

VECTOR

Marco teórico

Para entender este tema es necesario conocer los conceptos de **Paradigma De La Programación Orientada A Objetos, Clases, Objetos y Contenedores** así como la interrelación entre ellos.

Paradigma De La Programación Orientada A Objetos

Es una metodología especial de programación de software en la que un programa informático se diseña como la interrelación entre un conjunto de instancias conocidas como objetos.

Clase (Class)

Es la **definición** de un tipo de dato, hecha por el usuario, donde puede incluir los miembros de esta y sus operaciones.

Por miembros de clase entenderemos las variables y funciones que dotan a las clases de un comportamiento y/o operaciones.

Objeto (Object)

Es una unidad dentro del programa informático, con un comportamiento, que ocupa espacio de memoria.

Contenedor (Container)

Es un objeto muy parecido a una **struct**, ya que puedes almacenar una colección de otros **objetos**. Pero al ser una **clase** te permite definir operaciones entre los objetos dentro de él.

Como se puede seguir, el paradigma da entrada a los demás puntos, de hecho ya hemos convivido con él al declarar nuestras variables del tipo **int**, **double**, etc. La clase, como se puede apreciar, al usar la palabra **definición** mantiene este concepto en el plano de lo abstracto dentro de nuestro programa, ya que aunque es en él como tal no tiene ninguna influencia sobre él; a diferencia del objeto que su propia existencia dentro del programa ocasiona un efecto sobre él.

Por último tenemos el **contenedor**, que justamente es la parte que nos interesa ya que los vectores viven dentro de él, este es un

objeto que por su propia estructura nos permite manipular información bajo las operaciones definidas en él.

Definición de Vector

Un vector es un contenedor secuencial que actúa como un arreglo que cambia de tamaño.

Intuición de Vector

Como los arreglos, previamente vistos, un vector sirve para almacenar datos de forma contigua en la memoria, pero para poder cambiar de tamaño entonces este hace uso de los **dynamic allocated arrays**, esto quiere decir que haremos uso de arreglos dinámicos.

En líneas generales, usar **memoria dinámica** implica crear variables que no son **declaradas** antes del tiempo de ejecución, sino durante de él. El problema de la memoria dinámica es el costo en tiempo, ya que la reservación, búsqueda y recuperación de información se ralentiza.

Cuando usamos la memoria dinámica los espacios de memoria se van asignando de forma relativamente aleatoria, pero cuando usamos **dynamic allocated arrays** **señalamos** los espacios de memoria de manera contigua (sin reservar), así por medio de apuntadores es como se puede acceder a estos, logrando depositar, consultar o retirar información.

De forma intuitiva a diferencia de cuando se usaban los arreglos estáticos, en este caso los espacios de memoria no se reservan de forma contigua pero no fija, lo que nos permite borrar la información de alguno y en cierto sentido desconectarlo de la secuencia que vemos

Declaración y funciones importantes

Los vectores pertenecen a la biblioteca **#include <vector.h>** la cual a parte de tener una implementación bastante optimizada de estos, nos proporciona diversas funciones (operaciones) que nos permiten operarlos de una forma relativamente rápida.

Por parte de la implementación, esta consiste de tres secciones:

1. Nombre de la estructura `vector`
2. Tipo de dato `<tipo de dato>`
3. Nombre de variable `miVector`

En cuanto código, este se escribe de la siguiente forma:

```
vector<tipo de dato> miVector
```

Cabe destacar que los tipos de datos pueden variar en cuanto se desee.

Para hablar de las funciones importantes, hablaremos de aquellas en la biblioteca ya mencionada, pero también es importante mencionar algunas fuera de ella. En cuanto las de la misma biblioteca las dividiremos como:

- Iteradores

begin - Nos devuelve un iterador que apunta al primer elemento del vector

Uso: `miVector.begin()`

end - Nos devuelve un iterador que apunta al último elemento del vector.

Uso: `miVector.end()`

- Capacidad

.size() - Nos retorna el número de elementos almacenados en nuestro vector.

Uso: `miVector.size()`

.empty() - Nos devuelve un True o False, dependiendo de si el vector tiene elementos o no.

Uso: `miVector.empty()`

- Acceso a elementos

[] - Al igual que en los arreglos estáticos, se usa este operador para poder acceder al elemento que deseamos.

Uso: `miVector[i]`, con i el índice de una entrada existente.

.front() - Nos devuelve el primer elemento en el vector.

Uso: `miVector.front()`

.back() - Nos devuelve el último elemento en el vector.

Uso: `miVector.back()`

- Modificadores

.push_back() - introduce el valor que le indiquemos por detrás del vector.

Uso: `miVector.push_back(valor)`

.pop_back() - Elimina el último elemento del vector.

Uso: `miVector.pop_back()`

.erase() - Permite eliminar el valor que deseemos, de la posición que queramos.

Uso: `miVector.erase(iterador apuntando a la posición deseada)`

`miVector.erase(iterador apuntando a inicio de rango, iterador apuntando al final)`

.clear() - Elimina todos los elementos del vector.

Uso: `miVector.clear()`

Con respecto a las funciones fuera de la biblioteca ya mencionada cabe destacar las siguientes, que se encuentran en la biblioteca `algorithm`:

count() - Retorna las veces que aparece un elemento en el vector.

Uso: `count(inicio de rango, final de rango, elemento)`

find() - Devuelve un apuntador al primer elemento encontrado en un rango, si no lo encuentra devuelve el último elemento del vector.

Uso: `find(inicio de rango, final de rango, valor)`

search() - Devuelve un apuntador al primer elemento de un rango determinado, si no encuentra devuelve el último elemento del vector.

Uso: `search(inicio de conjunto, final de conjunto, inicio del rango, final del rango)`

reverse() - Invierte el orden de los elementos.

Uso: `reverse(inicio de rango, final de rango)`