# TD3: Algoritmos y Estructuras de Datos

Prof. Agustín Garassino, Gervasio Pérez

Segundo Semestre de 2024

Clase Teórica 9
Invariante de Representación
Memoria Dinámica en C++

# Resumen

En la clase de hoy veremos

- Repaso invariante de representación.
- Memoria dinámica en C++
- Manejo de memoria en clases

# Invariante de Representación – *Repaso*

## Invariante de Representación: Rep

- Son las condiciones que debe cumplir la estructura de representación para mantener su coherencia interna y para ser una instancia válida para representar el tipo abstracto que se quiere implementar.
- Todos los algoritmos pueden asumir que vale Rep en la Pre.
- Todos los algoritmos deben asegurar que vale Rep en la Post.
- El Rep puede ayudar a implementar algoritmos que satisfacen complejidades temporales deseadas.
- En el Rep sólo podemos usar los **observadores** de los tipos que usamos para describir las condiciones.

# Invariante de Representación – *Formal*

El invariante $Rep(e : estr)$ es un predicado lógico que toma una instancia de la estructura de representación *estr* y predica sobre los componentes de la estructura.

Ejemplo: $Rep_3$ de `Conjunto<T>` era "sin repetidos y ordenado"

$$Rep_3(e : estr) \equiv sinRepetidos(\texttt{e.\_elems}) \land ordenado(\texttt{e.\_elems})$$

$$ordenado(v : \texttt{vector<T>}) \equiv (\forall i : \texttt{int})\ 1 \leq i < |v| \implies v[i-1] \leq v[i]$$

donde la estructura *estr* era `{ vector<T> _elems; }`

# Invariante de Representación – Ejemplo

Si la estructura *estr* fuese `{vector<T> _elems; int _cardinal;}`
*Rep* de debería decir

"_elems sin repetidos y ordenado, y
_cardinal = |_elems|"

$$Rep(e : estr) \equiv sinRepetidos(\texttt{e.\_elems}) \wedge ordenado(\texttt{e.\_elems}) \wedge$$
$$|\texttt{e.\_elems}| = \texttt{e.\_cardinal}$$

El Rep también tiene que describir cómo se relaciona internamente la información de los distintos componentes de la estructura.

# Clases `std` en nuestras estructuras

Para leer y escribir invariantes en lógica formal, serán útiles los siguientes predicados y operadores:

- `vector<T> v`
  - $|v| : \textbf{int}$
  - $v[i] : T$, donde $0 \leq i < |v|$
- `set<T> s`
  - $\#(s) : \textbf{int}$, el cardinal del conjunto $s$
  - $x \in s : \textbf{bool}$, si $x$ pertenece al conjunto $s$
- `map<Tkey, Tvalue> m`
  - $claves(m) :$ `set<Tkey>`, el conjunto de claves del diccionario $m$.
  - $m[k] : Tvalue$, el valor asociado a la clave $k$ en el diccionario $m$. donde $k \in claves(m)$.
- `list<T> ls`
  - $|ls| : \textbf{int}$
  - $ls[i] : T$, donde $0 \leq i < |ls|$

Ejemplos de uso en Rep formal: ver Ejercicio 4 de la Guía 6.

# Memoria dinámica – Motivación

Si no contasemos con `vector<int>` ni las otras clases de la `std`,
¿Cómo podríamos diseñar una cadena de enteros?

Queremos un tipo de datos cadena de enteros que permita:

- almacenar 0 más elementos,
- conocer su longitud,
- acceder al $i$-esimo elemento,
- agregar un nuevo entero al final,
- calcular la suma de toda la cadena.

# C++: Cadena

```
1  class Cadena {
2    public:
3    // Constructor
4    Cadena();
5
6    // Observadores
7    int longitud() const;
8    int iesimo(int pos) const;
9    // Pre: 0 <= pos < longitud()
10
11   // Modificadores
12   void agregarAtras(int e);
13
14   // Otras operaciones
15   int sumar() const;
```

# C++: Uso de Cadena

Nos gustaría poder usar la Cadena de esta manera:

```
1  int main() {
2      Cadena c;
3      c.agregarAtras(1);
4      c.agregarAtras(1);
5      for (int i = 2; i < 10; i++){
6          int nuevo = c.iesimo(i-1) + c.iesimo(i-2);
7          c.agregarAtras(nuevo);
8      }
9      cout << c.sumar();
10 }
```

# Cadena – ¿estructura de representación?

¿Qué estructura de representación usamos? Recuerden que para este ejemplo queríamos evitar usar la `std`.

- ¿Esto alcanza?

```
1  private:
2    int valor;
```

- ¿Y esto alcanza?

```
1  private:
2    int valor0;
3    int valor1;
```

- ¿Ahora?

```
1  private:
2    int valor0; int valor1; int valor2; int valor3; int valor4;
3    int valor5; int valor6; int valor7; int valor8; int valor9;
```

# Cadena – ¿estructura de representación?

```
1  class Cadena {
2    public:
3      Cadena();
4      int longitud() const;
5      int iesimo(int pos) const;
6      void agregarAtras(int e);
7      int sumar() const;
8
9    private:
10     // ???
11 }
```

- No sabemos a priori cuántos enteros estarán contenidos en la cadena.
- Necesitamos una forma de "construir un nuevo entero" cuando haga falta, extendiendo la memoria que ocupa nuestra Cadena.
- Para hacer esto en C++ necesitamos usar memoria dinámica

# C++: Heap

El Heap es el espacio de memoria dinámica, está disponible para su uso a pedido y su límite de espacio es la cantidad de memoria que se encuentra libre en la computadora.

¿Cómo uso el heap?

a) Mediante clases como `vector`, `set`, `list`, `map`,... que usan el Heap internamente

b) Con los operadores de C++ **`new`** (*crear algo en el heap*) y **`delete`** (*borrar algo que había creado en el heap*).

# C++: T* operator new T

- El operador **new** T crea un nuevo objeto de tipo T en el heap, y devuelve un **puntero** (T*) a éste.
- El objeto creado **es independiente** del scope en el que es creado y lo "sobrevive".

```
1  int * crear_un_int(int valor) {
2    return new int(valor);
3  }
4  int f() {
5    int* i1 = crear_un_int(3);
6    int* i2 = crear_un_int(4);
7    return 0;
8  } // ¿qué sucederá con i1 e i2?  --> se pierden
```

¿Cómo accedo a los valores apuntados por i1 e i2?

# C++: T operator*(T*)

- El operator*(T*) devuelve una referencia al objeto de tipo T apuntado por el puntero.
- Podría pasar que el puntero tenga una dirección de memoria que aloja un valor "basura" no válido.
- Si el puntero es invalido o ya fue hecho **delete**, la operación puede abortar el programa..

```
1  int * crear_un_int(int valor) {
2    return new int(valor);
3  }
4  int f() {
5    int* i1 = crear_un_int(3);
6    int* i2 = crear_un_int(4);
7    return *i1 + *i2; // devuelve "7"
8  } // ¿qué sucederá con i1 e i2?  --> se pierden
```

# C++: `operator->`

- Cuando tenemos un `class T*` o `struct T*`, el `operator->` es una manera de acceder a métodos o variables miembro.
- Sintácticamente, para una variable p de tipo `class T*` o `struct T*`,
  - `p->f()` es equivalente a `*p.f()` y
  - `p->item` es equivalente a `*p.item`

Ejemplo: en breve...

# C++: operator delete *T

- El operador **delete**(T*) elimina al objeto de tipo T existente en el heap apuntado por el puntero.
- La variable de tipo puntero sigue existiendo, pero ya no apunta a un objeto válido.
- Se suele asignar la constante **nullptr** cuando un puntero ya no apunta a nada, o cuando inicialmente todavía no apunta a nada.

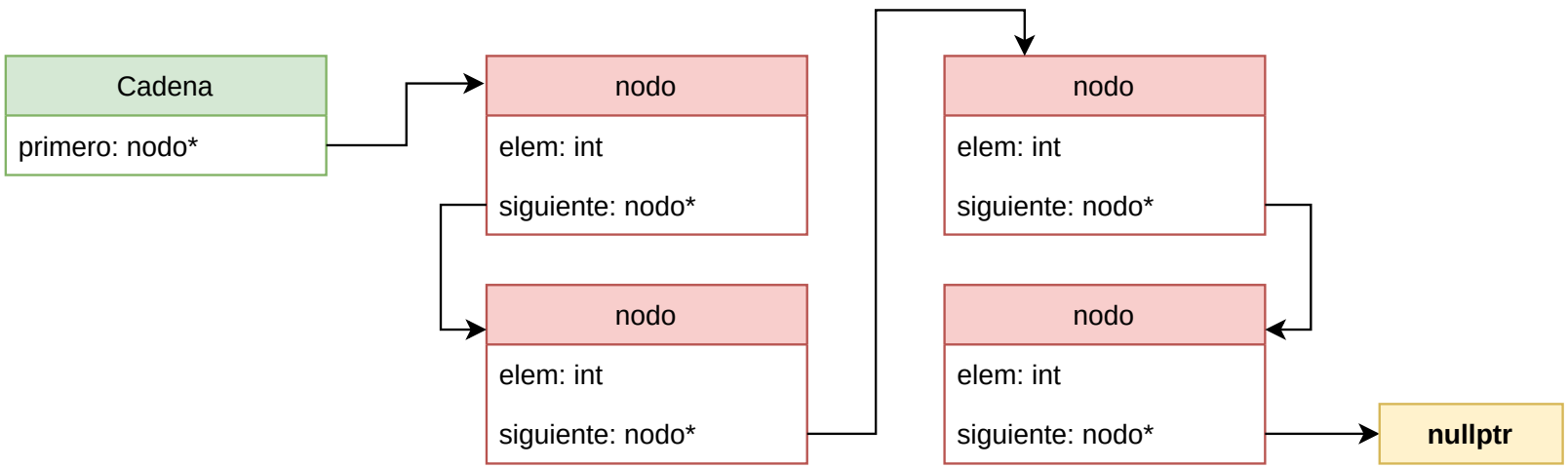
```
1   int mi_funcion() {
2     int *p1 = nullptr, *p2 = nullptr;
3     p1 = crear_un_int(3);
4     p2 = crear_un_int(4);
5     int n = *p1 + *p2;
6     delete p1; // borra del heap el entero apuntado por p1
7     delete p2; // borra del heap el entero apuntado por p2
8     p1 = nullptr; p2 = nullptr;
9     return n;
10  }
```

# Cadena – ¿estructura de representación?

```
1  class Cadena {
2    public:
3      Cadena();
4      int longitud() const;
5      int iesimo(int pos) const;
6      void agregarAtras(int e);
7      int sumar() const;
8
9    private:
10     // ???
11 }
```

# Cadena – Estructura propuesta

```
1  class Cadena{
2  public:
3    /* interfaz */
4  private:
5    struct nodo {
6        nodo(int e) : elem(e), siguiente(nullptr) {}
7        int elem;
8        nodo* siguiente;
9    };
10   nodo* primero;
11   // Rep: La cadena de nodos está correctamente conectada
12   //      (no tiene ciclos y, en el ultimo nodo, siguiente = nullptr)
13 };
```



# Cadena – Constructor

```
1  Cadena::Cadena() {
2    primero = nullptr;
3  }
```

# Cadena – longitud

```
1  int Cadena::longitud() const {
2    int res = 0;
3    nodo * actual = primero;
4    while (actual != nullptr) {
5      actual = actual->siguiente;
6      res++;
7    }
8    return res;
9  }
```

# Cadena – agregarAtras

```
1   void Cadena::agregarAtras(int e) {
2     if (primero == nullptr)
3         primero = new nodo(e);
4     else {
5         nodo* actual = primero;
6         while (actual->siguiente != nullptr)
7             actual = actual->siguiente;
8         actual->siguiente = new nodo(e);
9     }
10  }
```

# Cadena – iesimo

```
1   int Cadena::iesimo(int pos) const{
2     // Pre: 0 <= pos < longitud()
3
4     nodo* actual = primero;
5     int i = 0;
6     while (i < pos){
7       actual = actual->siguiente;
8       i++;
9     }
10    return actual->elem;
11  }
```

# Cadena – sumar

```
1  int Cadena::sumar() const{
2    int res = 0;
3    for(int pos = 0; pos < this->longitud(); pos++){
4      res = res + this->iesimo(pos);
5    }
6    return res;
7  }
```

# Cadena – Complejidad

| Algoritmo | Complejidad |
|---|---|
| `Cadena()` | $\Theta(1)$ |
| `longitud()` | $\Theta(n)$ |
| `iesimo(pos)` | $\Theta(n)$ |
| `agregarAtras(e)` | $\Theta(n)$ |
| `sumar()` | $\Theta(n^2)$* |

*Se resuelve en $\Theta(n)$ recorriendo la cadena de punteros.

# Cadena – sumar en O(n)

```
1  int Cadena::sumar() const{
2    int res = 0;
3    nodo * p = primero;
4    while (p != nullptr) {
5      res += p->elem;
6      p = p->siguiente;
7    }
8    return res;
9  }
```

# Cadena – Destructor

- Una clase `A` en C++ siempre tiene un método destructor: `~A()`
- El método destructor se invoca automáticamente cuando se termina el scope del objeto.
- El método destructor también se invoca cuando el objeto se destruye explícitamente con un **`delete`**.
- En general, si no usamos memoria dinámica de manera explítica, no es necesario que implementemos el método.
- En cambio, si usamos **`new`** en nuestros métodos tenemos que ocuparnos que toda la memoria que pedimos se libere con un **`delete`**.

# Cadena – Destructor

Se debe agregar el destructor a la interfaz de la clase:

```
1  class Cadena {
2    public:
3      Cadena();
4      int longitud() const;
5      int iesimo(int pos) const;
6      void agregarAtras(int e);
7      int sumar() const;
8      ~Cadena(); // Método destructor de Cadena
9
10   private:
11     // ...
12 }
```

# Cadena – Destructor

Y se debe dar su implementación, asegurandose de liberar toda la memoria del heap que se haya "pedido" para construir y modificar el objeto.

```
1  Cadena::~Cadena() {
2    nodo * actual = primero;
3    while (actual != nullptr) {
4      nodo* sig = actual->siguiente;
5      delete actual;
6      actual = sig;
7    }
8    primero = nullptr;
9  }
```

# Cadena – estructura alternativa

```
1  private:
2    struct nodo {
3      //...
4    };
5    nodo* primero;
6    nodo* ultimo;
7    int _longitud;
8    // Rep:
9    // - La cadena de nodos está correctamente conectada
10   // (no tiene ciclos)
11   // - _longitud coincide con la cantidad de objetos nodo
12   //   encadenados
13   // - ultimo coincide con el valor del último nodo*
14   //   alcanzable desde primero
15 };
```

¿Cómo afecta a los algoritmos vistos?

# Cadena – Algoritmos

```
1   int Cadena::longitud() const {
2     return _longitud;
3   }
4
5   void Cadena::agregarAtras(int e) const {
6     nodo* nuevo = new nodo(e);
7     if (ultimo != nullptr) {
8       ultimo->siguiente = nuevo;
9       ultimo = nuevo;
10    } else {
11      primero = nuevo;
12      ultimo = nuevo;
13    }
14    _longitud++;
15  }
```

# Cadena – Complejidad (mejorada)

| Algoritmo | Complejidad |
|---|---|
| `Cadena()` | $\Theta(1)$ |
| `longitud()` | $\Theta(1)$ |
| `iesimo(pos)` | $\Theta(n)$ |
| `agregarAtras(e)` | $\Theta(1)$ |
| `sumar()` | $\Theta(n)$ |

# T10: Resumen de hoy

Representación de un Tipo Abstracto de Datos:

- Ejemplos de Rep escrito formalmente.

Memoria Dinámica

- Memoria: Heap
- El tipo puntero
- Comandos **`new`** y **`delete`**
- Ejemplo: `Cadena`
  - Estructura y algoritmos
  - Análisis de complejidades
  - Mejora de la estructura e impacto en algoritmos

Guía de ejercicios:

- Ya pueden hacer la Guía 6: Introducción a Tipos Abstractos de Datos.