Administrivia

- For the next lab + portfolio we will use github
- We need to link github accounts to college accounts
    1. Create a github account at https://github.com (it's free)
    2. Submit your college id and github id here:

    https://forms.office.com/Pages/ResponsePage.aspx?id=B3WJK4zudUWDC0-CZ8PTB07FeicCPhVAsezU-PfpjRxUMEdHVDc0Q0s0SU5URTBONUtZOU5JWkdRWi4u

- Accounts will be created in batches
    - Happens a couple of times a day
    - Will make sure to do a run just before the Thu lab
- You can start the lab without a github account
    - Some parts can be completed later later

Lab vs portfolio

- The labs are now explicitly linked to the portfolio
    - Each lab task is tied to a portfolio task
    - Try to get people to learn before doing

- Makes the labs a bit longer and more procedural

- Makes the portfolio a touch easier (?)

Sound recording

- Apparently it didn't work on Thursday
  - No idea why
  - There is no way to recover the audio
- I'll attempt to go back and narrate them offline
  - Probably not going to happen till Thursday
- Remember: you can't rely on panopto...

Classes and objects : recap

Objects model things with *state* and *computation*

data + functions

Methods provide a safe way to change state

If users call methods correctly

then the class ensures the object state is valid

Users should usually not manipulate state directly

-> make member variables private

Objects should be created in a valid state

-> use a ***constructor*** to ensure initial state is valid

"Plain" objects continued

- Concepts for today
    - Class invariants
    - Reasoning about code
    - Lifetimes of objects

- C++ features
    - Destructors
    - Scope resolution operator

A manual string replacement

```
struct String
{
    int length;
    int capacity;
    char *buffer;
};

String *String_create();
void    String_destroy(String *s);

int     String_size(String *s);
void    String_resize(String *s);
char    String_at(String *s);
```

```
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
    String();


    void size();
    void resize();
    char at(int index);
};
```

length : the number of valid characters in the string
capacity : the size of the buffer

```
struct String
{
    int length;
    int capacity;
    char *buffer;
};

String *String_create()
{
    String *s=new String;
    s->length=0;
    s->capacity=0;
    s->buffer= nullptr;
    return s;
}
```

```
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
    String()
    {
        length=0;
        capacity=0;
        buffer=nullptr;
    }
};
```

```
struct String
{
    int length;
    int capacity;
    char *buffer;
};

String *String_create()
{
    String *s=new String;
    s->length=0;
    s->capacity=0;
    s->buffer= nullptr;
    return s;
}
```

```
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
    String()
    {
        length=0;
        capacity=0;
        buffer=nullptr;
    }
};
```

Combined declaration and definition

```
struct String
{
    int length;
    int capacity;
    char *buffer;
};

String *String_create();
...



String *String_create()
{
    String *s=new String;
    s->length=0;
    s->capacity=0;
    s->buffer=nullptr;
    return s;
}
```

```
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
    String()
    {
        length=0;
        capacity=0;
        buffer=nullptr;
    }
};
```

```
struct String
{
    int length;
    int capacity;
    char *buffer;
};

String *String_create();
...
```

```
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
    String();
    ...
};
```

```
String *String_create()
{
    String *s=new String;
    s->length=0;
    s->capacity=0;
    s->buffer=nullptr;
    return s;
}
```

```
String::String()
{

    length=0;
    capacity=0;
    buffer=nullptr;

}
```

```
struct String
{
    int length;
    int capacity;
    char *buffer;
};

String *String_create();
...
```

```
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
    String();
    ...
};
```

Declaration

Definition

```
String *String_create()
{
    String *s=new String;
    s->length=0;
    s->capacity=0;
    s->buffer=nullptr;
    return s;
}
```

```
String::String()
{

    length=0;
    capacity=0;
    buffer=nullptr;

}
```

```
struct String
{
    int length;
    int capacity;
    char *buffer;
};

String *String_create();
...
```

```
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
    String();
    ...
};
```

Scope resolution operator ::

```
String *String_create()
{
    String *s=new String;
    s->length=0;
    s->capacity=0;
    s->buffer=nullptr;
    return s;
}
```

```
String::String()
{

    length=0;
    capacity=0;
    buffer=nullptr;

}
```

```c
struct String
{
    int length;
    int capacity;
    char *buffer;
};

String *String_create();
char String_at(String *s, int index);
...
```

```cpp
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
    String();
    char at(int index);
};
```

```c
char String_at(String *s, int index)
{
    return s->buffer[index];
}
```

```cpp
char String::at(int index);
{
    return buffer[index];
}
```

```
struct String
{
    int length;
    int capacity;
    char *buffer;
};

String *String_create();
char String_at(String *s, int index);
...
```

```
char String_at(String *s, int index)
{
    return s->buffer[index];
}
```

```
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
    String();
    char at(int index);
};
```

Same name

Different scopes

```
char String::at(int index);
{
    return buffer[index];
}
```

Classes: separate definitions

- Methods can be defined outside the class
  - Can make the class declaration much easier to read
  - *(Also allows separate source-file compilation: coming up)*

- Need to use explicit scopes for separate definitions
  - *Inside the class declaration* - all members are in scope
  - *Outside the class declaration* - use the `::` operator

- "Inline" definitions are a matter of taste
  - Some people prefer to keep decl. and defn. together
  - Others think it cleaner to separate them

```
struct String
{
    int length;
    int capacity;
    char *buffer;
};

void String_append(String *s, char c);
```

```
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
    void append(char c);
};
```

```
void String_append(String *s, char c)
{
    assert( s->length <= s->capacity );
    if(s->length == s->capacity){
        int newCapacity = max(2*s->capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<s->length; i++){
            newBuffer[i] = s->buffer[i];
        }
        delete []s->buffer;

        s->capacity = newCapacity;
        s->buffer = newBuffer;
    }

    assert( s->length < s->capacity );
    s->buffer[s->length] = c;
    s->length += 1;
}
```

```
void String::append(String *s, char c)
{
    assert( s->length <= s->capacity );
    if(s->length == s->capacity){
        int newCapacity = max(2*s->capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<s->length; i++){
            newBuffer[i] = s->buffer[i];
        }
        delete []s->buffer;

        s->capacity = newCapacity;
        s->buffer = newBuffer;
    }

    assert( s->length < s->capacity );
    s->buffer[s->length] = c;
    s->length += 1;
}
```

```cpp
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;
}
```

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;
}
```

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;
}
```

Defensive programming

```cpp
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;
}
```

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;
}
```

Un-Common case

Common case

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;
}
```

Common case

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;
}
```

Class invariant

Should always be true at start of methods

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;
}
```

Class invariant

Should always be true at start of methods

We must ensure it is ***still*** true at end of method

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

Class invariant

Should always be true at start of methods

We must ensure it is ***still*** true at end of method

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

length < capacity
and
length' = length + 1
implies
length' <= capacity

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

x' is informal shorthand for "value of *x* after the operation"

length < capacity
and
length' = length + 1
implies
length' <= capacity

```cpp
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

1 - Allocate a new buffer

Guarantee that:
  newCapacity > capacity

Have to consider the case
  capacity == 0

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

2 – Copy the existing data to new buffer, and delete old buffer

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

3 – Update the state of the object.

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );     ← Will this always be true?
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

length == capacity
and
newCapacity > capacity
and
capacity' == newCapacity
and
length' == length
implies
length' < capacity'

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

```
length == capacity
        and
newCapacity > capacity
        and
capacity' == newCapacity
        and
length' == length
      implies
length' < capacity'
```

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

length == capacity
and
newCapacity > capacity
and
capacity' == newCapacity
and
length' == length
implies
length' < capacity'

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

length == capacity
and
newCapacity > capacity
and
capacity' == newCapacity
and
length' == length
implies
length' < capacity'

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

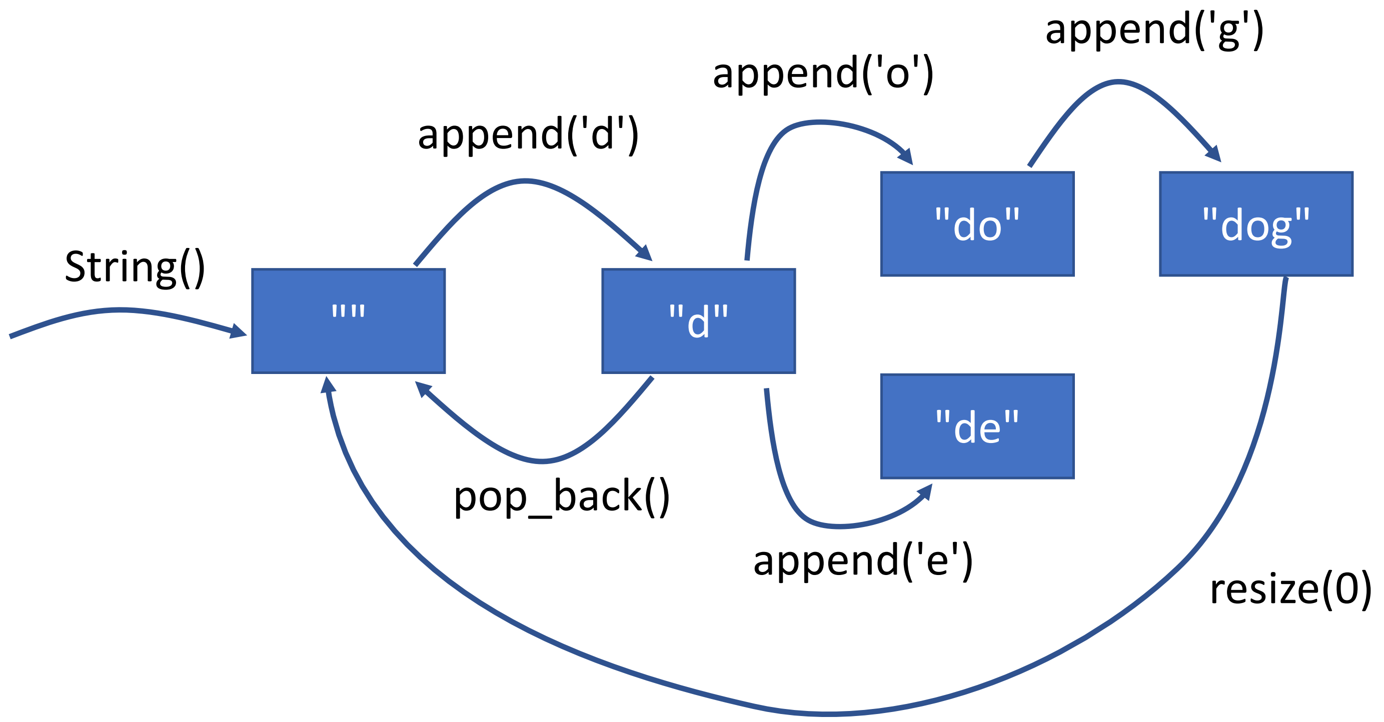
length == capacity
and
newCapacity > capacity
and
capacity' == newCapacity
and
length' == length
implies
length' < capacity'

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

length == capacity
and
newCapacity > capacity
and
capacity' == newCapacity
and
length' == length
implies
length' < capacity'

Objects: maintaining state

- Objects have both state and computation
  - Methods allow object users to move between states
  - Users should only care about publicly visible state



Objects: maintaining state

- Objects have both state and computation
  - Methods allow object users to move between states
  - Users should only care about publicly visible state

- It is often useful to think about state invariants
  - What properties or conditions should *always* be true?
  - Can these invariants be cheaply checked at run-time?
  - Can you prove to yourself the invariants hold?

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

Why double the size of the buffer?

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = capacity + 4;
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

Could just add a fixed amount

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = capacity + 4;
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

Un-Common case

Time is proportional to length

Time is fixed

Common case

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = capacity + 4;
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

```
int main()
{
    String s;

    char x;
    while(true){
        cin >> x;
        if( cin.fail() ){
            break;
        }
        s.append(x);
    }
}
```

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = capacity + 4;
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

```
int main()
{
    String s;

    for(int i=0; i<n; i+=1);
       s.append(...);
    }
}
```

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = capacity + 4;
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

Every 4th character takes time proportional to current length

Total cost:

$$\sum_{i=1}^{n} \begin{cases} i, & mod(i,4) = 0 \\ 1, & otherwise \end{cases}$$

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = capacity + 4;
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

Every 4th character takes time proportional to current length

Total cost (roughly):

$$\sim n^2$$

```cpp
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

Take time proportional to n for length=4,8,16,32,64,…

Total cost:

$$\sum_{i=1}^{n} \begin{cases} i, & if\ i = 2^p \\ 1, & otherwise \end{cases}$$

```
void String::append(char c)
{
    assert( length <= capacity );
    if(length == capacity){
        int newCapacity = max(2*capacity,4);
        char *newBuffer = new char[newCapacity];

        for(int i=0; i<length; i++){
            newBuffer[i] = buffer[i];
        }
        delete []buffer;

        capacity = newCapacity;
        buffer = newBuffer;
    }

    assert( length < capacity );
    buffer[length] = c;
    length += 1;

    assert( length <= capacity );
}
```

Take time proportional to n for length=4,8,16,32,64,…

Total cost:

$\sim n$

Objects : performance

- Objects should keep state implementation private
    - How the object manages its state should be irrelevant
    - The designer should be able to modify details of state

- Ideally you should only expose "efficient" methods
    - *Try* to design methods to encourage good usage
    - Watch out for inefficient across many method calls
    - Try to develop an intuition for what is "cheap" vs "expensive"

- ***But***: on this course we emphasise functional aspects
    - The main goal is to get it correct and working
    - Non-functional aspects like performance are secondary

```
struct String
{
    int length;
    int capacity;
    char *buffer;
};

String *String_create();

void String_destroy(String *s)
{
    delete[] s->buffer;
    delete s;
}
```

```
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
    String();

};
```

```
struct String
{
    int length;
    int capacity;
    char *buffer;
};

String *String_create();

void String_destroy(String *s)
{
    delete[] s->buffer;
    delete s;
}
```

```
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
    String();

    ~String()
    {
        delete[] buffer;
    }
};
```

```
struct String
{
    int length;
    int capacity;
    char *buffer;
};

String *String_create();

void String_destroy(String *s)
{
    delete[] s->buffer;
    delete s;
}
```

```
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
    String();

    ~String()
    {
        delete[] buffer;
    }
};
```

Tilde: ~
Same as home directory symbol
Warning: looks like a hyphen in some fonts if you're not careful

```
struct String
{
    int length;
    int capacity;
    char *buffer;
};

String *String_create();

void String_destroy(String *s)
{
    delete[] s->buffer;
    delete s;
}
```

```
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
    String();

    ~String()
    {
        delete[] buffer;
    }
};
```

Destructor

```
struct String
{
    int length;
    int capacity;
    char *buffer;
};

String *String_create();

void String_destroy(String *s)
{
    delete[] s->buffer;
    delete s;
}
```

```
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
    String();

    ~String()
    {
        delete[] buffer;
    }
};
```

?

```
struct String
{
    int length;
    int capacity;
    char *buffer;
};

String *String_create()
{
    String *s=new String;
    s->length=0;
    s->capacity=0;
    s->buffer= nullptr;
    return s;
}
```

?

```
void String_destroy(String *s)
{
    delete[] s->buffer;
    delete s;
}
```

?

```
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
    String()
    {
        length=0;
        capacity=0;
        buffer= nullptr;
    }


    ~String()
    {
        delete[] buffer;
    }
};
```

Objects : creation and destruction

- Objects do not manage their own storage
  - They are supposed to provide new types
  - Does an **`int`** manage it's own storage?
- Objects can be created in different ways
  - Allocated as a local variable
  - Created as a parameter
  - Dynamically allocated using new
  - Contained within a vector or list
- Objects ***can*** manage the storage of their members
  - An important use of objects is to hide raw new/delete

Objects : destruction time!

- Destructors are called when the object is destroyed
  - When a local variable's lifetime ends
  - When delete is called on an instance created with new
  - When the vector that contains it is destroyed
  - …

```
int main()
{
  String s;
  s.append('x');
}
```

```
int main()
{
  if(condition()){
    String s;
    s.append('z');
  }
}
```

```
void print(String s)
{
  cout << s.at(0);
}
```

Objects : destruction time!

- Destructors are called when the object is destroyed
  - When a local variable's lifetime ends
  - When delete is called on an instance created with new
  - When the vector that contains it is destroyed
  - …

```
int main()
{
   String *s = new String;
   s->append('x');
   delete s;
}
```

Constructor called 8 times

```
int main()
{
    String *s = new String[8];
    s[3].append('x');
    delete[] s;
}
```

Destructor called 8 times

Objects : destruction time!

- Destructors are called when the object is destroyed
  - When a local variable's lifetime ends
  - When delete is called on an instance created with new
  - When the vector that contains it is destroyed
  - ...

- Many classes do not need destructors
  - Destructors of member variables are called automatically
  - If you rely on vector, string, list, ... they will handle it all
  - The main case for destructors is calling `delete`

Copying objects

Values are often copied

```
String plural(String v)
{
    v.append('s');
    return v;
}
```

Copying objects

Values are often copied

```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```

Copying objects

Values are often copied

```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```

Parameter is copied

Copying objects

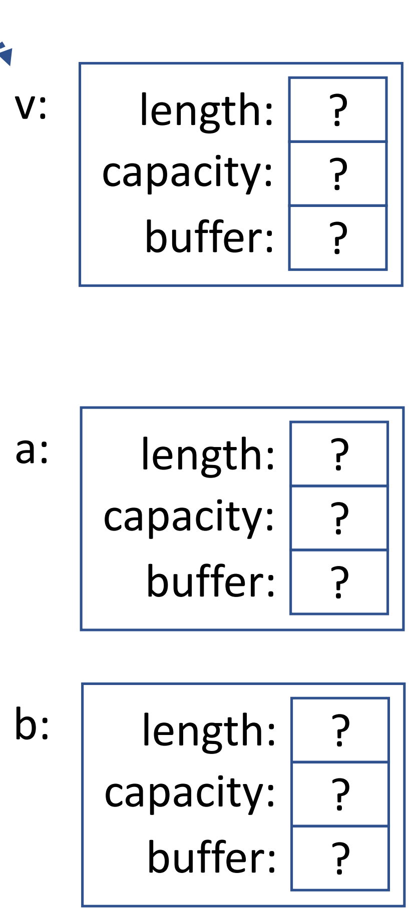
Values are often copied

```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```

Return value is copied

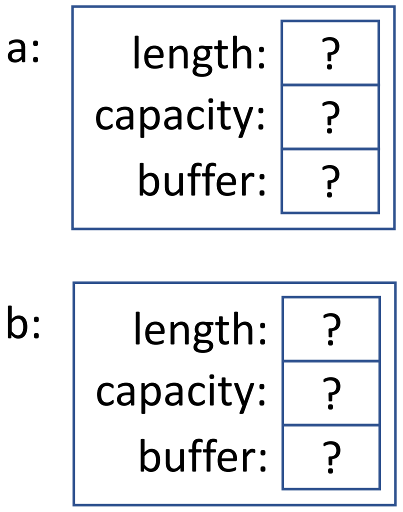
Copying objects

```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```



Copying objects

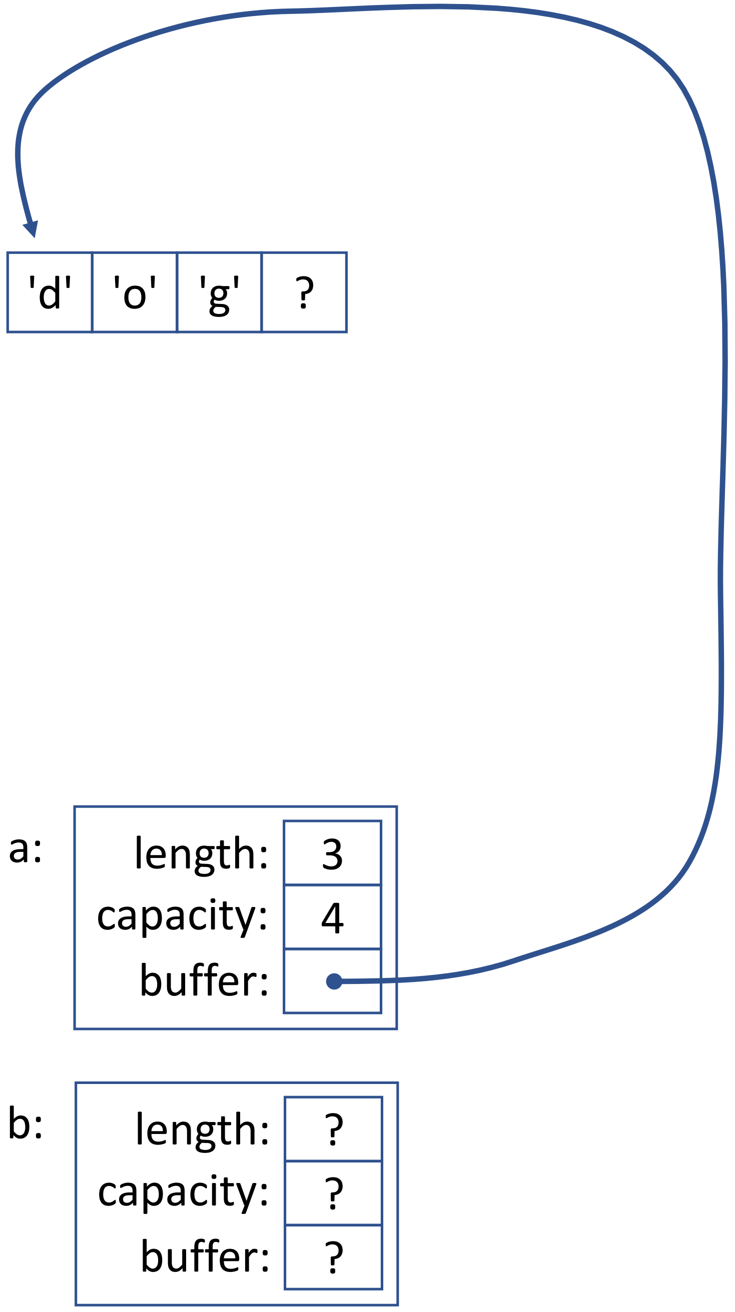
```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```



Copying objects

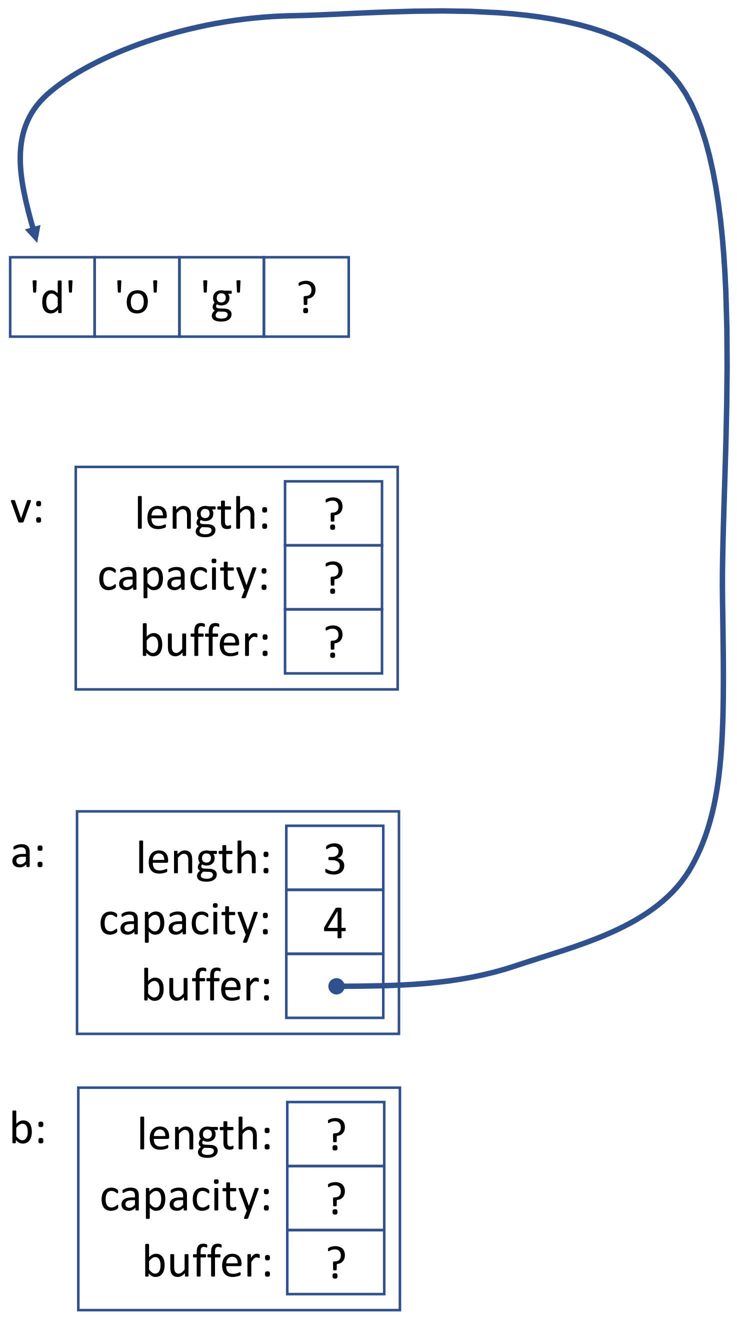
```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");
```

```
    String b = plural( a );

    cout << b.at(0) << endl;
}
```



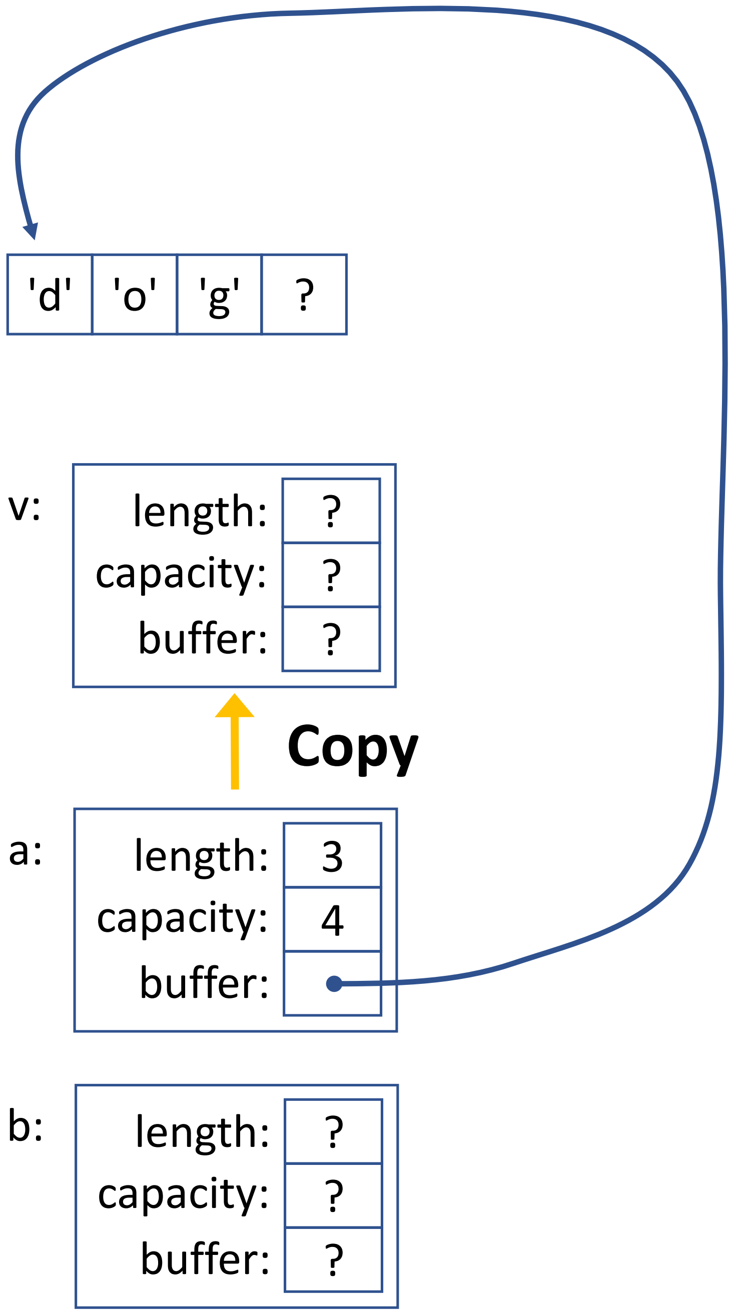
Copying objects

```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```



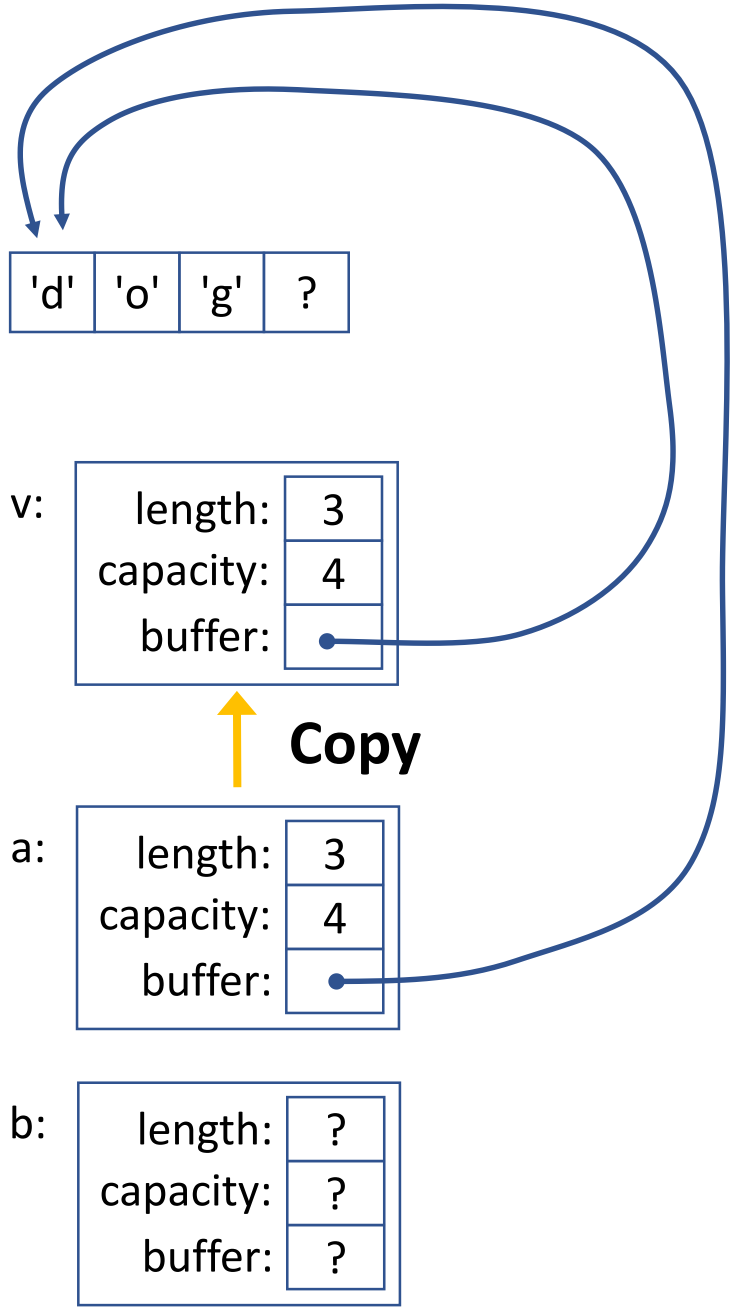
Copying objects

```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```



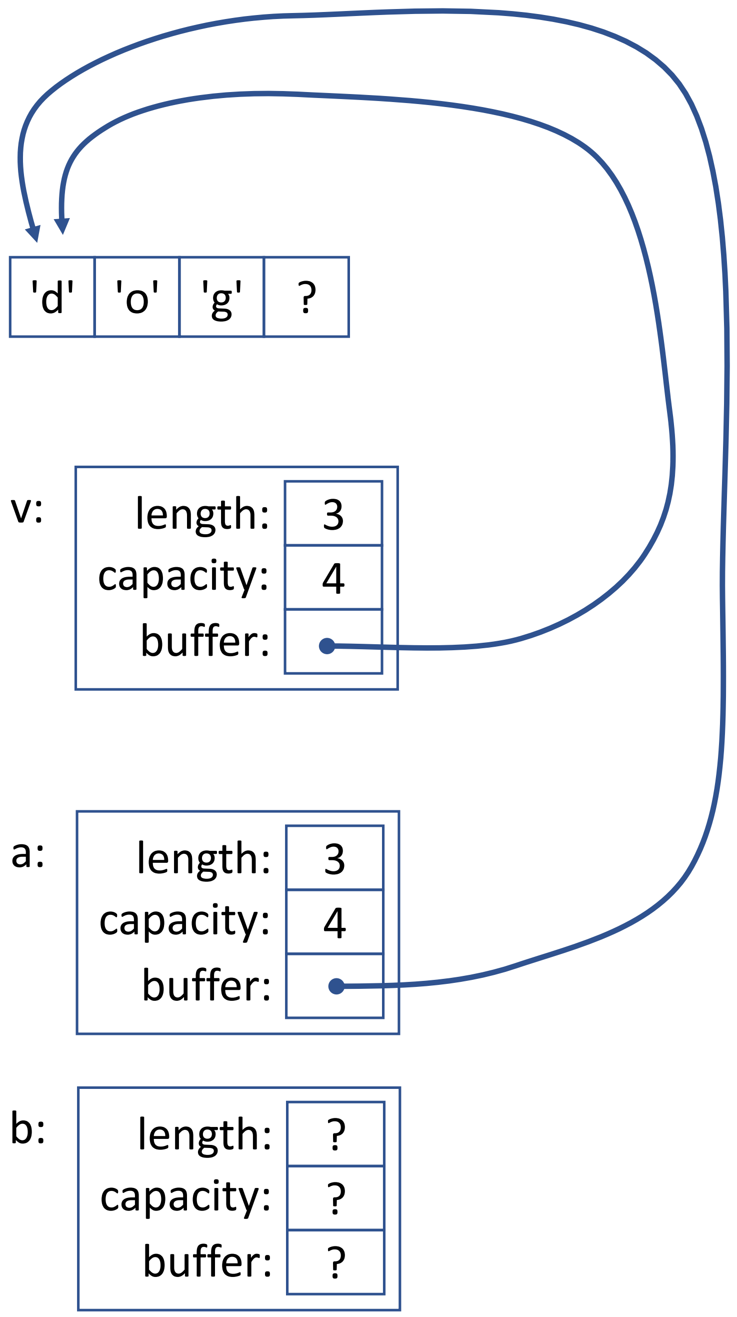
Copying objects

```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```



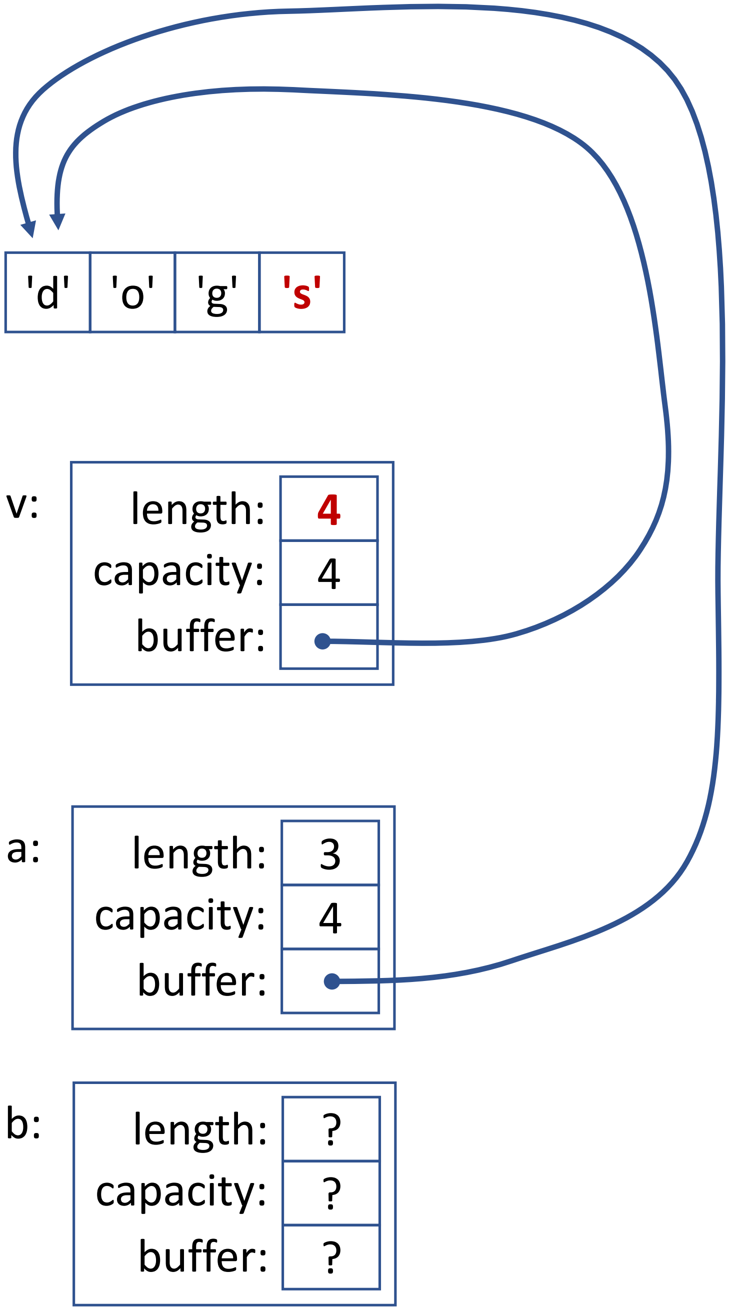
Copying objects

```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```



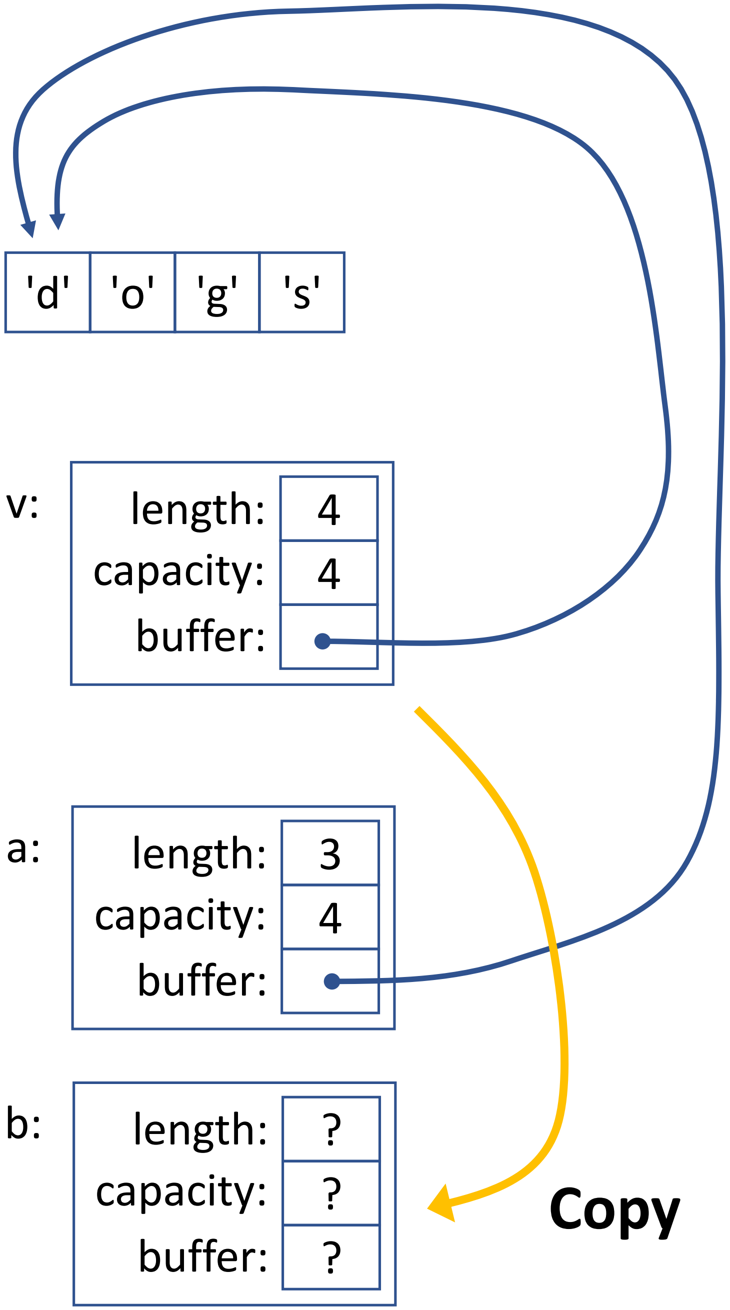
Copying objects

```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```



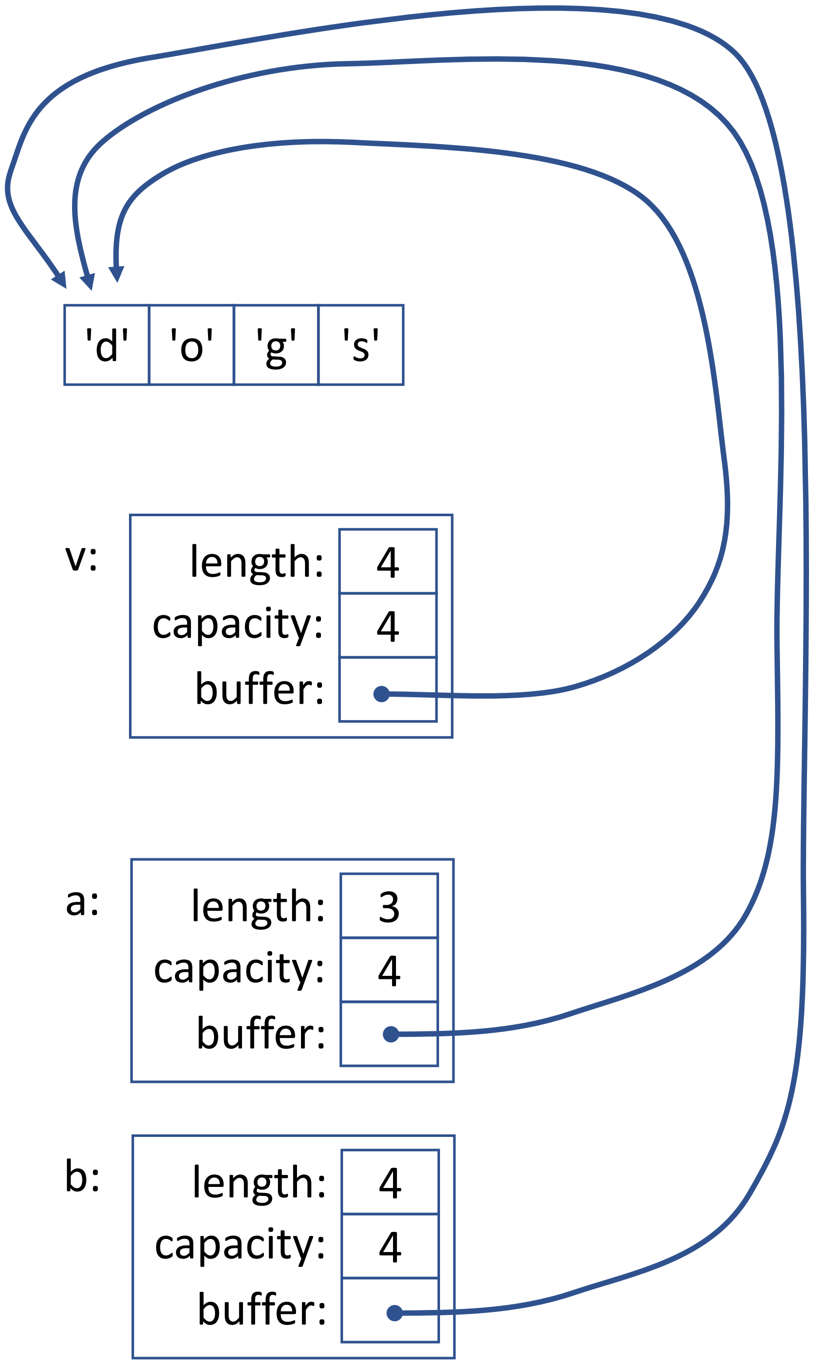
Copying objects

```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```



Copying objects

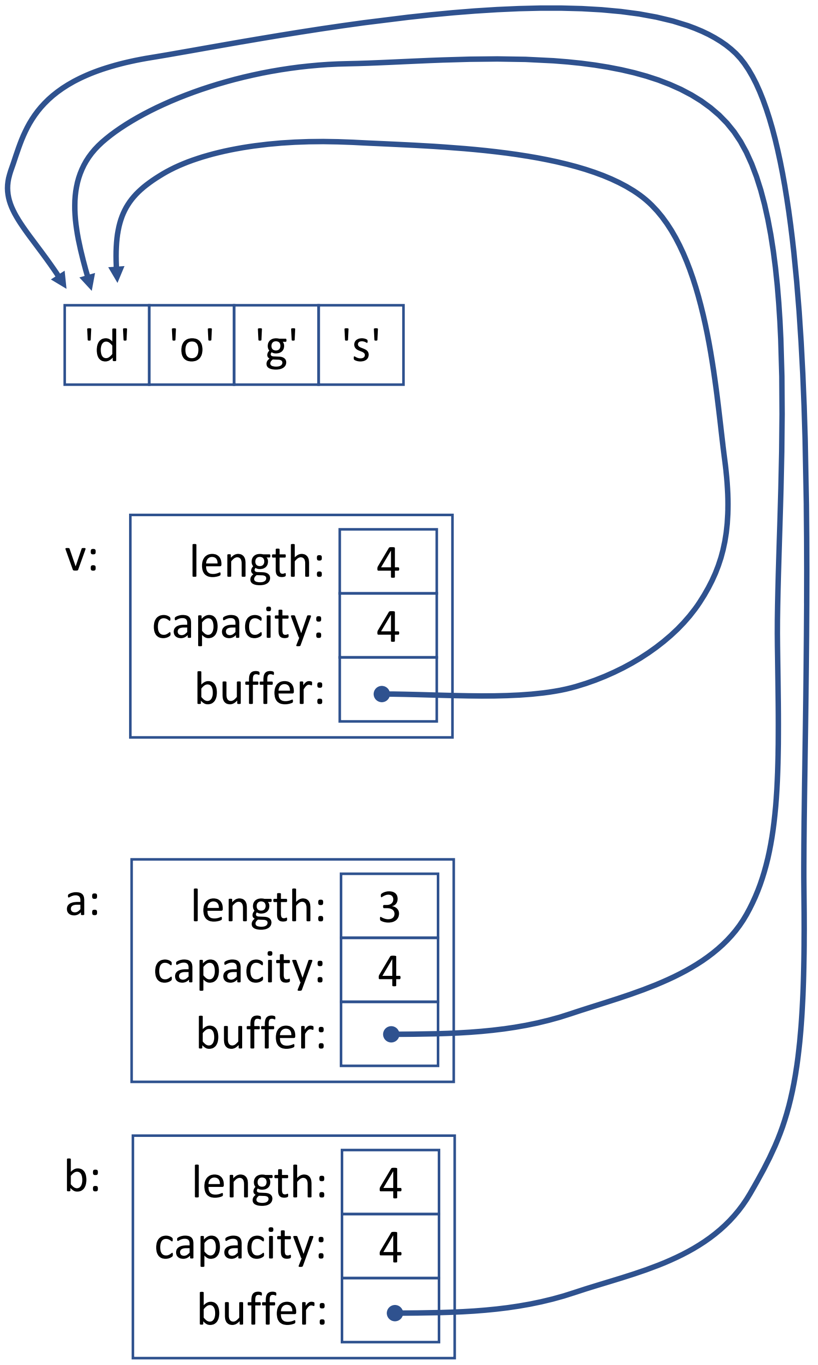
```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```



Copying objects

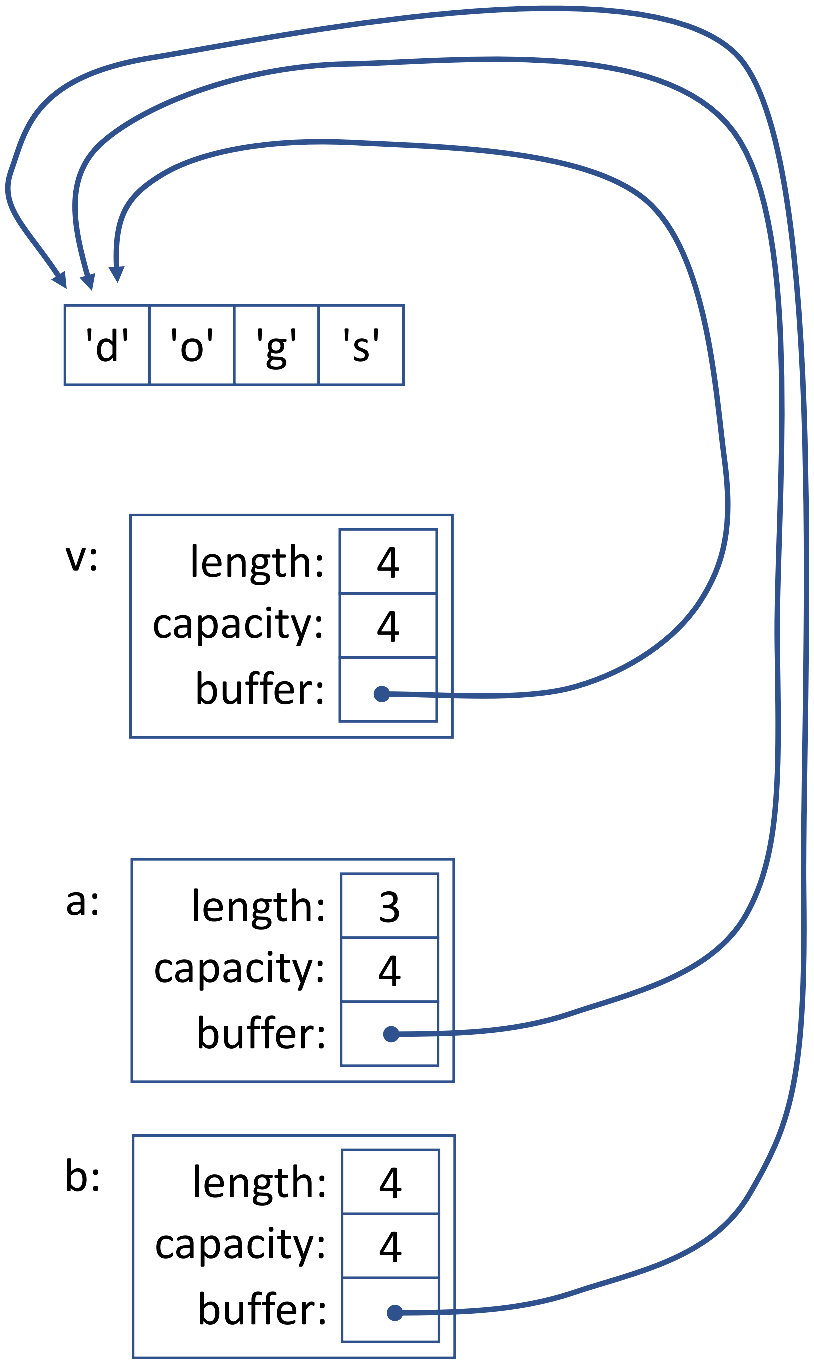
Parameter v is going out of scope and will be destroyed

```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```



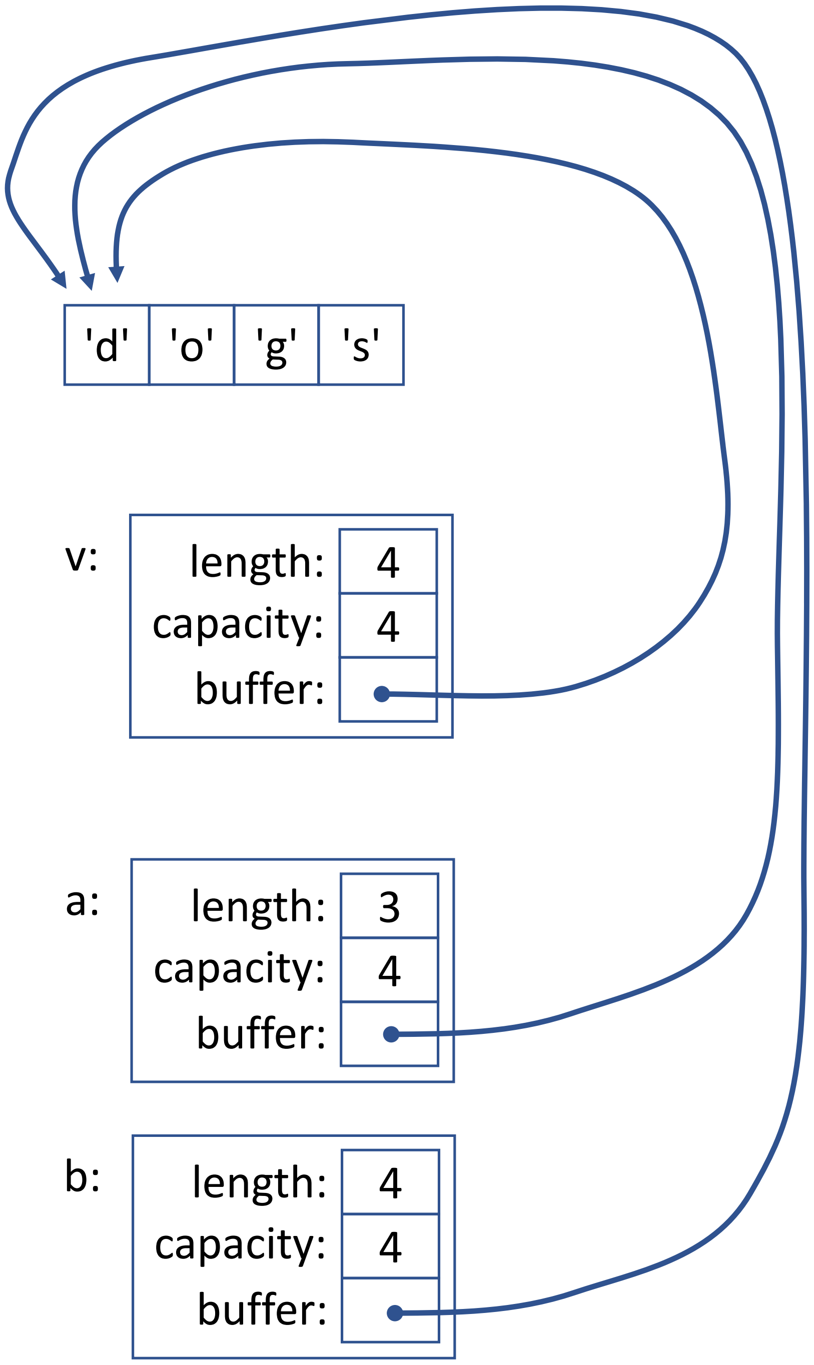
Copying objects

```
String::~String()
{
    delete []buffer;
}

String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```



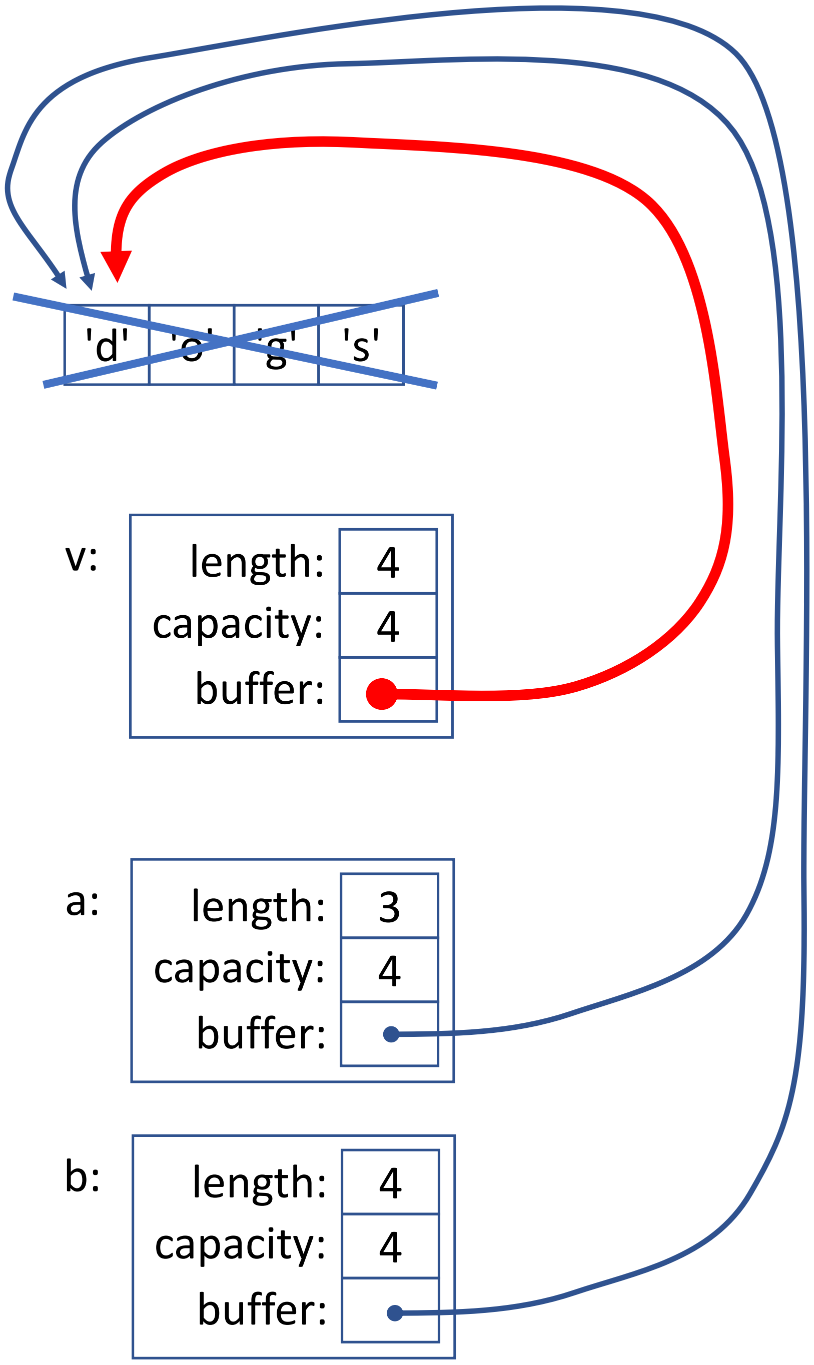
Copying objects

```
String::~String()
{
    delete []buffer;
}

String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```



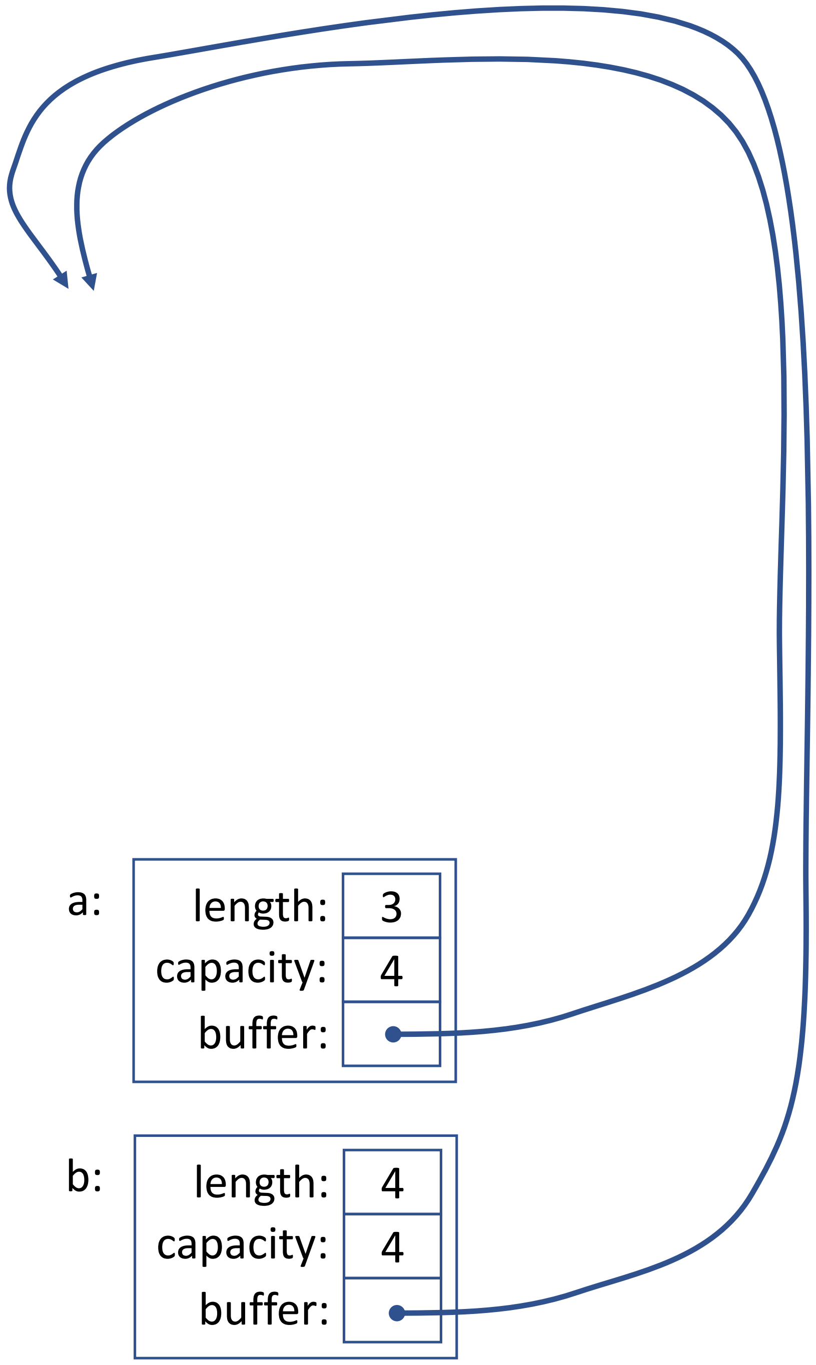
Copying objects

```
String::~String()
{
     delete []buffer;
}

String plural(String v)
{
     v.append('s');
     return v;
}

int main(int argc, char **argv)
{
     String a("dog");

     String b = plural( a );

     cout << b.at(0) << endl;
}
```



Copying objects

```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```



Fixing copies

- The problem is that copying violates assumptions
  - Each string should have a unique buffer
  - No two strings should point at the same buffer
- Fix 1 : pass the string around by pointer

Passing objects by pointer

```
String *plural(String *v)
{
    v->append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String *b = plural( &a );

    cout << b->at(0) << endl;
}
```

Passing objects by pointer

Does this mean the function will modify v?

```
String *plural(String *v);
```

Do we need to call delete on the return value?

Types should ideally send a clear message about how a function or method should be used

Fixing copies

- The problem is that copying violates assumptions
    - Each string should have a unique buffer
    - No two strings should point at the same buffer
- ~~Fix 1 : pass the string around by pointer~~
    - Error prone: *too easy to forget who owns what*
    - Clumsy : *doesn't really capture what we wanted*
- Fix 2 : add a ***copy constructor*** to the class

```
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
    String();

    String(const String &source)
    {
        length=source.length;
        capacity=source.length;
        buffer=new char[capacity];
        for(int i=0; i<length; i++){
            buffer[i] = source.buffer[i];
        }
    }

    ~String();
};
```

```
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
    String();

    String(const String &source)
    {
        length=source.length;
        capacity=source.capacity;
        buffer=new char[capacity];
        for(int i=0; i<length; i++){
            buffer[i] = source.buffer[i];
        }
    }

    ~String();
};
```

We have two new things here:

- `const` : a new keyword
- & : a new type modifier

```
class String
{
private:
    int length;
    int capacity;
    char *buffer;
public:
    String();

    String(const String &source)
    {
        length=source.length;
        capacity=source.capacity;
        buffer=new char[capacity];
        for(int i=0; i<length; i++){
            buffer[i] = source.buffer[i];
        }
    }

    ~String();
};
```

We have two new things here:

- `const` : a new keyword
- & : a new type modifier

For now:

source is a "read-only view" of an object

Read-only view of the String we want to copy

```
String::String(const String &source)
{
    // Copy the length and capacity verbatim
    length=source.length;
    capacity=source.capacity;

    // Create a new buffer just for us
    buffer=new char[capacity];

    // Copy the other string's data in
    for(int i=0; i<length; i++){
        buffer[i] = source.buffer[i];
    }
}
```

Read-only view of the String we want to copy

```
String::String(const String &source)
{
    // Copy the length and capacity verbatim
    length=source.length;
    capacity=source.capacity;

    // Create a new buffer just for us
    buffer=new char[capacity];

    // Copy the other string's data in
    for(int i=0; i<length; i++){
        buffer[i] = source.buffer[i];
    }
}
```

Read-only view of the String we want to copy

```
String::String(const String &source)
{
    // Copy the length and capacity verbatim
    length=source.length;
    capacity=source.capacity;

    // Create a new buffer just for us
    buffer=new char[capacity];

    // Copy the other string's data in
    for(int i=0; i<length; i++){
        buffer[i] = source.buffer[i];
    }
}
```

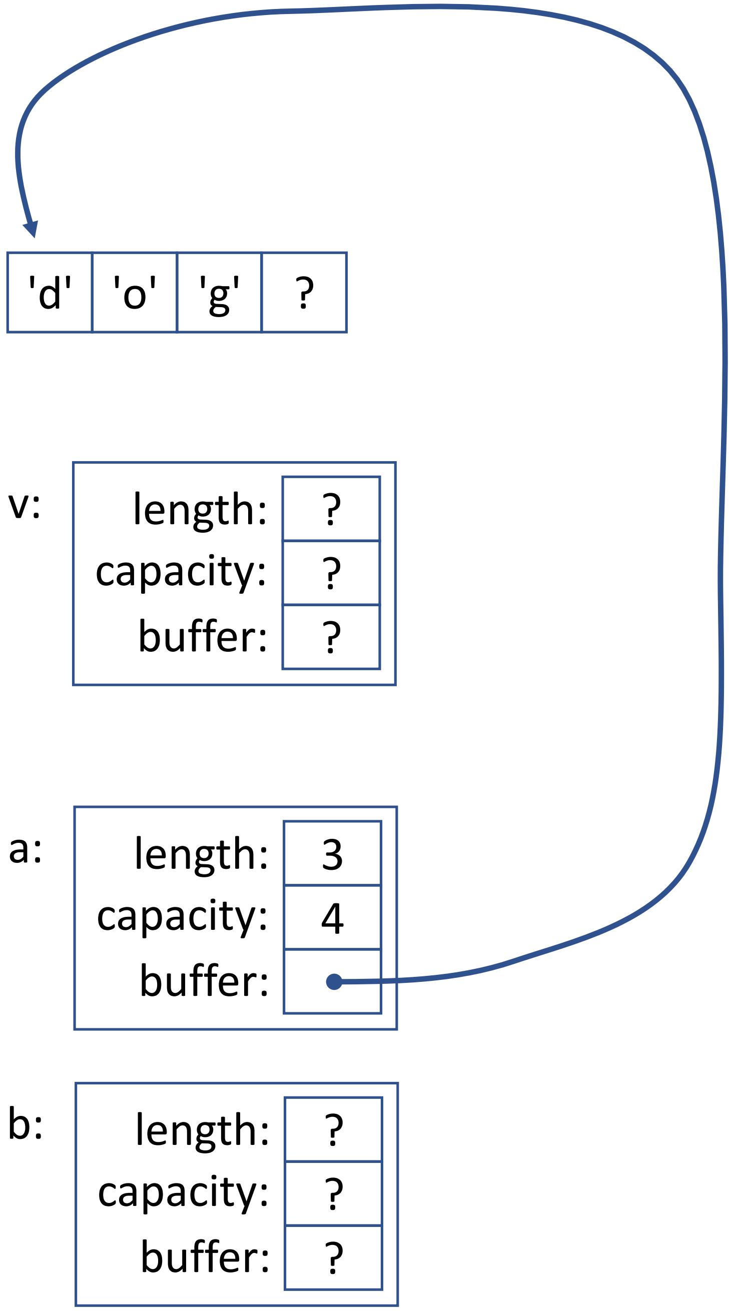
Read-only view of the String we want to copy

```
String::String(const String &source)
{
    // Copy the length and capacity verbatim
    length=source.length;
    capacity=source.capacity;

    // Create a new buffer just for us
    buffer=new char[capacity];

    // Copy the other string's data in
    for(int i=0; i<length; i++){
        buffer[i] = source.buffer[i];
    }
}
```

Newly constructed string has its own `buffer`, but it contains identical data to `source`

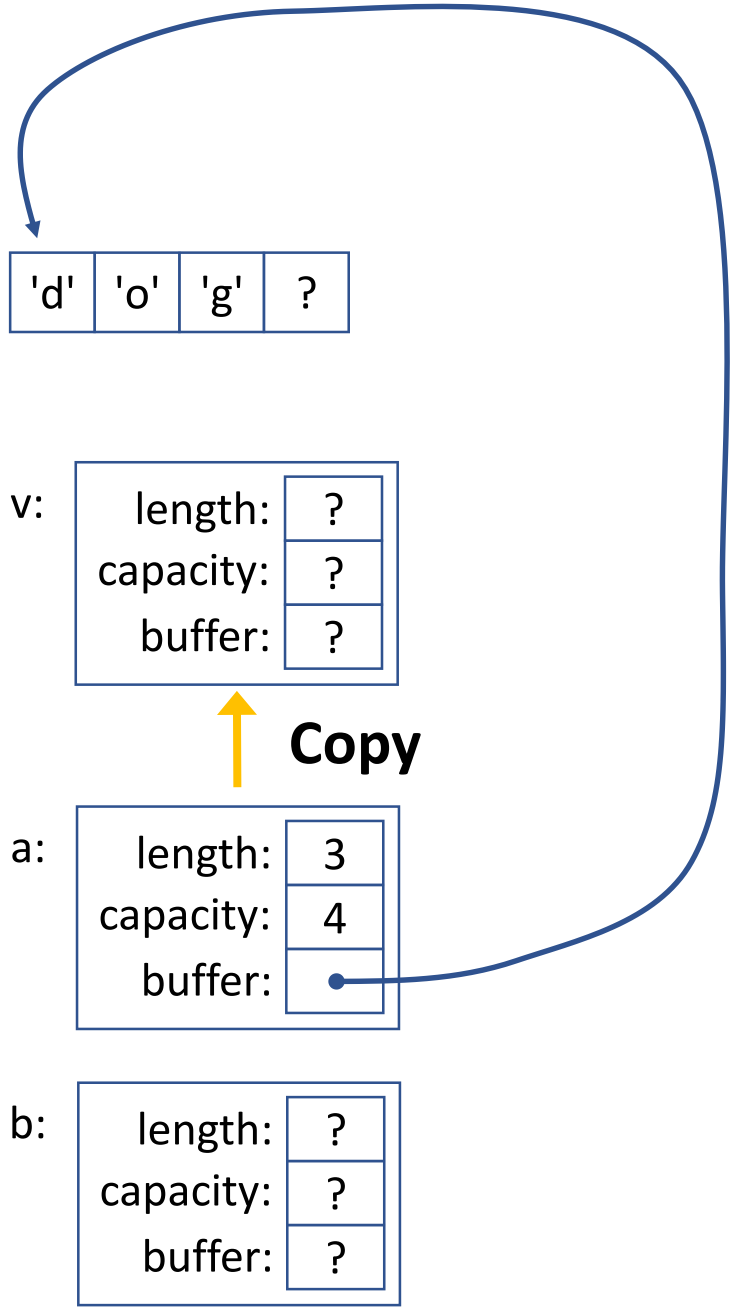
Copying objects

```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```



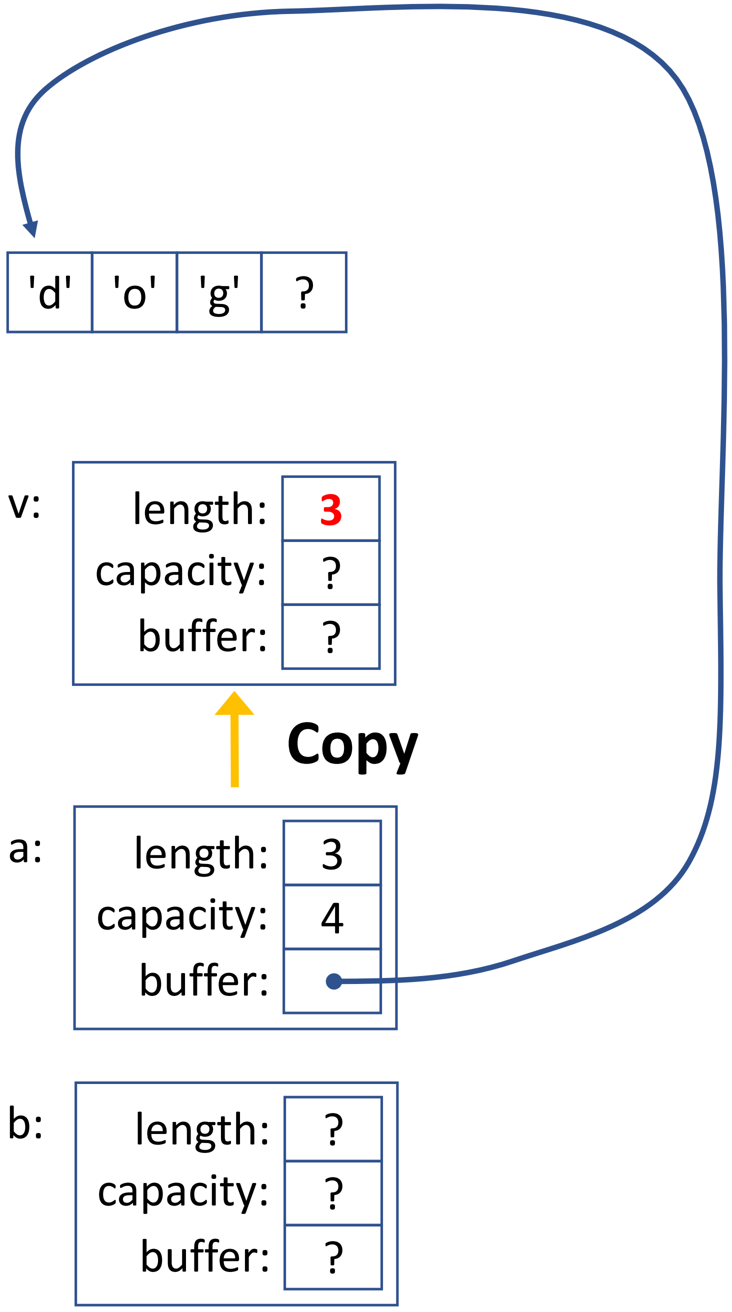
Copying objects

```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```



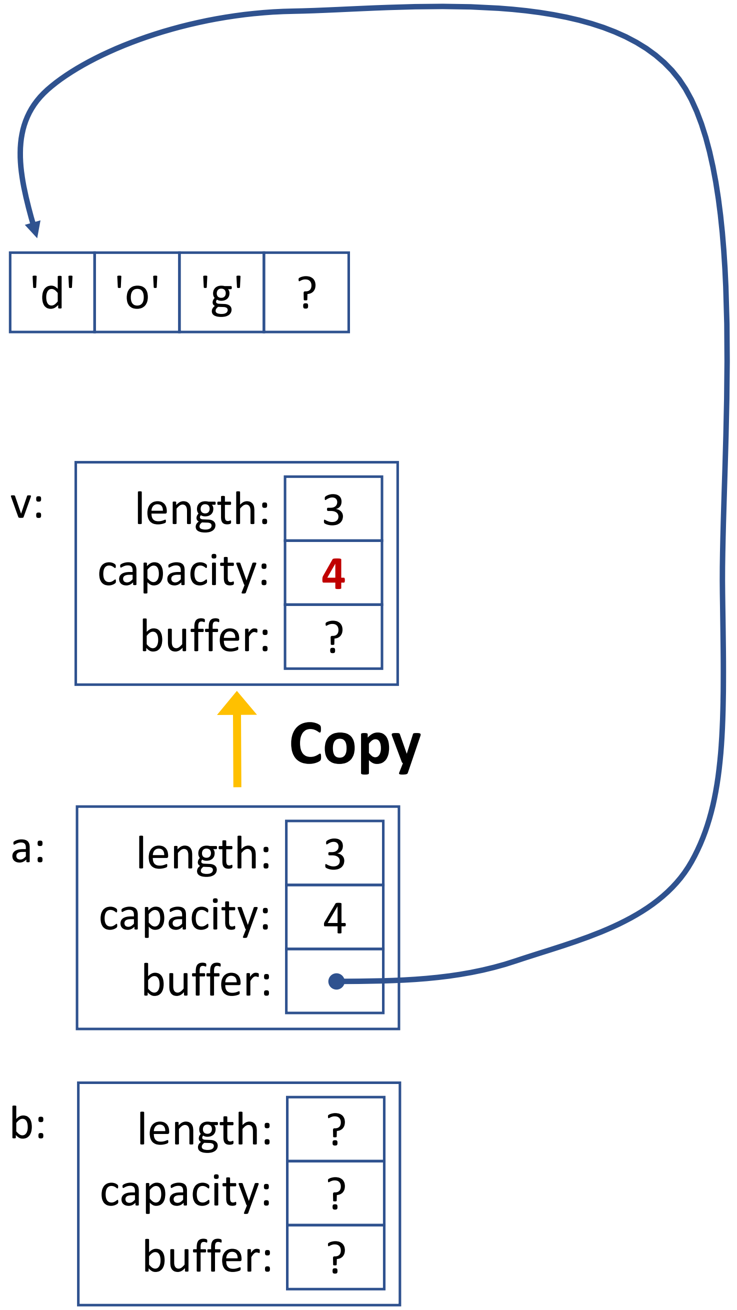
Copying objects

```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```



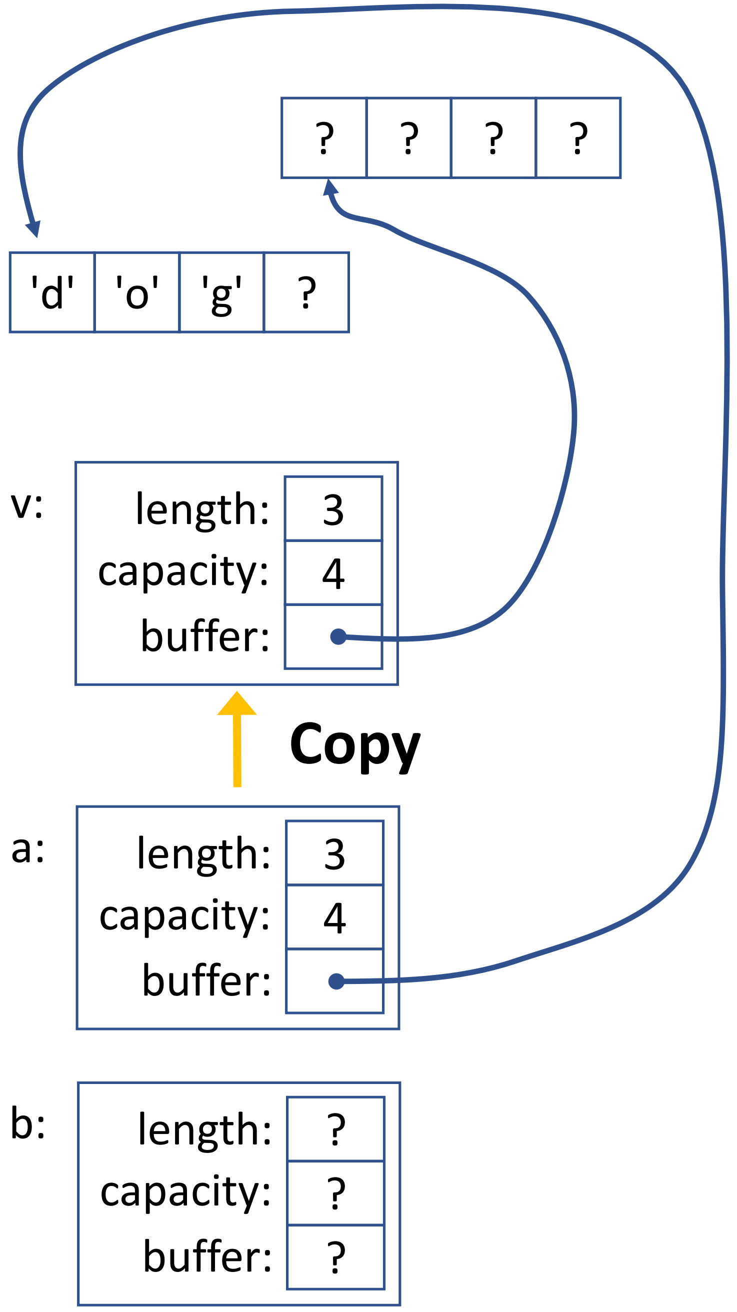
Copying objects

```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```



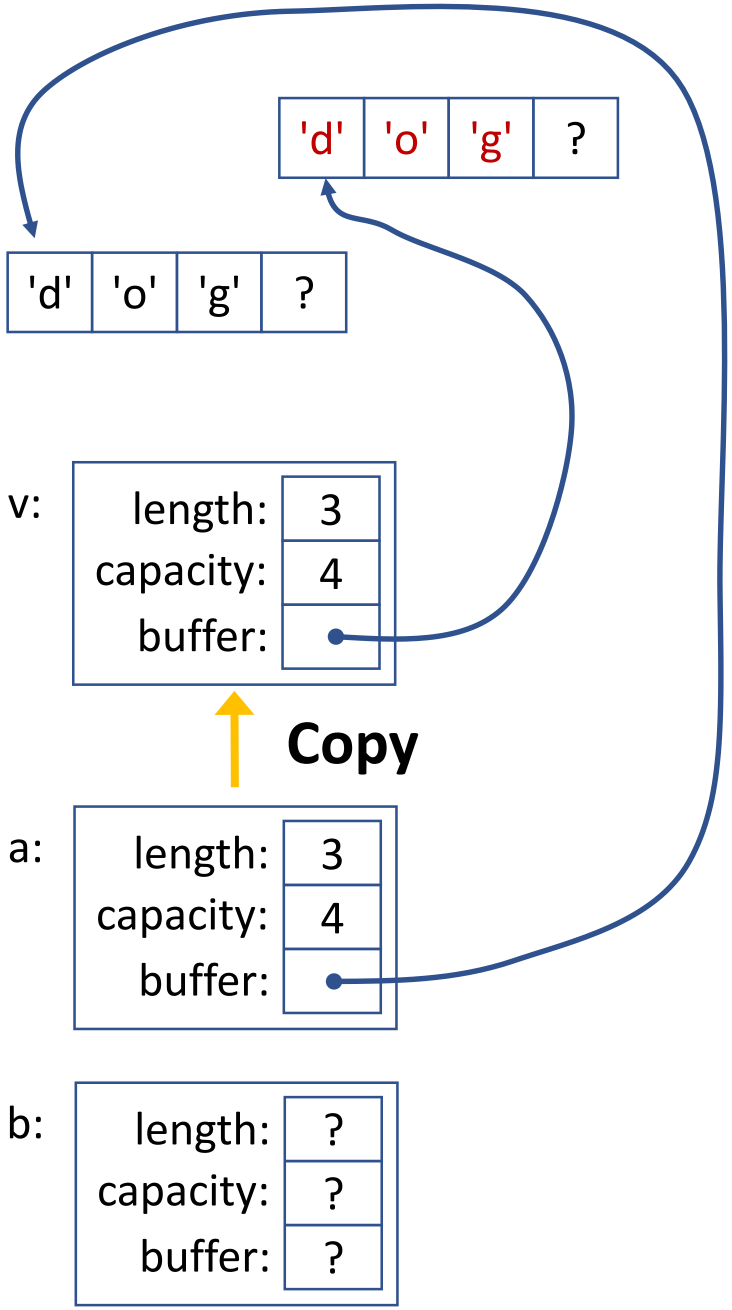
Copying objects

```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```



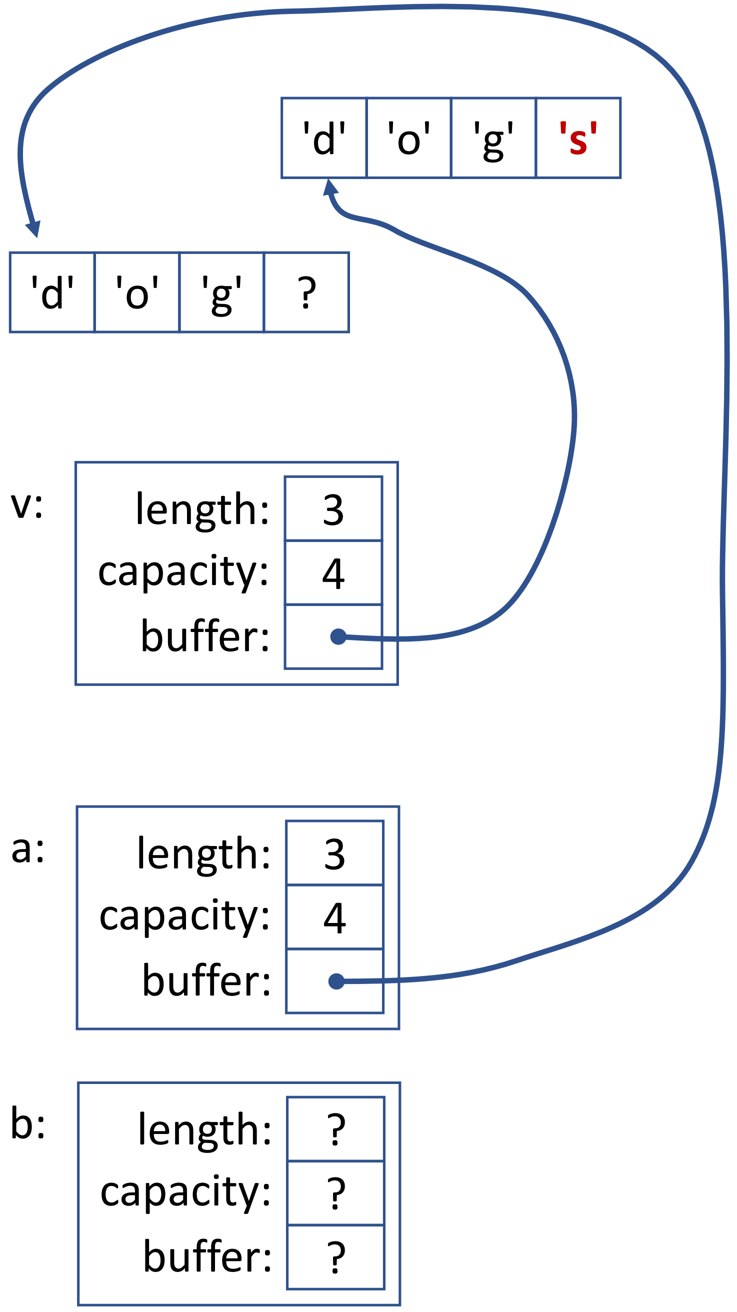
Copying objects

```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```



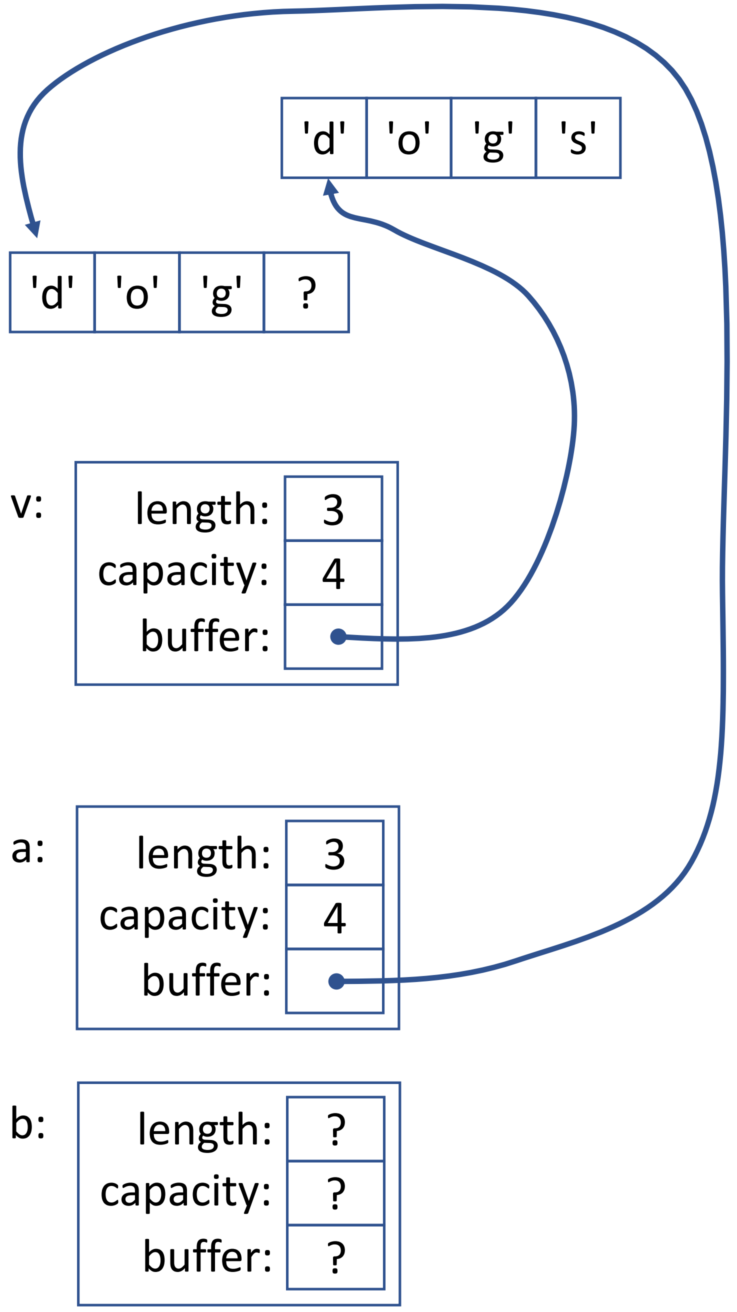
Copying objects

```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```



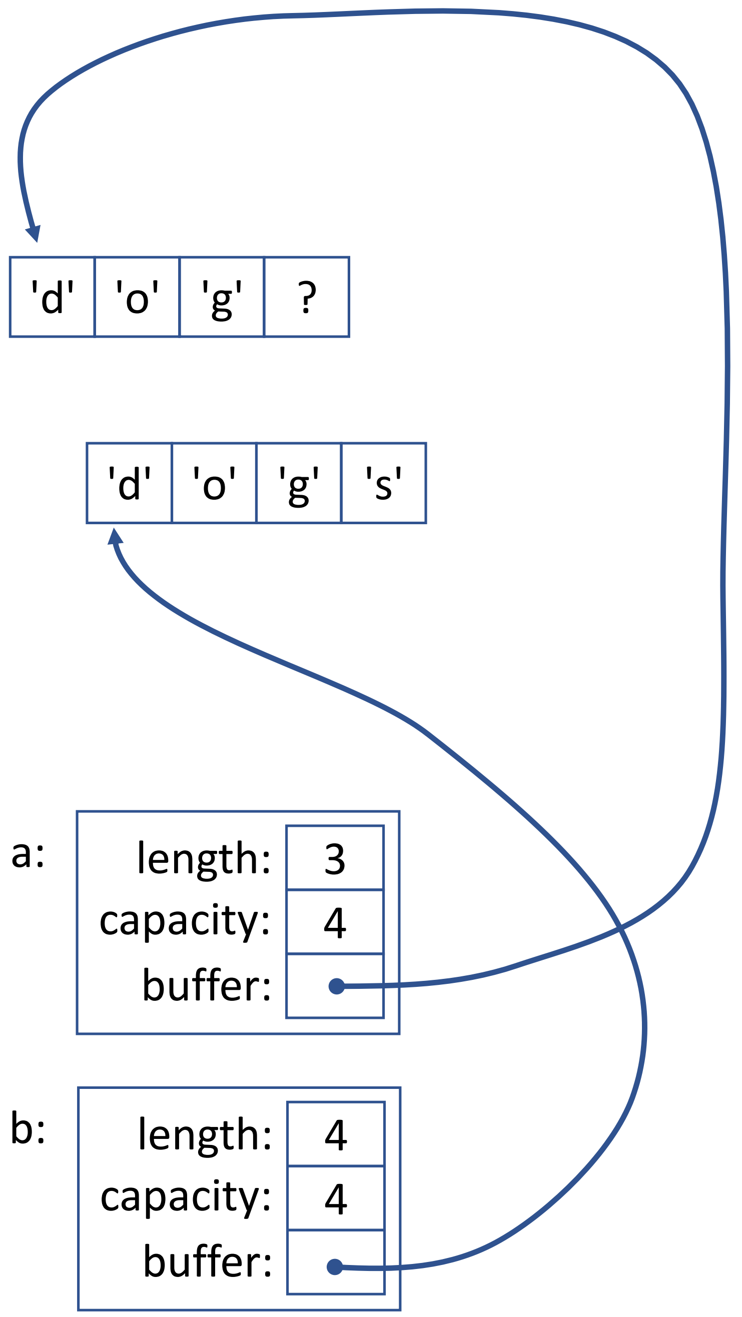
Copying objects

```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```



Copying objects

```
String plural(String v)
{
    v.append('s');
    return v;
}

int main(int argc, char **argv)
{
    String a("dog");

    String b = plural( a );

    cout << b.at(0) << endl;
}
```



Objects : copy constructor

- Objects need to protect their state
  - They particularly need to control ownership of memory
- The default copy constructor just copies all members
  - This works in *most* cases
- Copy constructor is needed when building types
  - e.g. if you are managing raw pointers

Next time : finishing plain objects

- Examining those lingering curiosities
  - What is `const`
  - What is "&" in a type?

    -> Pass by value versus pass by reference

- Overloading and operators

- Namespaces

- Typedefs