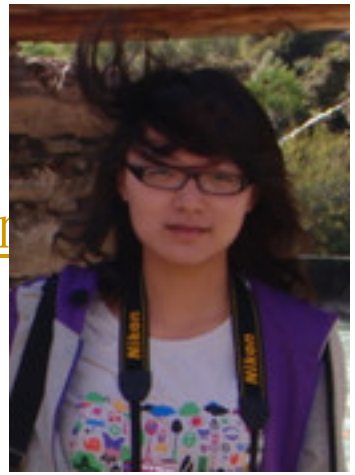
VE281

Data Structures and Algorithms

Pointers and Arrays;
Dynamic Memory Management

Teaching Assistant

- Tang, Biman (唐碧蔓)
- Email: DSAAfall2012@gmail.com
- Cell phone: 15216714434



Review

- Asymptotic Algorithm Analysis
  - Big-oh, big-omega, and theta notation
  - Common functions and their growth rate
- Analyzing Time Complexity of Programs
  - Loop statement
- Recap of Arrays and Pointers

Outline

- Pointers and Arrays
- Dynamic Memory Management
- Example: A Linear List Class

A Recap of Pointers

- Declaration: `int *bar;`
- Assigning address: `bar = &foo;`
  - The environment we get when we do this is:

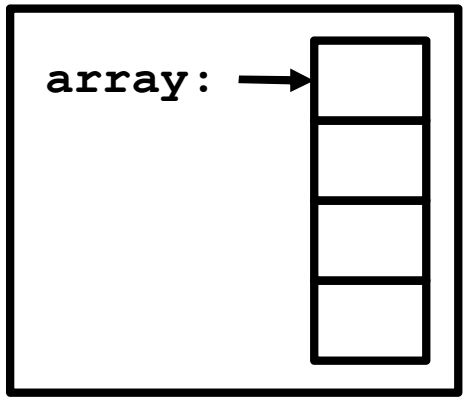
- Dereference: `*bar = 2;`
- Pointers as function arguments

```
void add_one(int *x) {
    *x = *x + 1;
}
```

Pointers and Arrays

- If you were to look at the **value** of the variable `array` (not `array[0]`) you'd find that it was exactly the same as the **address** of `array[0]`.
- In other words,

`(array==&array[0])` → `True`



Pointer Arithmetic

Enabling Array Traversal

```
int strlen(char *s)
  // REQUIRES: s is a NULL-terminated C-string
  // EFFECTS: returns the length of s, not
  // counting the NULL.
```

- We can implement **strlen** using only pointers and pointer arithmetic.

```
int strlen(char *s) {
    char *p = s;
    while (*p) {
        p++;
    }
    return (p - s);
}
```

Pointer Arithmetic

Enabling Array Traversal

```
int strlen(char *s) {
    char *p = s;
    while (*p) {
        p++;
    }
    return (p - s);
}
```

- Detailed explanation:
  - *p evaluates to “false” if p points to a NULL, true otherwise.
  - p++ advances by “one character”.
  - p-s computes the “number of characters” between p and s, which happens to be the

Common Bugs of Arrays

- Out-of-bound access, including
  - index variable not initialized
  - off-by-one error

What's the bug?

```
int y[4]={0,1,2,3};
int i;
cout << y[i] << endl;
```

Index variable not initia

```
const int size = 5;
int x[size];
for(int i=0; i<=size; i++)
  x[i] = i*2;
```

Off-by-one error

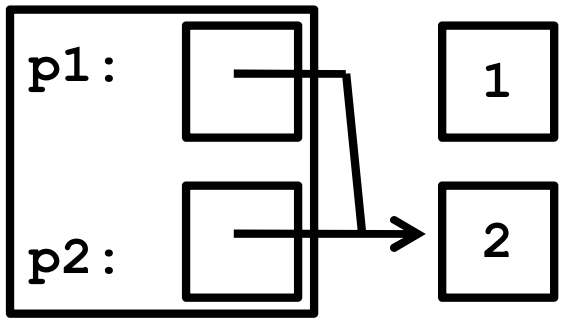
Dynamic Memory Allocation

- **new** and **delete**
- Common error: Memory leak
- Example:

```
int *p1 = new int(1);
int *p2 = new int(2);
p1 = p2;
```

Any problem?

  - This leaves us with:



There is no way to release the memory occupied by "1".

Dynamic Arrays

- Allocation

  `int *ia = new int[5];`

  Creates an array of five integers, and stores a pointer to the first element of that array in ia.

- Freeing an array

  `delete[] ia;`

  - Caution: for freeing array, use **`delete[]`** instead of the plain **`delete`**!

Outline

- Pointers and Arrays
- Dynamic Memory Management
- Example: A Linear List Class

Example: A Linear List Class

- $L = (e_0, e_1, \ldots, e_{N-1})$
- A list of integers.

- Operations

1. `int size() // return the size of the list`

   Example: `L1 = (1, 2, 3)`

   `L1.size() = 3`

2. `bool query(int v) // return true if v is in`
   `// the list; false otherwise`

Operations of the Linear List Class

```
3. void insert(int i, int v) // if 0 <= i <= N
   // (N is the size of the list), insert v at
   // position i; otherwise, throws BoundsError
   // exception.
```

Example: **L1 = (1, 2, 3)**

L1.insert(0, 5) = (5, 1, 2, 3);

L1.insert(1, 4) = (1, 4, 2, 3);

L1.insert(3, 6) = (1, 2, 3, 6);

L1.insert(4, 0) throws BoundsError

Operations of the Linear List Class

```
4. void remove(int i) // if 0 <= i < N (N is
   // the size of the list), remove the i-th
   // element; otherwise, throws BoundsError
   // exception.
```

Example: **L2 = (1, 2, 3)**

L2.remove(0) = (2, 3);

L2.remove(1) = (1, 3);

L2.remove(2) = (1, 2);

L2.remove(3) throws BoundsError

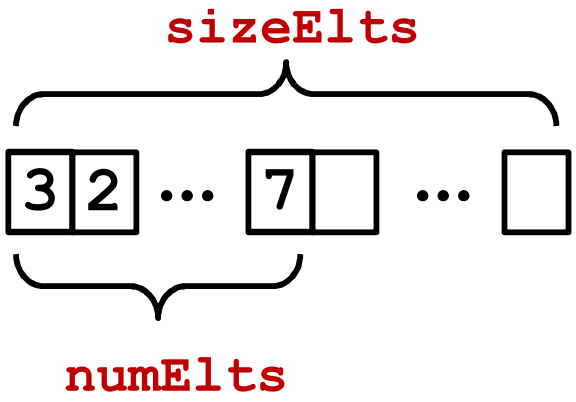
Implementation of Linear List Class

- A dynamic array-based implementation:

```
class List
{
  // OVERVIEW: an list of integers
  // with |size| <= sizeElts
  int *elts;     // pointer to dynamic array
  int sizeElts; // capacity of array
  int numElts;  // current occupancy
  ...
};
```

By default, these are private

Question: Why make them private?



Implementation of Linear List Class

- Constructor with default arguments

```
class List
{
  int *elts;     // pointer to dynamic array
  int sizeElts; // capacity of array
  int numElts;  // current occupancy
public:
  List(int size = MAXELTS);
  // EFFECTS: creates an array with specified
  // capacity. It defaults to MAXELTS if not
  // supplied.
...
};
```

Function Header for Constructor and Initialization Syntax

- Function Header of the Constructor

```
List(int size = MAXELTS);
```

  - The name of the function is the same as the name of the class.
  - This function doesn't have a return type.

- Initialization Syntax

```
List::List(int size):
  elts(new int[size]), sizeElts(size),
  numElts(0)
{
}
```

Destructor

- To prevent memory leak, we need to define a destructor

```
class List {
  int *elts;     // pointer to dynamic array
  int sizeElts; // capacity of array
  int numElts;  // current occupancy
public:
  List(int size = MAXELTS);
  // EFFECTS: creates an array with specified
  // capacity. It defaults to MAXELTS.
  ~List(); // Destro
...
};

List::~List(){
    delete[] elts;
}
```

Note that we have to use the array-based delete operator, not the "standard" delete operator, because the thing we are delete[]ing was created by new[]

Dynamic Arrays

Group Exercise

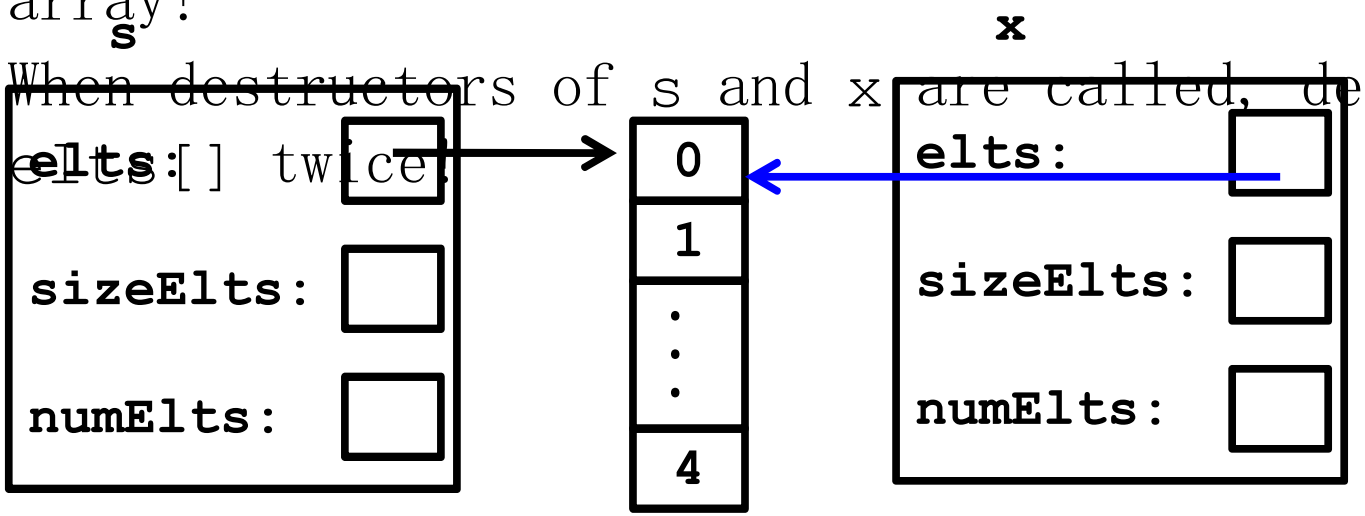
- Question: What happens in the following code?
- Hint: Classes are passed by-value, just like structs. They are also bitwise-copied, just like structs!

```
void foo(List x) {
  // do something
}
int main() {
  List s;
  s.insert(0,5);
  foo(s);
  s.query(5);
}
```

Dynamic Arrays

Fixing dangling pointers

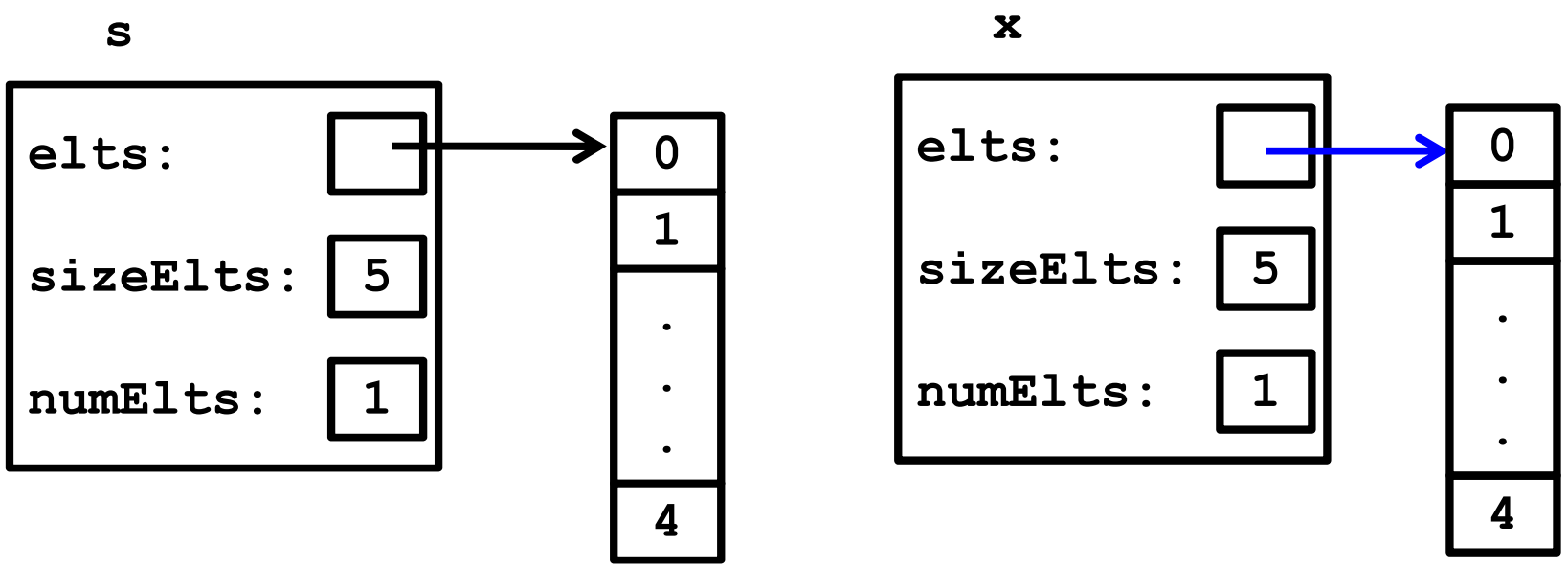
- So, what's the problem?
- The semantics of pass-by-value arguments specify that we should copy the contents of s to x, but unfortunately, only pointer value of elts is copied, not the array elts[].
- The two objects end up sharing the same elts[] array!
- When destructors of s and x are called, delete elts[] twice!



Dynamic Arrays

Fixing dangling pointers

- What we really want is to copy the entire **array**.



Dynamic Arrays

Shallow Copy versus Deep Copy

- When a class contains pointers to dynamic elements, copying it is tricky.
- If we just copy the "members of the class", we get a shallow copy.
- Usually, we want a **full** copy of **everything**. This is called a deep copy.

Copy Constructor

- We declare a copy constructor for our List class as follows:

```
class List {
  int *elts;     // pointer to dynamic array
  int sizeElts; // capacity of array
  int numElts;  // current occupancy
public:
  List(int size = MAXELTS);
  // Constructor with default arguments.
  ~List(); // Destructor
  List(const List &l); // Copy constructor
...
};
```

Function overloading. The argument is a reference to a const instance to copy from.

Copy Constructor

```
List::List(const List &l) {
  sizeElts = l.sizeElts;
  elts = new int[sizeElts];
  // Copy array
  for (int i = 0; i < l.sizeElts; i++) {
    elts[i] = l.elts[i];
  }
  // Establish numElts invariant
  numElts = l.numElts;
}
```

Shallow Copy versus Deep Copy

- What operation else could cause the shallow copy problem?
  - Assignment operator!
- We need to overload the assignment operator:

```
List &operator= (const List &l);
```

- Like the copy constructor, the assignment operator takes a reference to a const instance to copy from.
- However, it also **returns** a reference to the copied-to object.
  - **return *this;**

The Rule of the Big Three

- Specifically, if you have any dynamically allocated storage in a class, you must provide:
  - A destructor
  - A copy constructor
  - An assignment operator

- If you find yourself writing one of these, you almost certainly need all of them.

Implementation of Linear List Class

- So far, we have

```
class List {
  int *elts;     // pointer to dynamic array
  int sizeElts; // capacity of array
  int numElts;  // current occupancy
public:
  List(int size = MAXELTS);
  // Constructor with default arguments.
  ~List(); // Destructor
  List(const List &l); // copy constructor
  List &operator= (const List &l);
  // assignment operator
};
```

Maintenance methods

Operational Methods

```
void insert(int i, int v); // MODIFIES: this
// EFFECTS: If capacity is full, throws
// FullError; else if 0 <= i <= numElts
// inserts v at position i;
// else throws BoundsError.

void remove(int i); // MODIFIES: this
// EFFECTS: removes the i-th element if 0 <= i
// < numElts; throws BoundsError otherwise.

bool query(int v) const; // EFFECTS: returns true
// if v is in this, false otherwise.

int size() const; // EFFECTS: returns |this|.
```

Const Member Functions

```
int size() const;
```

- Each member function of a class has a extra, implicit parameter named **this**.
  - "**this**" is a pointer to the current instance on which the function is invoked.

- **const** keyword modifies the implicit **this** pointer: **this** is now a pointer to a const instance.
  - The member function **size()** cannot change the object on which **size()** is called.
  - By its definition, **size()** shouldn't change the object! Adding **const** keyword prevents any

Const Member Functions

- Implement **size()**

```
int List::size() const {
  return numElts;
}
```

- If a const member function calls other member functions, they must be const too!

```
void A::f() const {
  g(); // g must be a const member
       // function
}
```

Operational Methods: query

```
bool List::query(int v) const
// EFFECTS: returns true
// if v is in this, false otherwise.
{
    for (int i = 0; i < numElts; i++)
    {
        if (elts[i] == v)
            return true;
    }
    return false;
}
```

Operation Methods: insert

```
void List::insert(int i, int v) {
    if (numElts == sizeElts) throw FullError();
    if(i >= 0 && i <= numElts) {
        for(int k = i; k < numElts; k++)
            elts[k+1] = elts[k];
        elts[i] = v;
        numElts++;
    }
    else throw BoundsError();
}
```