Constructores y Lista de Inicialización (repaso)

Algoritmos y Estructuras de Datos II

Conceptos

- Constructor por defecto: `T::T()`
- Constructor por copia: `T::T(const T&)`
- Destructor: `T::~T()`
- Operador asignación: `T::operator=(const T&)`

Constructor por copia

¿Qué pasaría si Lista no tuviera constructor por copia?

```
class Lista {
  public:
    Lista();
    Lista(const Lista& otra);

    void agregarAtras(T&);
    int longitud() const;
    ...

  private:
    struct Nodo {
        T valor;
        Nodo* siguiente;
    }

    Nodo* primero;
}
```

```
int main() {
    Lista<int> l1;
    l1.agregarAtras(1);
    Lista<int> l2(l1);
    l2.agregarAtras(2);
    l1.longitud(); // ??
}
```

Constructor por copia

¿Qué pasaría si Lista no tuviera constructor por copia?

```
class Lista {
  public:
    Lista();
    Lista(const Lista& otra);

    void agregarAtras(T&);
    int longitud() const;
    ...

  private:
    struct Nodo {
        T valor;
        Nodo* siguiente;
    }

    Nodo* primero;
}
```

```
int main() {
    Lista<int> l1;
    l1.agregarAtras(1);
    Lista<int> l2(l1);
    l2.agregarAtras(2);
    l1.longitud(); // ??
}
```

¡Pizarrón!

¿Dónde, dónde está el constructor por copia?

¿Dónde se llama al constructor por copia en este ejemplo?

```
int maximo(Lista<int> l) {
    int max = l[0];
    for (int i = 1; i < l.longitud(); i++) {
        if (l[i] > max) {
            max = l[i];
        }
    }
    return max;
}

Lista<int> l1;
l1.agregarAtras(1);
l1.agregarAtras(3);
l1.agregarAtras(2);
int m = maximo(l);
```

¿Y en este?

```
Lista<int> l1;
l1.agregarAtras(1);
Lista<int> l2 = l1;
```

¿Y en esteeee?

```
Lista<int> rango(int desde, int hasta) {
    Lista<int> ret;
    for (int i = desde; i < hasta; i++) {
        ret.agregarAtras(i);
    }
    return ret;
}

Lista r = rango(5, 25);
```

¿Y en este otro?

```
Lista<int> l1;
l1.agregarAtras(1);
l1.agregarAtras(10);
Lista<int> l2;
l2 = l1;
```

¿Qué pasaría si Lista no tuviera operador de asignación?

```
class Lista {
  public:
    Lista();
    Lista(const Lista& otra);
    Lista& operator=(const Lista& otra);

    void agregarAtras(T&);
    int longitud() const;
    ...

  private:
    struct Nodo {
        T valor;
        Nodo* siguiente;
    }

    Nodo* primero;
}
```

```
int main() {
    Lista<int> l1;
    l1.agregarAtras(1);
    Lista<int> l2;
    l2.agregarAtras(10);
    l2 = l1;
    l2.agregarAtras(2);
    l1.longitud(); // ??
}
```

¿Qué pasaría si Lista no tuviera operador de asignación?

```
class Lista {
  public:
    Lista();
    Lista(const Lista& otra);
    Lista& operator=(const Lista& otra);

    void agregarAtras(T&);
    int longitud() const;
    ...

  private:
    struct Nodo {
        T valor;
        Nodo* siguiente;
    }

    Nodo* primero;
}
```

```
int main() {
    Lista<int> l1;
    l1.agregarAtras(1);
    Lista<int> l2;
    l2.agregarAtras(10);
    l2 = l1;
    l2.agregarAtras(2);
    l1.longitud(); // ??
}
```

¡Pizarrón!

Constructor por copia

¿Cómo evito usar la asignación en este caso?

```
class MaximoRapido {
  public:
    MaximoRapido(const Lista<int>& l);
    int maximo() const;

  private:
    Lista<int> _lista;
}

MaximoRapido::MaximoRapido(const Lista<int>& l) {
    _lista = l;
}
```

```
MaximoRapido::MaximoRapido(const Lista<int>& l) : _lista(l) {}

template<class T>
Lista::Lista(const Lista& otra) {
    _primero = null;
    for (int i = 0; i < otra.longitud(); i++) {
        this->agregarAtras(otra[i]);
    }
}
```

Lista de inicialización

```
class Fecha {
 public:
  // pre: anio > 0, mes \in [1, 12],
  // dia \in [1, diasEnMes(anio, mes)]
  Fecha(Anio anio, Mes mes, Dia dia);
  Fecha(Fecha fecha, Periodo periodo);

  Anio anio() const;
  Mes mes() const;
  Dia dia() const;

  bool operator==(Fecha o) const;
  bool operator<(Fecha o) const;

  void sumar_periodo(Periodo p);

 private:
  Anio _anio;
  Mes _mes;
  Dia _dia;

  void ajustar_fecha();
  void sumar_anios(Anio anios);
  void sumar_meses(Mes meses);
  void sumar_dias(Dia dias);
};
```

```
class Intervalo {
 public:
  // pre: desde < hasta
  Intervalo(Fecha desde, Fecha hasta);
  Intervalo(Fecha desde, Periodo periodo);

  Fecha desde() const;
  Fecha hasta() const;

  int enDias() const;

 private:
  Fecha _desde;
  Fecha _hasta;
};
```

```
class Periodo {
 public:
  Periodo(int anios, int meses, int dias);

  int anios() const;
  int meses() const;
  int dias() const;

 private:
  int _anios;
  int _meses;
  int _dias;
};
```

```
Intervalo::Intervalo(Fecha desde, Fecha hasta) {
  _desde = desde;
  _hasta = hasta;
}
```

Lista de inicialización

```
class Fecha {
 public:
  // pre: anio > 0, mes \in [1, 12],
  // dia \in [1, diasEnMes(anio, mes)]
  Fecha(Anio anio, Mes mes, Dia dia);
  Fecha(Fecha fecha, Periodo periodo);

  Anio anio() const;
  Mes mes() const;
  Dia dia() const;

  bool operator==(Fecha o) const;
  bool operator<(Fecha o) const;

  void sumar_periodo(Periodo p);

 private:
  Anio _anio;
  Mes _mes;
  Dia _dia;

  void ajustar_fecha();
  void sumar_anios(Anio anios);
  void sumar_meses(Mes meses);
  void sumar_dias(Dia dias);
};
```

```
class Intervalo {
 public:
  // pre: desde < hasta
  Intervalo(Fecha desde, Fecha hasta);
  Intervalo(Fecha desde, Periodo periodo);

  Fecha desde() const;
  Fecha hasta() const;

  int enDias() const;

 private:
  Fecha _desde;
  Fecha _hasta;
};
```

```
class Periodo {
 public:
  Periodo(int anios, int meses, int dias);

  int anios() const;
  int meses() const;
  int dias() const;

 private:
  int _anios;
  int _meses;
  int _dias;
};
```

```
Intervalo::Intervalo(Fecha desde, Fecha hasta) {
  _desde = desde;
  _hasta = hasta;
}
```

```
const_Intervalo.cpp: In constructor 'Intervalo::Intervalo(Fecha, Fecha)':
const_Intervalo.cpp:59:46: error: no matching function for call to 'Fecha::Fecha()'
 Intervalo::Intervalo(Fecha desde, Fecha hasta) {
                                              ^
```

Lista de inicialización

```
class Fecha {
 public:
  // pre: anio > 0, mes \in [1, 12],
  // dia \in [1, diasEnMes(anio, mes)]
  Fecha(Anio anio, Mes mes, Dia dia);
  Fecha(Fecha fecha, Periodo periodo);

  Anio anio() const;
  Mes mes() const;
  Dia dia() const;

  bool operator==(Fecha o) const;
  bool operator<(Fecha o) const;

  void sumar_periodo(Periodo p);

 private:
  Anio _anio;
  Mes _mes;
  Dia _dia;

  void ajustar_fecha();
  void sumar_anios(Anio anios);
  void sumar_meses(Mes meses);
  void sumar_dias(Dia dias);
};
```

```
class Intervalo {
 public:
  // pre: desde < hasta
  Intervalo(Fecha desde, Fecha hasta);
  Intervalo(Fecha desde, Periodo periodo);

  Fecha desde() const;
  Fecha hasta() const;

  int enDias() const;

 private:
  Fecha _desde;
  Fecha _hasta;
};
```

```
class Periodo {
 public:
  Periodo(int anios, int meses, int dias);

  int anios() const;
  int meses() const;
  int dias() const;

 private:
  int _anios;
  int _meses;
  int _dias;
};
```

```
Intervalo::Intervalo(Fecha desde, Fecha hasta)
  : _desde(desde), _hasta(hasta) {};
```

Lista de inicialización

```
class Fecha {
 public:
  // pre: anio > 0, mes \in [1, 12],
  // dia \in [1, diasEnMes(anio, mes)]
  Fecha(Anio anio, Mes mes, Dia dia);
  Fecha(Fecha fecha, Periodo periodo);

  Anio anio() const;
  Mes mes() const;
  Dia dia() const;

  bool operator==(Fecha o) const;
  bool operator<(Fecha o) const;

  void sumar_periodo(Periodo p);

 private:
  Anio _anio;
  Mes _mes;
  Dia _dia;

  void ajustar_fecha();
  void sumar_anios(Anio anios);
  void sumar_meses(Mes meses);
  void sumar_dias(Dia dias);
};
```

```
class Intervalo {
 public:
  // pre: desde < hasta
  Intervalo(Fecha desde, Fecha hasta);
  Intervalo(Fecha desde, Periodo periodo);

  Fecha desde() const;
  Fecha hasta() const;

  int enDias() const;

 private:
  Fecha _desde;
  Fecha _hasta;
};
```

```
class Periodo {
 public:
  Periodo(int anios, int meses, int dias);

  int anios() const;
  int meses() const;
  int dias() const;

 private:
  int _anios;
  int _meses;
  int _dias;
};
```

```
Intervalo::Intervalo(Fecha desde, Periodo periodo)
  : _desde(desde), _hasta(desde) {

    _hasta.sumar_periodo(periodo);
};
```

Lista de inicialización

```
Fecha::Fecha(Fecha fecha, Periodo periodo) {
  _anio = fecha.anio();
  _mes = fecha.mes();
  _dia = fecha.dia();

  sumar_periodo(periodo);
}
```

```
Intervalo::Intervalo(Fecha desde, Periodo periodo)
  : _desde(desde), _hasta(desde , periodo) {

};
```

Lista de inicialización

A partir de C++11

```
Fecha::Fecha(Fecha fecha, Periodo periodo)
  : Fecha(fecha) {
    this->sumar_periodo(periodo);
}

Lista::Lista(const Lista& otra) : Lista() {
    for (int i = 0; i < otra.longitud(); i++) {
        agregarAtras(otra[i]);
    }
}
```

Iteradores y Algoritmos Genéricos en C++

Algoritmos y Estructuras de Datos II

Colecciones

Conocemos los siguientes tipos de *colecciones*:

- Arreglo.
- Secuencia.
- Conjunto.
- Multiconjunto.
- Diccionario.

Operaciones sobre colecciones

Preguntas típicas sobre colecciones:

- Dado un elemento, ¿está en la colección?

  $x \in \texttt{conj}$

  $\text{def?}(x, \texttt{dicc})$

- Listar todos los elementos de una colección.
- Encontrar el elemento más chico de la colección.
- *etc.*

Operaciones sobre colecciones

¿Cómo recorremos una colección?

- **Arreglo.** Tamaño y acceder al i-ésimo (`operator[]`).
- **Secuencia.** prim y fin.
- **Conjunto.** dameUno y sinUno.
- **Multiconjunto.** dameUno y sinUno.
- **Diccionario.** claves y obtener.

¿Hay una manera uniforme de recorrerlas?

Operaciones sobre colecciones

¿Cómo hacemos para mirar los primeros 5 elementos de una lista simplemente enlazada en C++?

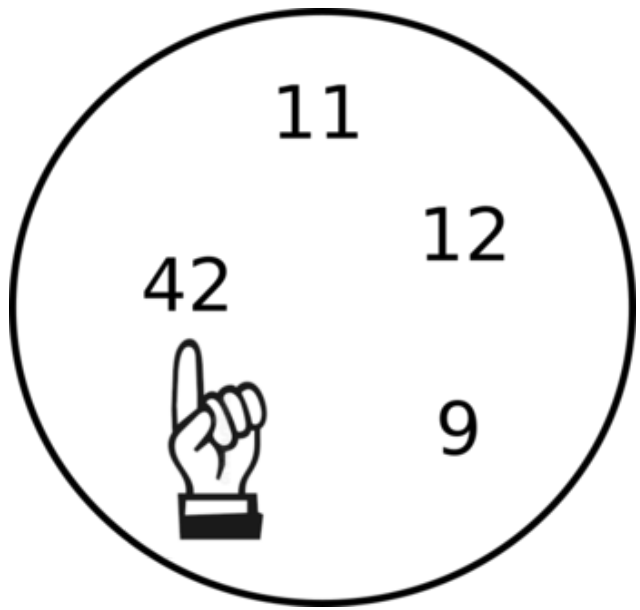
- Supongamos que la operación fin es destructiva:
  `void Lista<T>::sacarPrimero() { ... }`
- ¿Qué pasa con la complejidad?

Iteradores

Un **iterador** es una manera abstracta de recorrer colecciones, independientemente de su estructura.

Informalmente

$$\text{iterador} = \text{colección} + \text{dedo}$$



Iteradores

Operaciones con iteradores:

- ¿Está posicionado sobre un elemento?
- Obtener el elemento actual.
- Avanzar al siguiente elemento.
- Retroceder al elemento anterior. *(Bidireccional)*
- Modificar el valor del elemento actual. *(Mutable)*

Iteradores en C++

Tipos

Si `T` es un tipo de colección:

- `T::iterator`, `T::const_iterator`
  Tipo de los iteradores mutables e inmutables.
  Por ejemplo `vector<int>::iterator` es un tipo.
- `T::value_type`
  Tipo de los elementos que almacena la colección.
  Por ejemplo `vector<int>::value_type` es `int`.

Creación de iteradores

Si `col` es una colección de tipo `T`:

- `col.begin()`
  Iterador posicionado sobre el primer elemento de la colección.
- `col.end()`
  Iterador posicionado sobre el final de la colección.
  (Después del último elemento).

Iteradores en C++

Operaciones con iteradores

Si `it` es de tipo `T::iterator` o `T::const_iterator`:

- `*it` — Obtiene el elemento actual.
  Si `it` es un `T::iterator`, es un lvalue.
  Si `it` es un `T::const_iterator`, no es un lvalue.
- `it->campo` — Equivalente a `(*it).campo`.
- `++it` — Avanza al siguiente elemento.
- `--it` — Retrocede al elemento anterior.
- ...

Iteradores en C++

Operaciones de la colección usando iteradores

- `T::iterator T::insert(T::iterator pos, const T::value_type& elem)`

  Inserta un elemento en la posición indicada.
- `T::iterator T::erase(T::iterator pos)`

  Elimina el elemento en la posición indicada.
- ...

Iteradores en C++

Ejemplo (recorrer)

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

vector<int>::iterator it = v.begin();
while (it != v.end()) {
  cout << *it;
  ++it;
}

for(vector<int>::iterator it = v.begin(); it != v.end(); ++it){
  cout << *it;
}
```

Iteradores en C++

Ejemplo (eliminar)

```
vector<int> v = {1, 2, 3, 4};
vector<int>::iterator it = v.begin();
it += 2;
v.erase(it);                                    // 1 2 4
```

Iteradores en C++

Ejemplo (insertar)

```
vector<int> v = {1, 2, 3, 4};
vector<int>::iterator it = v.end();
--it;
v.insert(it, 10);                                   // 1 2 3 10 4
```

Iteradores en C++

Muchas veces el compilador puede inferir los tipos:

Ejemplo (auto)

```
vector<int> v = {1, 2, 3, 4};
auto it = v.end();
--it;
v.insert(it, 10);
```

(No abusar de esta funcionalidad).

Iteradores en C++

Ejemplo (iteradores mutables vs. inmutables)

```
void mostrar(const vector<int>& v) {
  for (vector<int>::iterator it = v.begin();
       it != v.end(); ++it) {
    cout << *it;
  }
}

int main() {
  vector<int> v{1, 2, 3, 4};
  mostrar(v);
  return 0;
}
```

Iteradores en C++

Ejemplo (iteradores mutables vs. inmutables)

```
void mostrar(const vector<int>& v) {
  for (vector<int>::iterator it = v.begin();
       it != v.end(); ++it) {
    cout << *it;
  }
}

int main() {
  vector<int> v{1, 2, 3, 4};
  mostrar(v);
  return 0;
}
```

```
In function void mostrar(const std::vector<int>&):
conversion from std::vector<int>::const_iterator [...]
to non-scalar type std::vector<int>::iterator [...]
```

Iteradores en C++

Ejemplo (iteradores mutables vs. inmutables)

```
void mostrar(const vector<int>& v) {
  for (vector<int>::const_iterator it = v.begin();
       it != v.end(); ++it) {
    cout << *it;
  }
}

int main() {
  vector<int> v{1, 2, 3, 4};
  mostrar(v);
  return 0;
}
```

Iteradores en C++

Ejemplo (for basado en rangos)

```
void mostrar(const vector<int>& v) {
  for (int x : v) {
    cout << x;
  }
}

int main() {
  vector<int> v{1, 2, 3, 4};
  mostrar(v);
  return 0;
}
```

Iteradores en C++

Ejemplo (for basado en rangos)

```
void mostrar(const vector<int>& v) {
  for (int x : v) {
    cout << x;
  }
}

int main() {
  vector<int> v{1, 2, 3, 4};
  mostrar(v);
  return 0;
}
```

- En general se acepta la sintaxis `for (T x : col)` siempre que `col` sea una colección con la interfaz de iteradores descripta arriba.

Algoritmos genéricos

Recibiendo una colección genérica:

```
template<class Coleccion>
bool pertenece(const Coleccion& c,
               typename const Coleccion::value_type& x) {
  for (auto& y : c) {
    if (x == y) {
      return true;
    }
  }
  return false;
}

int main() {
  vector<int> v{1, 2, 3, 4};
  int dos = 2;
  cout << pertenece(v, dos);
}
```

(Ojo con el typename).

Algoritmos genéricos

Recibiendo un iterador genérico:

```
template<class Iterador>
bool pertenece(Iterador desde, Iterador hasta,
               typename Iterador::value_type& x) {
  for (auto it = desde; it != hasta; ++it) {
    if (x == *it) {
      return true;
    }
  }
  return false;
}

int main() {
  vector<int> v{1, 2, 3, 4};
  int dos = 2;
  cout << pertenece(v.begin(), v.end(), dos);
}
```