1151044 - Programación Orientada a Objetos
23 de febrero de 2026

1. Sobre el manejo de excepciones

En principio, las excepciones se pueden usar simplemente como un mecanismo adicional de control de flujo. Considérese una secuencia de llamadas a función anidadas (por ejemplo, que `main` llama a `f`, que `f` llama a `g`, que `g` llama a `h`, etcétera), en el que la última llamada encuentra la solución a un problema dado. Para que dicho valor le sea transferido a `main`, tendríamos que regresarlo de cada llamada a función, desde la última hasta la primera. Una excepción permitiría «aventar» o «lanzar» el resultado y dejar que `main` lo capture. Un ejemplo de la vida real se puede encontrar en el programa `https://racc.mx/uam/trimestre-actual/2026-i/poo/ejemplos/caso_estudio/sin_excepciones.cpp` que no usa excepciones, y en el programa `https://racc.mx/uam/trimestre-actual/2026-i/poo/ejemplos/caso_estudio/con_excepciones.cpp` que sí las usa. Ambos códigos realizan la misma tarea y no es importante entender qué están haciendo; lo que importa es contrastar los códigos y entender cómo las excepciones pueden ayudar a simplificar esta situación en particular.

Aún así, las excepciones se usan casi siempre para notificar *errores normalmente fatales*. Por ejemplo, el `operator[]` de `std::vector` no valida que el índice proporcionado sea válido y el programa puede morir si se usa uno.

```
std::vector<int> v;     // arreglo vacío
std::cout << v[5];      // muy mal: el índice es inválido
std::cout << v[-123];   // ¡peor aún!
```

Sin embargo, `std::vector` también ofrece la función miembro `.at` que funciona como el `operator[]` pero que valida el índice proporcionado. Podemos hacer algo similar con nuestra implementación de `arreglo_dinamico`:

```
#include <stdexcept>

class arreglo_dinamico {
   //...
public:
   //...
   T& at(int i) {
      if (0 <= i && i < tam) {
         return p[i];
      } else {
         throw std::out_of_range("Indice invalido");
      }
   }
};
```

El tipo de dato `std::out_of_range` es un `struct` de la biblioteca de C++ definido en `<stdexcept>` con un constructor que toma una cadena que denota el mensaje de error. La excepción se puede capturar en `main`, aunque la ocurrencia de tal excepción es evidencia clara de un bug en el programa y no queda claro qué podríamos hacer para intentar recuperarnos del error.

```
int main( ) {
   try {
      std::vector<int> v;
      std::cout << v.at(5);
   } catch (const std::exception& ex) {
      std::cout << ex.what( );
   }
   // podemos continuar normalmente porque capturamos al excepción
   // pero la excepción era evidencia de que algo muy malo pasó
}
```

Los tipos de excepciones estándar definidos en `<stdexcept>` se pueden capturar como `std::excepcion`. Esto se puede gracias a un tema que pronto veremos. La función miembro `.what( )` devuelve el mensaje de error.

Otro ejemplo de excepción estándar es `std::bad_alloc`, que es elevada cuando le pedimos memoria a `new` pero ésta ya no puede entregarla (básicamente porque a nuestra computadora se le acabó la memoria RAM). No hay mucho que podamos hacer en esta situación; lo más sano tal vez simplemente sea no capturar la excepción y dejar que ésta termine el programa.

Las excepciones son el único mecanismo con el que un constructor puede notificar de un error. Nótese que un constructor *no tiene tipo de retorno*. Esto significa que no podemos devolver un booleano para indicar si la construcción

tuvo éxito o no; tendremos que tomar la decisión entre elevar la excepción directamente en el constructor, o bien, dejar el objeto en un estado inválido y recordar verificar tal estado posteriormente.

```
class archivo {
public:
   archivo(const std::string& ruta) {   // similar a std::vector<char> pero para cadenas
      //...
   }
}

int main( ) {
   archivo a("tarea.txt");
   // si no pudimos abrir el archivo en el constructor ¿qué hacemos?
   // ¿elevamos una excepción o dejamos el objeto en un estado invalido?
}
```

2. Sobre las secuencias y contenedores estándar de C++

El tipo `std::vector<T>` no es el único tipo de la biblioteca de C++ que usa memoria dinámica. A expensas de volver a listar `std::vector`, tres de las más importantes son:

- `std::vector<T>` de `<vector>`: modela un arreglo dinámico. Ya hemos visto ejemplos en clase, por ejemplo en `https://racc.mx/uam/trimestre-actual/2026-i/poo/ejemplos/38_vector.cpp`.
- `std::string` de `<string>`: similar a `std::vector<char>` (incluso tiene funciones miembro `push_back`, `pop_back` y el `operator[]`) pero que, a diferencia de `std::vector`, sí se puede leer e imprimir como un todo directamente con `std::cin` y `std::cout`. Puede verse un ejemplo en `https://racc.mx/uam/trimestre-actual/2026-i/poo/ejemplos/54_ejemplo_string.cpp`.
- `std::deque<T>` de `<deque>`: similar a `std::vector<T>` pero que también tiene funciones miembro `push_front` y `pop_front` para agregar elementos por la izquierda y quitar el primer elemento, respectivamente. Pareciera entonces que `std::deque` es mejor que `std::vector`, pero su representación interna es mucho más complicada y por eso puede ser un poco más lento y usar un poco más de memoria. Puede verse un ejemplo en `https://racc.mx/uam/trimestre-actual/2026-i/poo/ejemplos/55_doble_cola_deque.cpp`.