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( )

```
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;
};
```

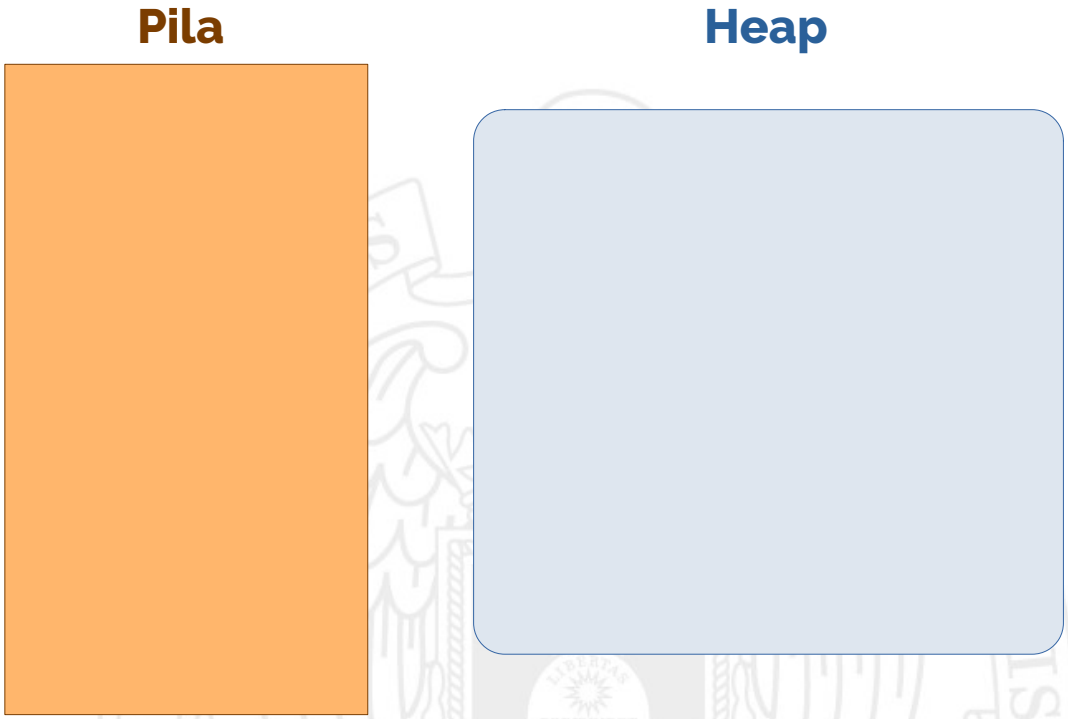
Comportamiento en memoria de at( )

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

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

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

```
[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;
};
```

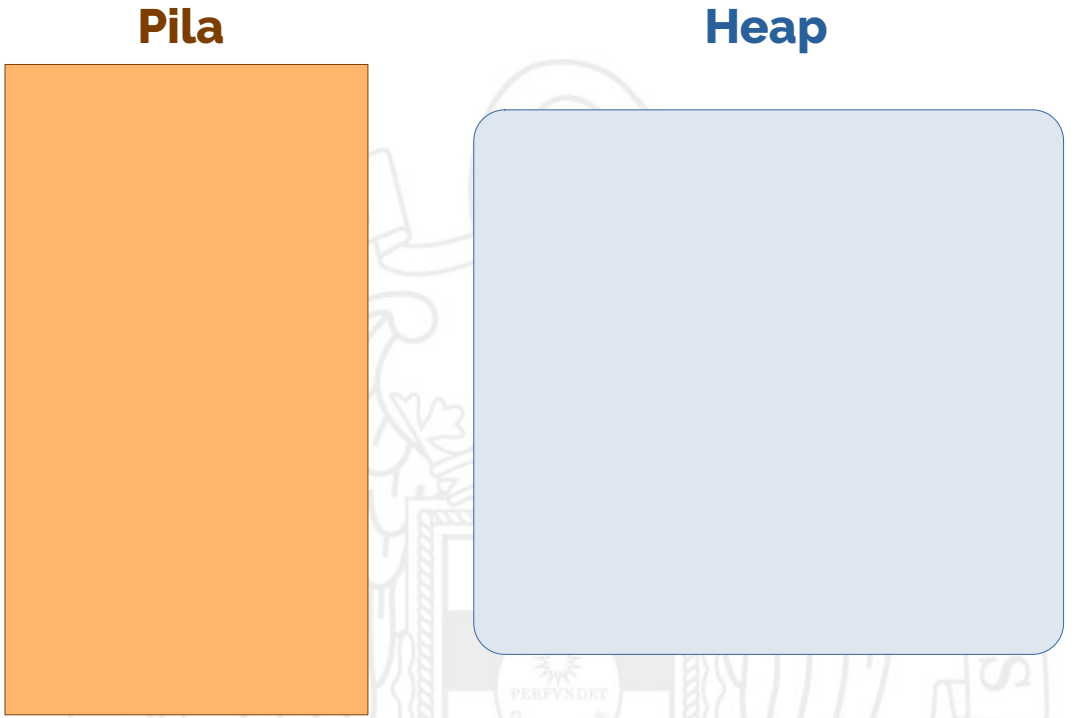
¿Y si at( ) devolviese una referencia?

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

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

l.display();
```

```
[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";
```

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.

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

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

```
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.

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];
  }

   ...

};
```

Versión constante

Versión no constante

Solución: dos versiones para at( )

```
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()`

```
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];
}
```

¡Cuidado con las referencias!

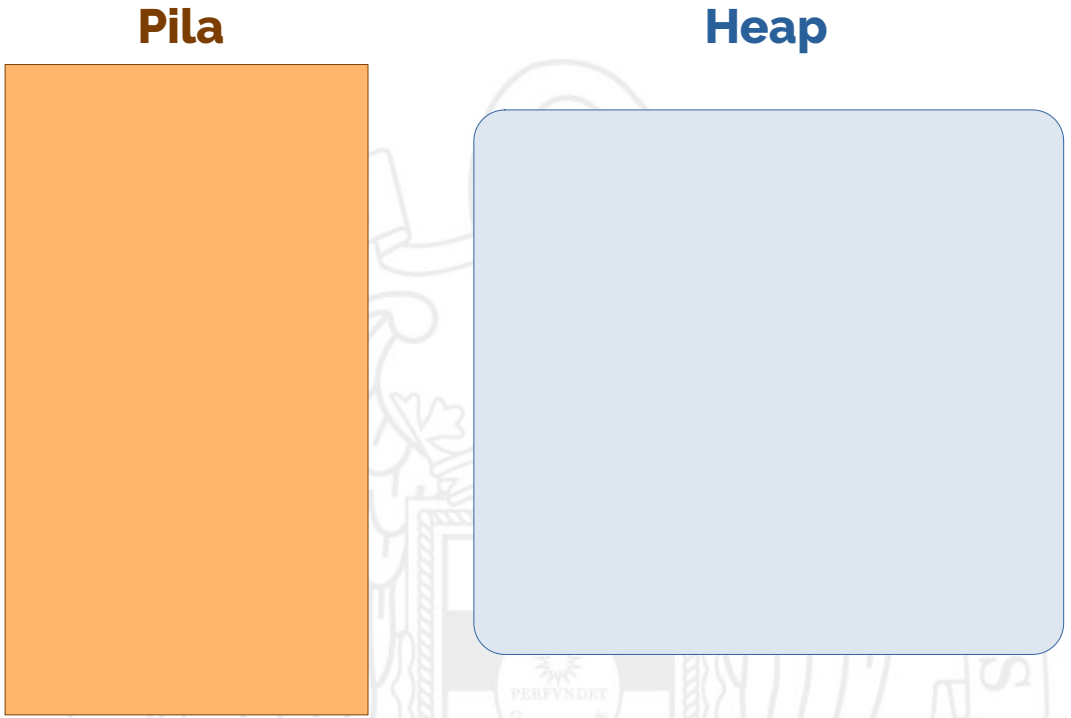
```
int main() {
  ListArray l(3);
  l.push_back("Javier");
  l.push_back("Simona");
  l.push_back("Jerry");

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

  l.push_back("David");

  primero = "Javier Francisco";

  return 0;
}
```



¡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";
```

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