiene un `vector` pueden ser `int`, `float`, `char`, `string`, o hasta `vector` nuevamente. La única condición es que para un vector dado, todos sus elementos serán de un mismo y único tipo (recordar que los arreglos son estructuras homogéneas). El tipo de elemento se debe explicitar al definir el arreglo, a continuación de la palabra `vector` y entre signos mayor y menor. Ej:

```
vector<int> lista(200);
for (int i=0; i<200; i++)
    cin >> lista[i];
```

En este ejemplo, se dice que el tipo de la variable `lista` es `vector<int>` (es decir, arreglo lineal de enteros), y que tiene 200 elementos.

Se debe tener en cuenta que en C++ se utilizan posiciones "en base 0". Esto es, que la primer posición válida es la posición 0, por tanto para una declaración como la del ejemplo, donde se indica un arreglo de 200 elementos, las posiciones válidas del arreglo irán desde 0 a 199 inclusive (nótese el signo de menor estricto en la condición de parada del for).

Es importante destacar además que C++ no verifica la validez de los índices que se utilizan para acceder a los elementos de un arreglo. Por lo tanto, si se intenta acceder al elemento de la posición 200 (o cualquier otra mayor a 199) el acceso será inválido. Si ese fuera el caso, el comportamiento del programa es *indefinido*. Esto quiere decir que el programa podría continuar sin evidenciar el error, arrojar resultados incorrectos, o detenerse (no necesariamente en el punto del error, podría detenerse en cualquier punto posterior). Debido a esto, los errores en los índices suelen ser muy difíciles de diagnosticar en la práctica.

En la declaración de un arreglo mediante el tipo vector, como se observa en el ejemplo, el tamaño del mismo se coloca entre paréntesis luego de su identificador. Sin embargo, a diferencia de los arreglos que utilizamos en pseudocódigo, en C++ podemos realizar dos operaciones adicionales sobre un arreglo previamente definido: consultar y modificar su tamaño. Para consultar el tamaño de un arreglo `x` se utiliza la expresión `x.size()`. Para cambiar el tamaño, se utiliza la expresión `x.resize(N)`, donde `N` será el nuevo tamaño. Entonces, en el siguiente ejemplo:

```
vector<int> v(10);
cout << "v tiene " << v.size()
     << " elementos:" << endl;
for(unsigned int i=0; i<v.size(); ++i)
    cout << v[i] << endl;
v.resize(20);
cout << "v tiene " << v.size()
     << " elementos:" << endl;
for(unsigned int i=0; i<v.size(); ++i)
    cout << v[i] << endl;
```

el primer cout mostrará el mensaje "v tiene 10 elementos:" y a continuación el primer for mostrará los 10 enteros (posiciones del 0 al 9), mientras que el segundo cout mostrará el mensaje "v tiene 20 elementos" y a continuación el for los 20 enteros (posiciones del 0 al 19).

Notar que para los contadores de los ciclos se utiliza el tipo `unsigned int`[1] en lugar de `int` cuando se compara con el `.size()` del vector. Esto se debe a que como un índice no puede ser negativo, la mayoría de los compiladores utiliza enteros sin signo para representarlos. Si se intenta utilizar `int` el programa compila y funciona correctamente, pero en la compilación se genera un warning advirtiendo la diferencia de tipos en las comparaciones.

Inicialización de los elementos

Para inicializar un arreglo en el programa podemos recorrer cada elemento del mismo y asignar los valores correspondientes, tal como lo hacíamos en pseudocódigo.

```
// arreglo x formado por 100 enteros al azar
vector<int>  x(100);
for (int i=0; i<100; i++)
    x[i] = rand();
```

En este ejemplo, al crear el arreglo, se reserva memoria suficiente para los 100 elementos enteros que luego serán completados con valores aleatorios.

Cuando se requiera inicializar todos los elementos de un vector con un mismo valor (por ejemplo, colocar 0 en cada posición), este valor se puede ingresar a continuación de la dimensión, separándolo de esta mediante una coma:

```
// arreglo x formado por 100 unos
vector<int>  x(100,1);
for (int i=0; i<100; i++)
    cout << x[i] << endl;
```

De igual forma, al cambiar el tamaño con “resize” se puede agregar también un valor con el cual se llenarán las nuevas posiciones en el caso de que el nuevo tamaño sea mayor que el anterior.

Finalmente, existe además una sintaxis alternativa que permite definir e inicializar el arreglo con una secuencia de valores fija en una única instrucción, que se introduce como una lista delimitada por llaves, y con sus elementos separados por comas:

```
vector<float> v = { 1.618, 3.14, 42.0, 299e8 };
```

En este caso, el vector tomará por dimensión 4 (la longitud de la lista, que no hace falta explicitar) e inicializará las 4 posiciones con los 4 valores de la lista.

Resúmen de ejemplos

- Definir un vector de 100 elementos de tipo float, inicializados en 3.14:

```
vector<float> v(100, 3.14);
```

[1]Dado que el tipo de entero utilizado puede variar de un compilador a otro, lo más correcto sería en realidad utilizar el tipo size_t, ya cada compilador reemplaza size_t por el tipo de entero que efectivamente utiliza. Por el momento vamos a omitir este detalle, que será analizado más adelante en otra asignatura.

- Definir un vector de 4 elementos de tipo int a partir de una lista de valores:

```
vector<int> v = { 1, 2, 3, 4 };
```

- Asignar un valor a una posición, y mostrarlo:

```
v[i] = rand(); cout << v[i];
```

- Obtener el tamaño del arreglo:

```
cout << v.size();
```

- Cambiar el tamaño del arreglo:

```
v.resize(50);
```

- Mostrar todos los elementos:

```
for(unsigned int i=0; i<v.size(); ++i)
    cout << v[i] << endl;
```

Existen muchas otras operaciones que se pueden hacer con el tipo vector, pero no serán presentadas en este curso porque no serán necesarias para resolver los ejercicios que se plantearán en la práctica, y principalmente para evitar complicaciones innecesarias, ya que muchas de ellas requieren conceptos y sintaxis adicionales que se desarrollarán mucho más adelante.

Arreglos bi/multi-dimensionales

Lamentablemente no existe en C++ estándar una forma sencilla de operar con arreglos multidimensionales dinámicos, de forma similar a como se opera con el tipo vector. Para resolver este problema, se debe recurrir bibliotecas externas. En esta sección describiremos el uso de una biblioteca llamada “matriz” generada por esta cátedra con el fin de solucionar este problema. Entonces, para operar con matrices (arreglos bi-dimensionales), deberemos incluir esta biblioteca mediante la directiva `#include<matrix>`.

La diferencia entre una biblioteca estándar (como vector) y una externa (como matrix) radica en que la primera forma parte de cualquier entorno de desarrollo C++ (se instala junto con el compilador), mientras que la segunda debe obtenerse por separado (en este caso, la cátedra le indicará en la práctica cómo descargar los archivos adicionales).

Utilizando el tipo matrix, un arreglo bidimensional se declara de una forma muy similar a la que se utilizó previamente para vectores:

```
matrix<int> mat(10,6);  // matriz mat de 10x6 enteros
matrix<float> tab(3,5); // matriz de 3x5 flotantes
```

En este caso se deben especificar las dos dimensiones entre paréntesis, y opcionalmente también se puede agregar un valor para que todas las posiciones de la matriz tomen ese valor:

```
matrix<int> mat(3,2,42);  // matriz mat de 3x2 enteros
                          // con todas las posiciones
                          // inicializadas en 42
```

Al utilizar .size() para obtener el tamaño de una matriz, dado que ahora se trabaja con 2 dimensiones, se debe explicitar entre los paréntesis cuál de estos dos tamaños se desea consultar:

```
for (unsigned int i=0; i<mat.size(0); i++) {
   for (unsigned int j=0; j<mat.size(1); j++) {
      cout<<"fila "<<i<<" columna "<<j<<":";
      cin>>mat[i][j];
   }
}
```

Es importante señalar una diferencia respecto a la sintaxis que se utilizó en la etapa de pseudocódigo: cuando se indican dos índices para acceder a un elemento de una matriz, cada índice debe ir rodeado por un par de corchetes (en pseudocódigo se utilizaba un solo par de corchetes, y los índices se separaban por una coma).

Finalmente, cuando operamos sobre arreglos bi-dimensionales, es común en C++ asumir que el primer índice se corresponde con el número de filas, mientras que el segundo con el número de columnas. El ejemplo siguiente contiene el código C++ que permite mostrar una matriz dispuesta en filas y columnas en la pantalla.

```
#include<iomanip>
#include<iostream>
#include<matrix>
using namespace std;

int main() {
    matrix<int> m = { {12,34,56}, {78,90,100} };
    for (unsigned int i=0;i<m.size(0);i++) {
        // muestra la fila i
        for (unsigned int j=0; j<m.size(1); j++)
              cout<<setw(4)<<m[i][j];
        cout<<endl;  // avanza a la próxima línea
    }
}
```

En este ejemplo se ha inicializado una matriz `m` de 6 elementos enteros donde se asignan los datos por filas. Notar que la lista de valores con que se inicializa la matriz, es en realidad como una lista de vectores, donde cada uno se corresponde con una fila la matriz.

Arreglos y funciones

Es posible utilizar arreglos como parámetros de funciones o como tipos de retorno, sin realizar ninguna consideración adicional. Sin embargo, dado que en

general un arreglo contendrá muchos elementos, será recomendable pasarlos por referencia cuando sean argumentos (para evitar copiar tantos elementos), y utilizar el calificativo const para prevenir las modificaciones involuntarias:

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

vector<int> crea_vector(int n, int min, int max) {
   vector<int> v(n);
   for(int i=0; i<n; i++)
      v[i] = rand()%(max-min+1)+min;
   return v;
}

void muestra_vector(string nom, const vector<int> &v) {
    for (unsigned int i=0;i<v.size();i++)
        cout << nom << "[" << i << "] = " << v[i] << endl;
}

int main() {
    int n;
    cout << "Tamaño: ";
    cin >> n;
    vector<int> v = crea_vector(n, 0, 100);
    cout << "El vector es: " << endl;
    muestra_vector("v", v);
}
```

En el ejemplo, la primer función genera un nuevo arreglo con `n` enteros aleatorios entre `min` y `max`, y lo retorna. La segunda lo muestra anteponiendo a cada elemento el nombre del arreglo y su índice.

Matrices y funciones para arreglo lineales

La implementación de matrix que estamos utilizando esconde en realidad un vector de vectores. Es decir, una matriz de 3x2 enteros es un arreglo de 3 elementos, donde cada elemento es un vector de 2 enteros. Esto permite pasar una fila de una matrix a una función que reciba un vector.

```
#include<iostream>
#include<matrix>
#include<vector>
using namespace std;

void muestra_vector(const vector<int> &v) {
    for (unsigned int i=0;i<v.size();i++)
        cout << v[i] << endl;
}

int main() {
   matrix<int> m = { { 1, 2, 3, 4 },
                     { 2, 4, 6, 8 },
                     { 3, 5, 7, 9 } };
   cout << "Segunda fila: " << endl;
   muestra_vector( m[1] );
}
```

Nótese que solo se puede obviar el segundo índice. Es decir, este enfoque no sirve para mostrar una columna en lugar de una fila. Es por esto que en C++ es común asociar al primer índice con la el número de fila, y al segundo con el de columna.

Definición de registro

Definimos como registro a la estructura de datos formada por un conjunto de elementos (no necesariamente de igual tipo), donde cada elemento está identificado por un nombre (identificador) único dentro de dicho conjunto.

El tipo *struct.*

Un arreglo es una estructura de datos homogénea, es decir solo admite una colección de elementos de igual tipo. A menudo se requiere organizar múltiples datos de una misma entidad en una única estructura, pero admitiendo información de diferente naturaleza (tipo). En este caso se utilizan registros. En C y C++ se denomina a este tipo de estructuras **struct.** Un **struct** en C++ agrupa una colección de variables, que pueden ser de diferente tipo. Cada variable (denominada variable miembro, campo, o atributo) debe declararse individualmente con su propio y único identificador. Su sintaxis general es la siguiente:

```
struct nombre {
       miembro 1;
```

```
        miembro 2;
        miembro 3;
        .....
        miembro n;
};
```

En la definición es obligatorio explicitar la palabra clave **`struct`** y a continuación el identificador (**`nombre`** en el ejemplo) de la estructura. Luego, entre llaves deben definirse cada uno de los campos o miembros. Estos campos o miembros pueden ser variables simples, arreglos, o incluso instancias de otros structs.

Tomemos el siguiente ejemplo:

ficha	
	apellido
	nombres
	dni
	edad
	cant_materias

Los nombres de estos miembros deben ser diferentes, pero pueden coincidir con el identificador de alguna otra variable definida fuera de dicha estructura, ya que sólo serán válidos dentro del struct. Un struct como el del ejemplo podría declararse de la siguiente manera:

```
struct ficha {
    string apellido;
    string nombres;
    long dni;
    int edad;
    int cant_materias;
};
```

Al definir una estructura estamos planteando el esquema de la composición pero sin definir ninguna variable en particular. Es decir, en el ejemplo decimos cómo van a ser las fichas, pero todavía no existe ninguna ficha en particular. Luego de esta definición, `ficha` podrá ser considerado como un nuevo tipo de datos, y usado para declarar variables de ese tipo:

```
ficha x,y;  // x e y se declaran de tipo ficha
```

Se puede utilizar una lista de inicialización de forma similar a la que se presentó para arreglos, listando los valores para cada campo en el orden en que fueron declarados en el struct:

```
ficha x = { "Lopez", "Gerardo", 24567890, 21, 11 };
```

En este ejemplo, `ficha` es el nombre de la estructura, `x` la variable de tipo ficha y los datos especificados entre { } son los valores iniciales de los miembros `apellido, nombres, dni, edad, cant_materias.`

Procesamiento de una variable struct

Para procesar la información relacionada a una estructura podemos operar con sus miembros individualmente o en ocasiones con la estructura completa. Para acceder a un miembro individual debemos utilizar el identificador de la variable **`struct`**, un punto de separación y el nombre del miembro componente.

```
variable.miembro
```

Considere el siguiente código de ejemplo.

```
1. #include <iostream>
2. using namespace std;
3. //-------------------------------------------------------
4. struct registro{
5.    string ape;
6.    string nom;
7.    long dni;
8. };
  //-------------------------------------------------------
9. int main( ) {
10.   registro f,z;
11.   getline(cin,f.ape);
12.   getline(cin,f.nom);
13.   cin>>f.dni;
14.   cin.ignore();
15.   z=f;
16.   cout<<z.ape<<" "<<z.nom<<"---DNI:"<<z.dni<<endl;
17.   return 0;
18.}
```

Observe en la línea 10 la declaración de las variables f y z. Las líneas 11, 12 y 13 permiten asignar datos ingresando los valores de los miembros en modo consola. En la línea 14 se asigna la variable struct completa f a una variable de igual tipo z. En 15 se muestran los miembros de z.

Es posible entonces asignar un struct a otro. Esto permite entonces, pasar efectivamente por copia structs a funciones, o hacer funciones que retornen structs. Sin embargo no es posible mostrar (cout) o leer (cin) un struct completo en una sola instrucción. Estas operaciones deben realizarse sobre cada uno de sus campos por separado.

Arreglos y structs

Es posible emplear arreglos como miembros de una composición **`struct`** y también definir un arreglo cuyos elementos sean estructuras:

```
vector<ficha> z= { {"Lopez",  "Gerardo",24567890,21,11},
                   {"Giménez","Ana",    25689901,20,15},
                   {"Zapata", "Andrés", 26701231,19,8 },
                   {"Farías", "Marina", 23199870,22,15},
                   {"Martino","Manuel", 24500654,21,10} };
```

En el ejemplo anterior se declara e inicializa un arreglo z para elementos de tipo ficha. Recordando que un arreglo se opera por elementos, y dado que cada elemento es un struct, cada uno a su vez se debe operar por campo, entonces:

- mostrar el miembro dni del tercer elemento del arreglo debemos utilizar:

```
cout<<z[2].dni;
```

- cambiar el miembro edad a 22 en el 4to elemento del arreglo z debemos utilizar:

```
z[3].edad=22;
```

- mostrar un listado con los miembros dni y apellido de cada ficha del arreglo z, se codifica de la siguiente forma:

```
for (unsigned int i=0; i<z.size(); i++) {
    cout << z[i].apellido << " " << z[i].dni << endl;
}
```

En el segundo ejemplo :

```
struct EquipoFutbol {
    string nombre;
    string tecnico;
    vector<string> titulares;
    vector<string> suplentes;
};
```

se define un struct que representa un equipo de fútbol, y que contiene, además del nombre del equipo y de su técnico, un arreglo para los nombres de los jugadores titulares, y otro para los nombres de los suplentes. La lista de titulares se podría cargar y mostrar de la siguiente forma:

```
EquipoFutbol e1;
// cargar
e1.titulares.resize(11);
for (int i=0; i<11; i++)
    getline(cin,e1.titulares[i]);
// mostrar
for (int i=0; i<11; i++)
    cout << e1.titulares[i] << endl;
```