TD3: Algoritmos y Estructuras de Datos

Prof. Agustín Garassino, Gervasio Pérez

Segundo Semestre de 2024

Clase Teórica 8
Tipos Abstractos de Datos

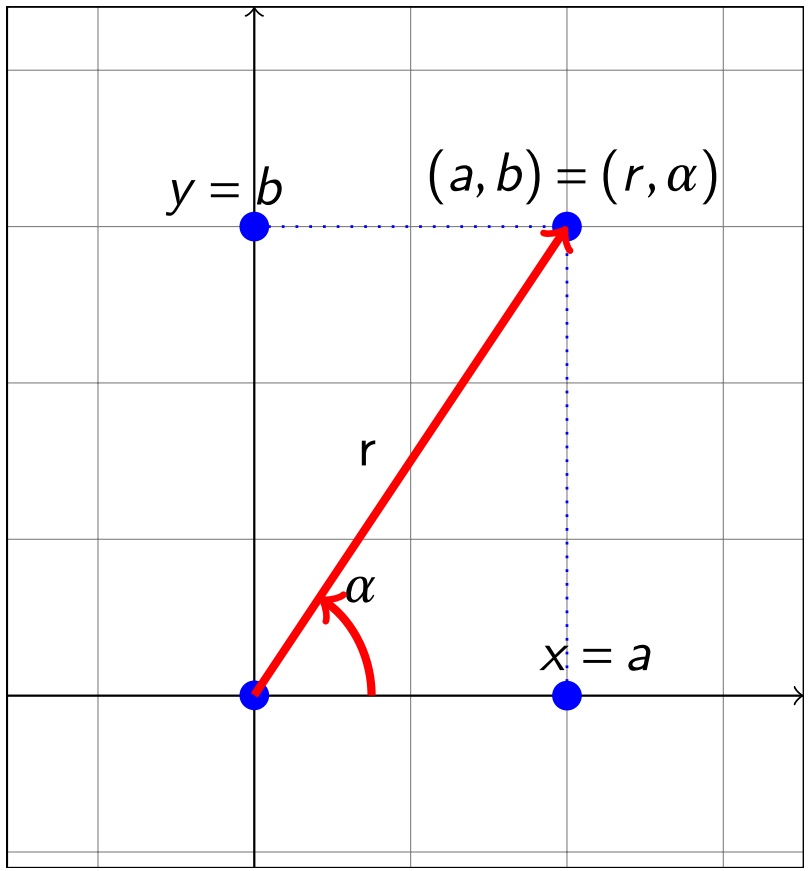
Resumen

En la clase de hoy veremos

- Encapsulamiento y abstracción
- Un repaso de clases en C++
- Métodos Observadores y Modificadores de una clase
- Estructura de representación

Ejemplo: Punto2D

Queremos modelar un Punto2D para describir números de $\mathbb{R}^2$ representados de manera cartesiana (coordenadas (x, y)) y de manera polar (r, α) donde α es el ángulo contrarreloj del punto con respecto al eje $x+$ y r es el módulo (distancia al centro).



Punto2D: propuesta inicial

```
1     void sumar(float x1, float y1, float x2, float y2,
2                float & xres, float & yres){
3       xres = x1 + x2;
4       yres = y1 + y2;
5     }
6     void crear_polar(float r, float ang,
7                      float & xres, float & yres){
8       xres = r * cos(ang);
9       yres = r * sin(ang);
10    }
11    float modulo(float x, float y){
12      return sqrt(x*x + y*y);
13    }
14    float angulo(float x, float y){
15      return atan2(y, x);
16    }
```

¿Qué problemas ven con esta propuesta? Tengo una abstracción del comportamiento pero no de los datos.

Abstracción para abordar problemas complejos

Cuando un problema computacional crece en complejidad (de comportamiento y de datos), es útil aplicar técnicas de abstracción.

Un primer paso para hacerlo es escribir funciones con nombres declarativos que abstraigan el **comportamiento** de operaciones complejas.

El paso siguiente es escribir nuevos tipos de datos que abstraigan **comportamiento y datos** de entidades, conceptos y requerimientos que sean relevantes a nuestro dominio.

Para esto, definiremos tipos abstractos, que combinan información y algoritmos, y abstraen las dificultades de manejar esa combinación.

- Liskov: Inventing Data Abstraction
- How Data Abstraction changed Computing forever | TEDxMIT

Punto2D: interfaz del tipo abstracto

```
1   class Punto2D {
2   public:
3
4     float x() const; // coordenada x
5     float y() const; // coordenada y
6
7     Punto2D(float cx, float cy);
8     static Punto2D crear_polar(float r, float ang);
9
10    void espejar_y();
11
12    Punto2D operator+(const Punto2D & b) const;
13    float modulo() const;
14    float angulo() const;
15
16  private:
17    // Implementación
18  }
```

Tipo abstracto – composición

Un tipo abstracto tiene los siguientes componentes:

Especificación: (pública)

- Interfaz: describe las operaciones que se pueden realizar sobre una instancia del tipo abstracto.

Implementación: (privada)

- Estructura de Representación: describe cómo se implementa el tipo abstracto en base a otros tipos.
- Algoritmos: implementan las operaciones que describe la interfaz, operando sobre la estructura de representación respetando y manteniendo su coherencia interna.

Tipo abstracto – Interfaz

La interfaz de un tipo debe describir:

- comportamiento funcional (operaciones, tipos de los parámetros de entrada y salida, Pre y Post condiciones)
- comportamiento no funcional (complejidad temporal de las operaciones, aliasing, manejo de memoria).

Una buena interfaz minimiza la visibilidad de las decisiones internas que se hayan tomado para implementar el tipo abstracto.

Vamos a describir un tipo abstracto con cuatro tipos de operaciones:

- Observadores: devuelven toda la información que caracteriza a una instancia, sin modificarla; deberían ser un conjunto minimal
- Constructores: crean nuevas instancias del tipo abstracto
- Modificadores: modifican la instancia, pueden devolver información.
- Otras operaciones: otros observadores no esenciales

Punto2D en C++ : parte pública

```
1  class Punto2D {
2  public:
3    // observadores
4    float x() const; // coordenada x
5    float y() const; // coordenada y
6
7    // constructores
8    Punto2D(float cx, float cy);
9    static Punto2D crear_polar(float r, float ang);
10
11   // modificadores
12   void espejar_y();
13
14   // otras operaciones:
15   Punto2D operator+(const Punto2D & b) const;
16   float modulo() const;
17   float angulo() const;
18 /* ... sigue ... */
```

Ejemplo: Tipo Punto2D – Espec. de operaciones

Tipo Punto2D

Observadores:

- `float x() const // coordenada x`
- `float y() const // coordenada y`

Constructores:

- `Punto2D(float cx,float cy)`
  Post: Crea un `Punto2D` con `cx` como coordenada `x`, `cy` como coordenada `y`.
- `static Punto2D crear_polar(float r, float ang)`
  Post: Crea un `Punto2D` con `r*cos(ang)` como coordenada `x`, `r*sin(ang)` como coordenada `y`.

Los métodos **`static`** son "de clase" y se ejecutan sin un objeto **`this`** sobre el cual operar. En este ejemplo, `crear_polar` sigue el patrón de constructor con nombre: una función que oficia de constructor.

Modificadores:

- `void espejar_y()`
  Post: Modifica la instancia de `Punto2D` de modo que sus coordenadas pasan a ser `x()`, `-y()`.

Otras operaciones:

- `Punto2D operator+(const Punto2D & b) const`
  Post: Devuelve un `Punto2D` con coordenadas `x()` y `y()` iguales a la suma de las coordenadas de `this` y `b`.
- `float modulo() const`
  Post: Devuelve el módulo del `Punto2D` (distancia al origen).
- `float angulo() const`
  Post: Devuelve el ángulo del `Punto2D` (en radianes).

Punto2D: estructura de representación

Podemos elegir entre varias estructuras de representación.
¿Hay alguna que sea mejor que otra?

```
class Punto2D {
public:
  float x() const;
  float y() const;

  /* resto de la interfaz */

private:
  // coordenadas cartesianas
  float _x, _y;
};
```

```
class Punto2D {
public:
  float x() const;
  float y() const;

  /* resto de la interfaz */

private:
  // coordenadas polares
  float _r, _ang;
};
```

Abstracción de datos: sin importar la estructura de representación, la interfaz y su especificación funcional siguen siendo las mismas.

Ejemplo: Tipo Conjunto<T>

Objetivo: abstraer el comportamiento de los conjuntos matemáticos, que contienen elementos de tipo T.

Consideraciones:

- La operación que caracteriza a un conjunto es el $\in$ (pertence).
- Es decir, un conjunto C queda definido completamente por qué elementos contiene o no.
- Otras operaciones que tiene sentido modelar: $\cup$ (unión) y $\cap$ (intersección).
- También es necesario poder agregar/quitar de a un elemento al conjunto.

Ejemplos: `Conjunto<int>`, `Conjunto<string>`, `Conjunto<Punto2D>`, `Conjunto<Conjunto<int>>`, etc.

Ejemplo: Tipo Conjunto<T> – Interfaz

Observadores:

- `bool pertenece(const T & elem) const`

Constructores:

- `Conjunto<T>()`
  Post: Crea un `Conjunto<T>` vacío.

Modificadores:

- `void agregar(const T& e)`
  Post: Modifica `this` agregando el elemento e.
- `void quitar(const T& e)`
  Post: Modifica `this` quitando el elemento e, si no contenía e queda igual.

Más modificadores:

- `void unir_con(const Conjunto<T> & c)`
  Post: Modifica `this` agregando todos los elementos de c.
- `void intersecar_con(const Conjunto<T> & c)`
  Post: Modifica `this` manteniendo sólo los elementos que aparecían originalmente en `this` y en c simultáneamente.

Otras operaciones:

- `int cardinal() const`
  Post: Devuelve la cantidad de elementos del conjunto.

Conjunto<T> en C++: interfaz[1]

```
1  template <typename T>
2  class Conjunto {
3  public:
4    // observadores
5    bool pertenece(const T& elem) const;
6
7    // modificadores
8    void unir_con(const Conjunto& c);
9    void intersecar_con(const Conjunto& c);
10   void agregar(const T& elem);
11   void quitar(const T& elem);
12
13   // otras operaciones
14   int cardinal() const;
15 /** ... sigue ... **/
```

[1] En breve veremos qué significa `template <typename T>`.

Conjunto<T> en C++: estructura de representación

Como representación de nuestro conjunto utilizaremos un vector para almacenar sus elementos.

```
// ... //
private:
  vector<T> _elementos;
}
```

¿Cualquier instancia de `vector<T>` es válida como representación de un conjunto?

- `{}`
- `{3, 8, 5, -2}`
- `{-2, 3, 5, 8, 5, -2}`
- `{-2, 3, 5, 8}`

Tipo abstracto – Estructura de Representación

- La estructura de representación describe con qué tipos de datos concretos construiremos una instancia del tipo abstracto.
- Para implementar el mismo tipo abstracto, podríamos elegir diferentes estructuras dependiendo del contexto.

Invariante de Representación: Rep

- La estructura de representación debe mantener su coherencia interna. Estas propiedades se documentan formalmente en un Invariante de representación.
- Todos los algoritmos pueden asumir que vale Rep en la Pre.
- Todos los algoritmos deben garantizar que vale Rep en la Post.
- El Rep puede ayudar a implementar algoritmos que satisfacen complejidades temporales deseadas.

Conjunto<T> en C++: invariante de representación

En este ejemplo, podemos elegir entre distintos Rep:

- **Opción 1**: No pedir nada sobre `_elementos`.
  $Rep_1 \equiv Verdadero$

  Ejemplos: `{}`, `{3, 8, 5, -2}`, `{-2, 3, 5, 8, 5, -2}`, `{-2, 3, 5, 8}`

- **Opción 2**: Pedir que `_elementos` no tenga repetidos.
  $Rep_2 \equiv (\forall i,j : \texttt{int})\ 0 \leq i < j < |\texttt{\_elementos}| \implies \texttt{\_elementos[i]} \neq \texttt{\_elementos[j]}$

  Ejemplos: `{}`, `{3, 8, 5, -2}`, `{-2, 3, 5, 8}`

- **Opción 3**: Pedir que `_elementos` no tenga repetidos y esté ordenado de manera creciente.
  $Rep_3 \equiv (\forall i : \texttt{int})\ 0 \leq i < |\texttt{\_elementos}| - 1 \implies \texttt{\_elementos[i]} < \texttt{\_elementos[i+1]}$

  Ejemplos: `{}`, `{-2, 3, 5, 8}`

Conjunto<T> en C++: invariante de representación

Rep_1: no pedir nada. Rep_2: sin repetidos. Rep_3: sin repetidos y ordenado.

Algoritmo	Rep_1	Rep_2	Rep_3
`agregar`	$O(1)$ amortizado[2]	$O(n)$	$O(n)$
`quitar`	$O(N)$	$O(n)$	$O(n)$
`pertenece`	$O(N)$	$O(n)$	$O(\log n)$
`cardinal`	$O(N \log N)$ (mirar [3])	$O(1)$	$O(1)$
`unir_con(b)`	$O(M)$	$O(n \cdot m)$	$O(n+m)$

- N = Cantidad de elementos de **`this`** contando todas las veces que se agregó un elemento que ya estaba presente.
- n = **`this`**`->cardinal()` (elementos distintos)
- m = `b.cardinal()`

[2] `vector<T>.push_back()` tiene complejidad $O(1)$ amortizado, lo que significa que si se hacen n inserciones, el costo total es $O(n)$.

[3] Se podría hacer más eficiente utilizando un `map` o `set` implementado sobre tabla de hash.

Tipo abstracto – Algoritmos

Una vez definida una interfaz y elegida una estructura, estamos en condiciones de escribir los algoritmos que implementarán la interfaz definida, operando sobre la estructura de manera adecuada.

- Nuestro código será responsable de mantener el Rep de la estructura de representación.
- El código puede aprovechar las restricciones del Rep para cuestiones de, por ejemplo, eficiencia.

Funciones auxiliares:

- En general, las definiermos en la parte privada de la clase.
- No formarán parte de la interfaz.
- Se podría definir Pre y Post pero no lo haremos, explicaremos en castellano su comportamiento.

Conjunto<T> en C++: interfaz

```
1  template <typename T>
2  class Conjunto {
3    /*...*/
4  }
```

La palabra clave **template**

- Si queremos implementar clases como vector<T> que tiene un parámetro de tipo T, necesitamos, en C++, usar templates para poder dejar a T a elección del usuario.
- Un template define una "receta" para crear clases y funciones. Por ejemplo, vector es un template, y vector<**int**>, vector<**float**> y vector<vector<**bool**>> son tres posibles clases concretas que se fabrican a partir de ese template.
- Instanciar un template es análogo a hacer *copy-paste* del template reemplazando T por el tipo elegido.

Conjunto<T> en C++: algoritmos

Elegimos Rep_2 ("sin repetidos") como Invariante de representación.

```
1  template <typename T>
2  bool Conjunto<T>::pertenece(const T& e) const {
3    return _elementos.end() != std::find(_elementos.begin(),
4                                         _elementos.end(),e);
5  }
6
7  template <typename T>
8  void Conjunto<T>::agregar(const T& e) {
9    if (!pertenece(e))
10      _elementos.push_back(e);
11 }
12 template <typename T>
13 void Conjunto<T>::quitar(const T& e) {
14   if (pertenece(e)) { // Lo pongo al final y quito el último
15    std::swap(std::find(_elementos.begin(), _elementos.end(), e),
16              _elementos.end()-1);
17    _elementos.pop_back();
18   }
19 }
```

Conjunto<T> en C++: algoritmos (II)

```
1   template <typename T>
2   void Conjunto<T>::unir_con(const Conjunto<T>& c) {
3       for (const T& e: c._elementos){
4           if (!pertenece(e))
5               _elementos.push_back(e);
6       }
7   }
8   template <typename T>
9   void Conjunto<T>::intersecar_con(const Conjunto<T>& c) {
10      for (const T& e: _elementos){
11          if (!c.pertenece(e))
12            quitar(e);
13      }
14  }
```

T09: Resumen de hoy

Interfaz de un Tipo Abstracto de Datos:

- Abstracción de comportamiento y datos.
- Distinción entre observadores y modificadores.

Implementación de un Tipo Abstracto de Datos:

- Estructura de representación
- Invariante de representación (Rep).
- Vínculo entre Rep, algoritmos y orden de complejidad.

Sintaxis de `template` en C++ para usar parámetros de tipo.