ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

Modificación de listas mediante referencias

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Comportamiento en memoria de at( )

Nos ha faltado la operación de actualizar un elemento de la lista.

```
class ListArray {
public:
  std::string at(int index) const {
    assert (0 <= index && index < num_elems);
    return elems[index];
  }
  ...

private:
  int num_elems;
  int capacity;
  std::string *elems;
};
```

ya que es un array de strings

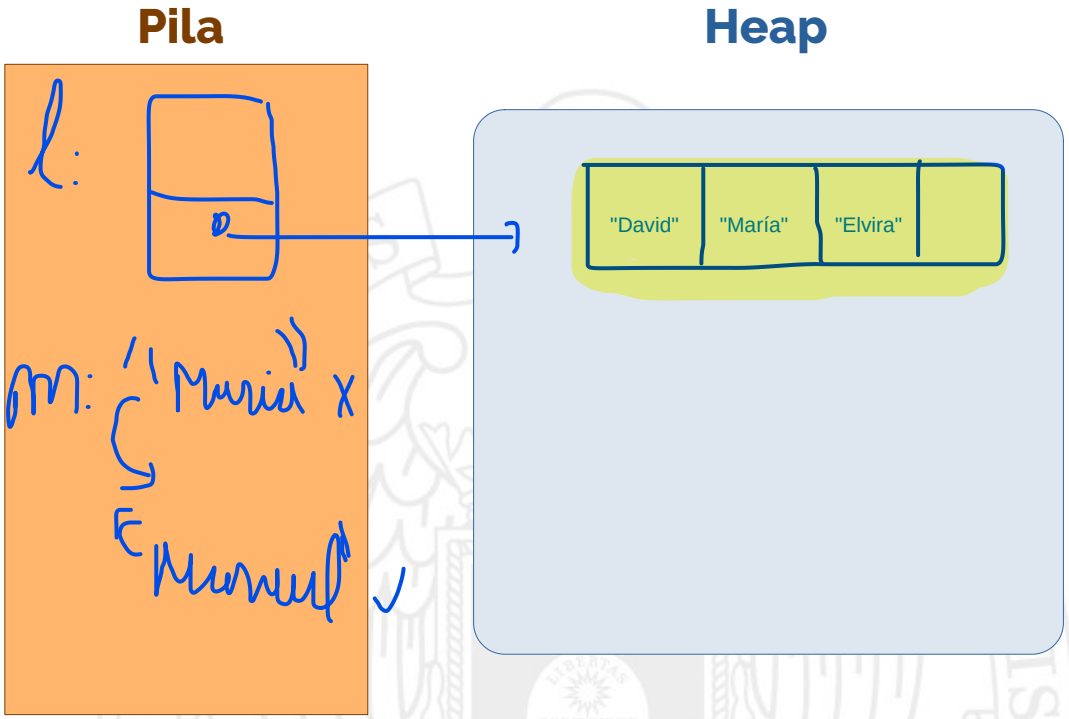
obtiene el elemento de la lista que se encuentra en la posición indicada como parámetro

Comportamiento en memoria de at( )

```
ListArray l;
l.push_back("David");
l.push_back("Maria");
l.push_back("Elvira");

std::string m = l.at(1);

m = "Manuel"
l.display();
```

pero no se modifica l. Tiene sentido

```
[David, Maria, Elvira]
```



Comportamiento en memoria de at( )

```
class ListArray {
public:

  std::string & at(int index) const {
    assert (0 <= index && index < num_elems);
    return elems[index];
  }
  ...

private:
  int num_elems;
  int capacity;
  std::string *elems;
};
```

Ahora devuelve una referencia a un String &

Él ha tachado el const.

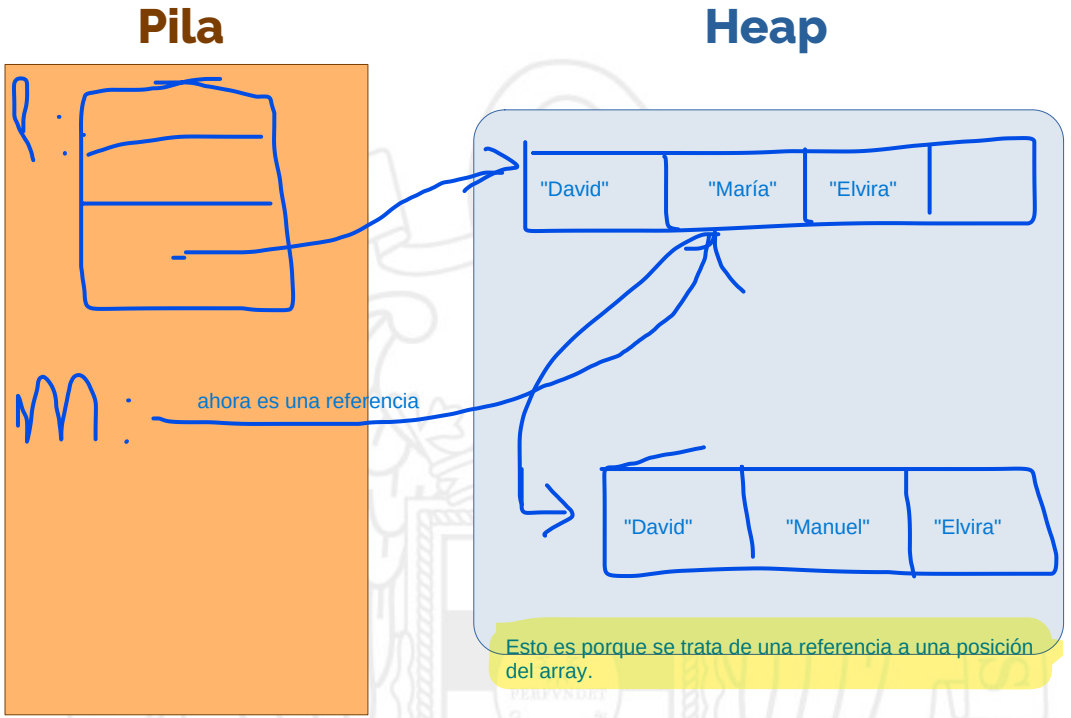
Misma implementación del método y vamos a ver cómo se comporta la esta nueva versión

¿Y si at( ) devolviese una referencia?

```
ListArray l;
l.push_back("David");
l.push_back("Maria");
l.push_back("Elvira");

std::string &m = l.at(1);
m = "Manuel"

l.display();
```

ahora m contiene una referencia

```
[David, Manuel, Elvira]
```



¿Y si at( ) devolviese una referencia?

```
ListArray l;
l.push_back("David");
l.push_back("Maria");
l.push_back("Elvira");

std::string &m = l.at(1);
m = "Manuel"

l.display();
```

equivale a

```
l.at(1) = "Manuel";
```

Por eso no era const porque si que altera el estado del objeto inicial, en este caso del array.

Consecuencias

- Haciendo que `at()` devuelva una referencia al elemento del array permitimos la posibilidad de **actualizar** elementos de la lista, sin necesidad de necesitar un método específico para ello.

Esto es bueno

- Pero, a cambio, la función ha dejado de ser `const`.

Esto no es del todo bueno, ya que no podemos utilizar at como queramos.

- Por ejemplo, la siguiente función dejaría de ser aceptada por el compilador:

referencia constante

```
int contar_caracteres(const ListArray &l) {
  int suma = 0;
  for (int i = 0; i < l.size(); i++) {
    suma += l.at(i).length();
  }
  return suma;
}
```

No puede llamarse a `at()`, porque `l` es una referencia constante.

como at no es un método const, y l es una referencia constante a ListArray, no puede llamarse a at(). at() debería de ser un método constante (const.).

Solución: dos versiones para at( )

```
class ListArray {
public:

  const std::string & at(int index) const {
    assert (0 <= index && index < num_elems);
    return elems[index];
  }


  std::string & at(int index) {
    assert (0 <= index && index < num_elems);
    return elems[index];
  }

  ...

};
```

utilizar dos versiones

NO SE PUEDE MODIFICAR DESDE FUERA.

Versión constante

constante y devuelve una referencia constante.

SI SE PUEDE MODIFICAR DESDE FUERA.

Versión no constante

no constante y devuelve una referencia NO constante

TENER AMBAS VERSIONES IMPLEMENTADAS SEGÚN LO QUE DICE EL TEACHER.
C++ SABE A CUAL LLAMAR EN FUNCIÓN DEL CONTEXTO.

Solución: dos versiones para at( )

L ES UNA REFERENCIA CONSTANTE

```
int contar_caracteres(const ListArray &l) {
  int suma = 0;
  for (int i = 0; i < l.size(); i++) {
    suma += l.at(i).length();
  }
  return suma;
}
```

Se llama a la versión constante de at( )

```
  ListArray l;
  ...
  l.at(1) = "Manuel";
```

Se llama a la versión no constante de at( )

Referencias en front( ) y back( )

Lo mismo que hemos hecho en at hacerlo en estos métodos.

```
const std::string & front() const {
  assert (num_elems > 0);
  return elems[0];
}

std::string & front() {
  assert (num_elems > 0);
  return elems[0];
}

const std::string & back() const {
  assert (num_elems > 0);
  return elems[num_elems - 1];
}

std::string & back() {
  assert (num_elems > 0);
  return elems[num_elems - 1];
}
```

Versión constante de front y de back

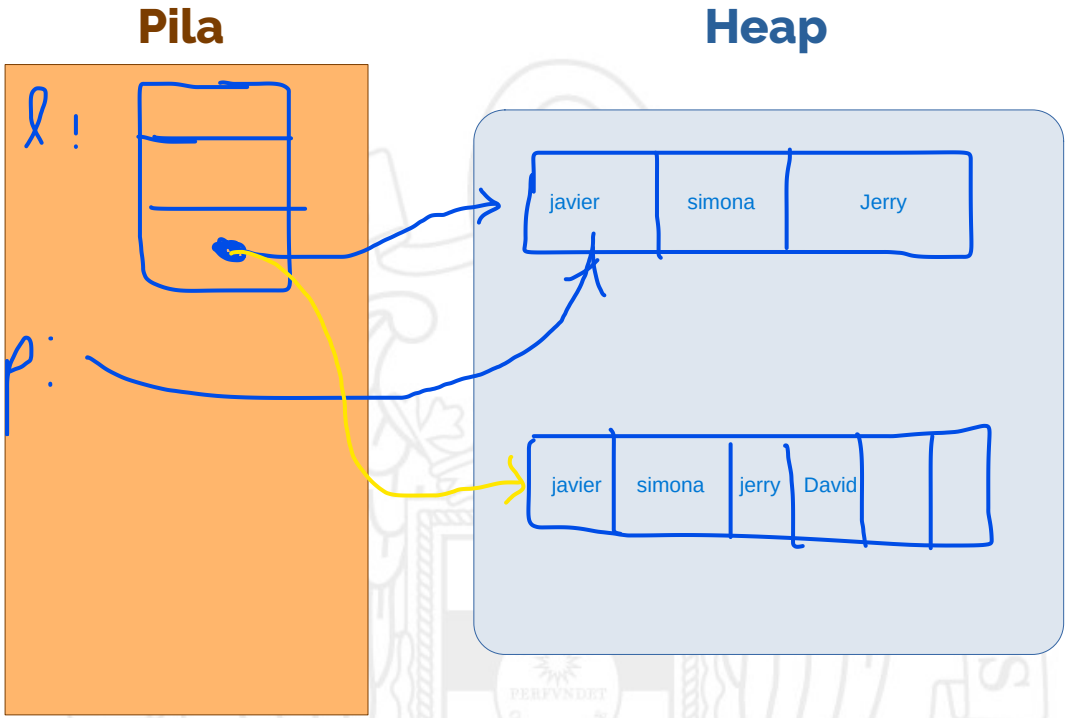
¡Cuidado con las referencias!

```
int main() {
  ListArray l(3); capacidad de 3 posiciones
  l.push_back("Javier");
  l.push_back("Simona");
  l.push_back("Jerry");

  std::string &primero = l.front();

  l.push_back("David");

  primero = "Javier Francisco";

  return 0;
}
```

pongo "p" en vez de primero

Modificamos una región de memoria que ya no nos pertenece porque ya la hemos liberado por haber tenido que redimensionar el array

Esto hay que evitarlo.



¡Cuidado con las referencias!

- Si se obtiene una referencia a un elemento de la lista, debe hacerse uso de esa referencia (para leer o modificar el valor apuntado por la referencia) **antes** de añadir o eliminar otros elementos de la lista.

```
l.front() = "Javier Francisco";
```

referencia al primer elemento y luego la modifico

obtengo referencia

```
std::string &primero = l.front();
...
primero = "Javier Francisco";
```

la modifico más adelante.