Good Programming Practices

Andrew Showers, Salles Viana
ALAC

```
/**
* Code Readability
*/
if (readable()) {
    be_happy();
} else {
    refactor();
}
```

Overview

- Code Refactoring
- Variable Naming and Access
- Tips for Readability
- Commenting and Documentation
- Black-Box Testing
- Const/Reference

Disclaimer

A style guide is about consistency and improving readability of code.

Know when to be inconsistent, guidelines will serve as a rule of thumb and not an absolute. Ask yourself if the rules being applied make the code more readable.

Code Refactoring

- Rewriting code for clarity, not bug fixing. Similar to writing paper drafts

- Rewrites may include:
  - Documentation / Comments
  - Change in Flow
  - Variable Naming Conventions
  - Creation of Functions / Classes
    - Simplification
    - Eliminating Duplicate Code
    - Re-usable

Variable Naming Conventions

- camelCase

  ballRadius

- Underscore

  between words -> ball_radius

  at start (private/protected) -> _speed

- Uppercase for constants

  GRAVITY

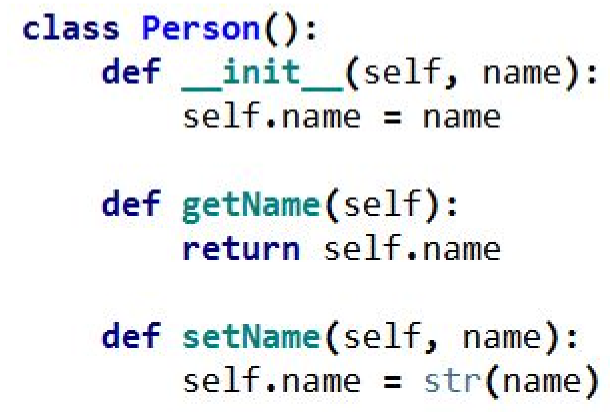
- Capitalize first word for classes

  Person()

Variable Access

- Avoid global variables unless it drastically simplifies code
  - Use your intuition. If the variable is used throughout the entire program, global is probably fine

- Avoid public variables for classes. Enforce the idea of encapsulation
  - Instead of public variables, create getters and setters



```python
class Person():
    def __init__(self, name):
        self.name = name

    def getName(self):
        return self.name

    def setName(self, name):
        self.name = str(name)
```

Avoid Redundant Labeling

Eliminate redundancy in order to create more readable code

```
import audio

core = audio.AudioCore()
controller = audio.AudioController()
```

VS

```
import audio

core = audio.Core()
controller = audio.Controller()
```

Avoid Deep Nesting

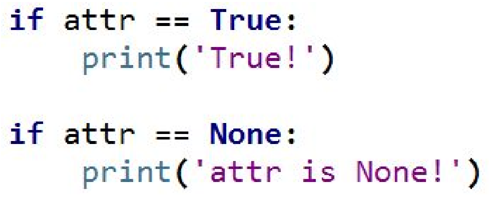
```python
# Function returns True if word has at least 5 letters,
# is an odd number of letters, and starts with the letter a
def word_check(word):
    if len(word) >= 5:
        if len(word) % 2 == 1:
            if word[0] == "a":
                return True

    return False
```

vs

```python
# Function returns True if word has at least 5 letters,
# is an odd number of letters, and starts with the letter a
def word_check(word):
    if len(word) < 5:
        return False

    if len(word) % 2 == 0:
        return False

    if word[0] != "a":
        return False

    return True
```

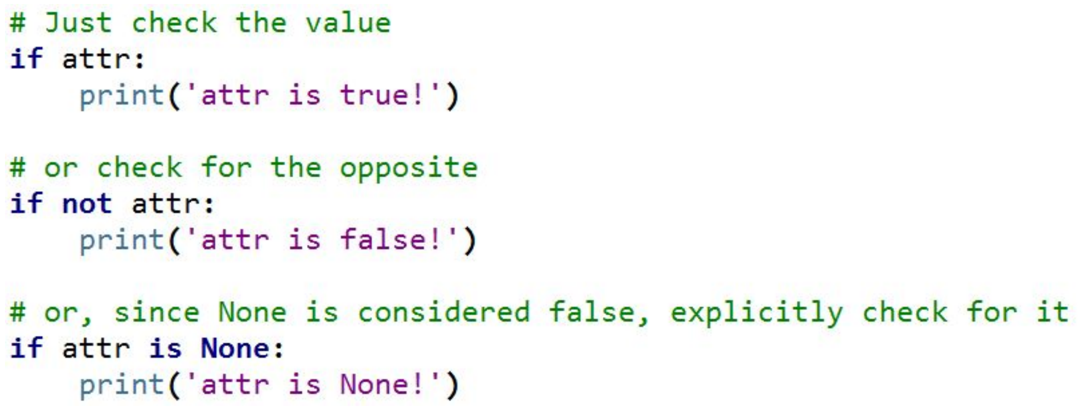
Avoid Explicit Comparisons (when possible)

https://docs.python.org/3/library/stdtypes.html#truth-value-testing



```python
if attr == True:
    print('True!')

if attr == None:
    print('attr is None!')
```

VS



```python
# Just check the value
if attr:
    print('attr is true!')

# or check for the opposite
if not attr:
    print('attr is false!')

# or, since None is considered false, explicitly check for it
if attr is None:
    print('attr is None!')
```

Avoid Long Lines

- Too many operations per line is confusing to read

```
points = np.asarray(open(sys.argv[1]).read().strip().replace("\n"," ").split(" ")).astype(np.float).reshape((-1, 2))
```

vs

```
# Read file and strip whitespace from beg and end
file = open(sys.argv[1])
file_contents = file.read().strip()

# Parse file contents into an array of floats
file_contents = file_contents.replace("\n"," ").split(" ")
points = np.asarray(file_contents).astype(np.float)
points = points.reshape((-1, 2))
```

One Statement per Line

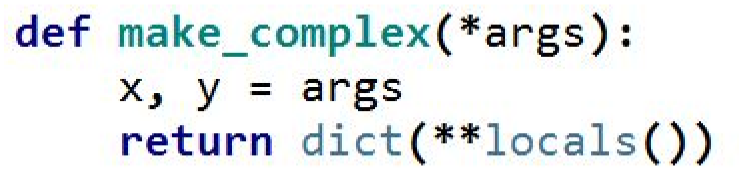
```python
print('one'); print('two')

if x == 1: print('one')

if <complex comparison> and <other complex comparison>:
    # do something
```

vs

```python
print('one')
print('two')

if x == 1:
    print('one')

cond1 = <complex comparison>
cond2 = <other complex comparison>
if cond1 and cond2:
    # do something
```

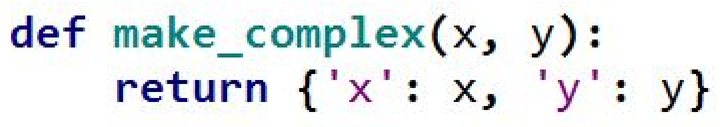
Strive for Simplicity

Code should be explicit and straightforward



```python
def make_complex(*args):
    x, y = args
    return dict(**locals())
```

vs



```python
def make_complex(x, y):
    return {'x': x, 'y': y}
```

Strive for Simplicity (cont.)

Use list comprehensions, filter(), and map() where applicable

```
# Filter elements greater than 4
a = [3, 4, 5]
b = []
for i in a:
    if i > 4:
        b.append(i)
```

VS

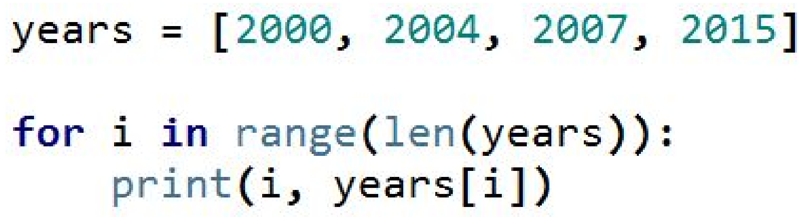
```
a = [3, 4, 5]
b = [i for i in a if i > 4]
```

VS

```
a = [3, 4, 5]
b = filter(lambda x: x > 4, a)
```

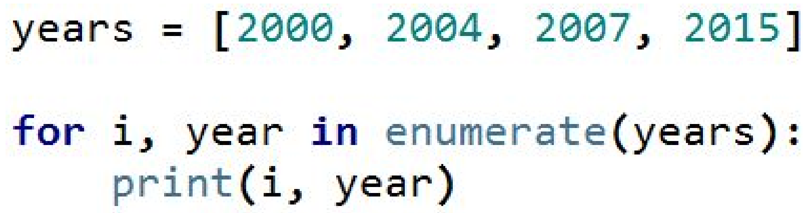
Strive for Simplicity (cont.)

Use enumerate to keep track of index and element values



```
years = [2000, 2004, 2007, 2015]

for i in range(len(years)):
    print(i, years[i])
```

VS



```
years = [2000, 2004, 2007, 2015]

for i, year in enumerate(years):
    print(i, year)
```

Commenting

- Explain logic in a clear and understandable manner
    - Avoid jargon when possible
    - Aim for explanation to be understood by non-programmers

- Spacing and logical grouping
    - Parsing data
    - Solving a particular subproblem
    - Displaying results

- Keep them up to date
    - Outdated comments lead to more confusion

Commenting (cont.)

- Avoiding obvious comments

```python
# print the age
print(age)
```

- When possible, rewrite the code so no comments are necessary

```python
# If the sign is a stop sign
if sign.color == 'red' and sign.sides == 8:
    stop()
```

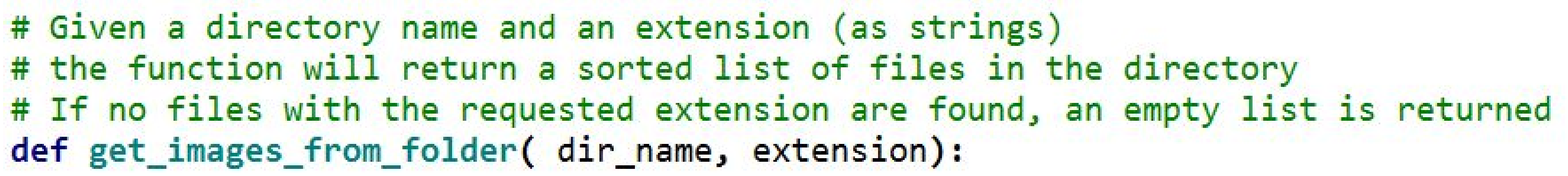
vs

```python
def is_stop_sign(sign):
    return sign.color == 'red' and sign.sides == 8

if is_stop_sign(sign):
    stop()
```

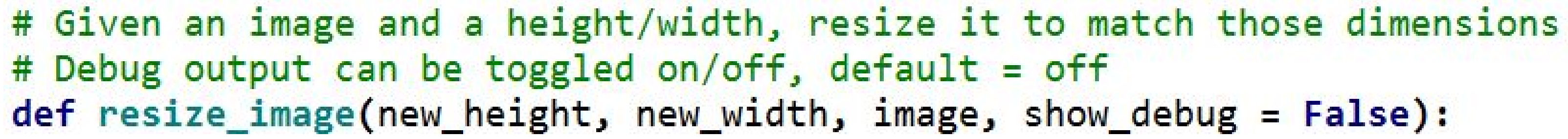
Commenting (cont.)

“At the beginning of every routine, it is helpful to provide standard, boilerplate comments, indicating the routines purpose, assumptions, and limitations. A boilerplate comment should be a brief introduction to understand why the routine exists and what it can do."
https://msdn.microsoft.com/en-us/library/aa260844(v=vs.60).aspx

Examples of function boilerplate:



```
# Given a directory name and an extension (as strings)
# the function will return a sorted list of files in the directory
# If no files with the requested extension are found, an empty list is returned
def get_images_from_folder( dir_name, extension):
```



```
# Given an image and a height/width, resize it to match those dimensions
# Debug output can be toggled on/off, default = off
def resize_image(new_height, new_width, image, show_debug = False):
```

Documentation

- Sphinx
  - http://www.sphinx-doc.org/en/stable/index.html
  - http://www.sphinx-doc.org/en/stable/ext/example_google.html

- EpyDoc
  - http://epydoc.sourceforge.net

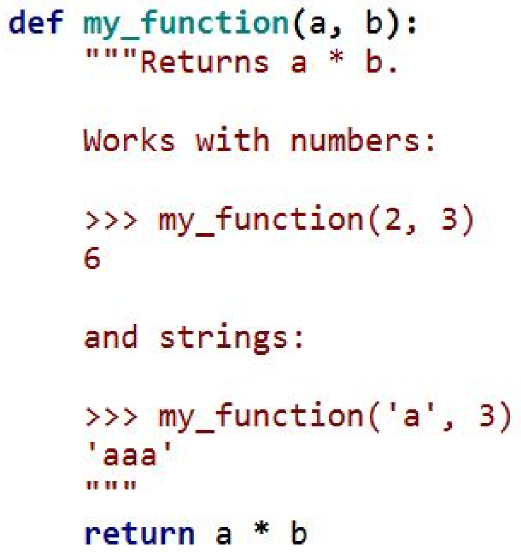
- PythonDoc
  - http://effbot.org/zone/pythondoc.htm#syntax

Black-Box Testing

- Given a function, you know what the output should be for given inputs
  - Select cases which cover all typically expected behavior
  - Software can verify function still works by running these tests

Input
Black Box
Output

- DocTest
  - https://en.wikipedia.org/wiki/Doctest



```
def my_function(a, b):
    """Returns a * b.

    Works with numbers:

    >>> my_function(2, 3)
    6

    and strings:

    >>> my_function('a', 3)
    'aaa'
    """
    return a * b
```

Avoid Convoluted Tricks

Just because you can doesn't mean you should

Examples:

- change how objects are created and instantiated
- change how the Python interpreter imports modules
- embedding C routines in Python

Exceptions exist, ask yourself if it is absolutely necessary (such as performance)

Some common mistakes seen in DS homeworks

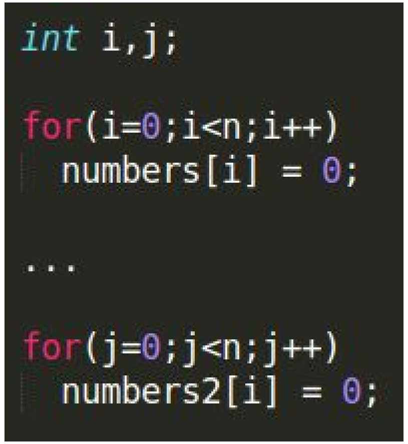
Some of these problems may also apply to CS-1

Not using local variables properly

- Even local variables may be “too global”
- Try to avoid declaring variables too soon

What is the problem with the following example?

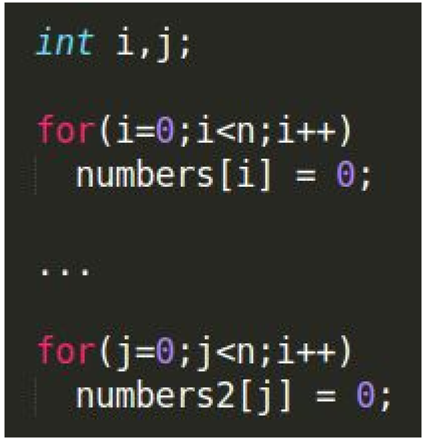
- Common cause: Ctrl-C + Ctrl-V



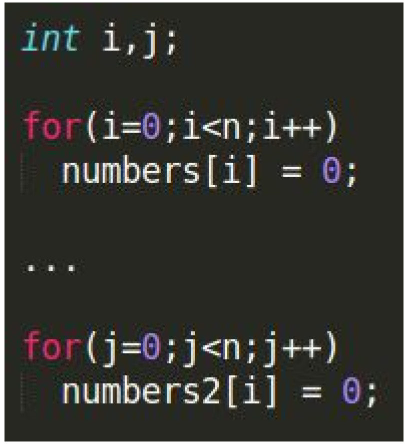
```
int i,j;

for(i=0;i<n;i++)
  numbers[i] = 0;

...

for(j=0;j<n;j++)
  numbers2[i] = 0;
```

Not using local variables properly

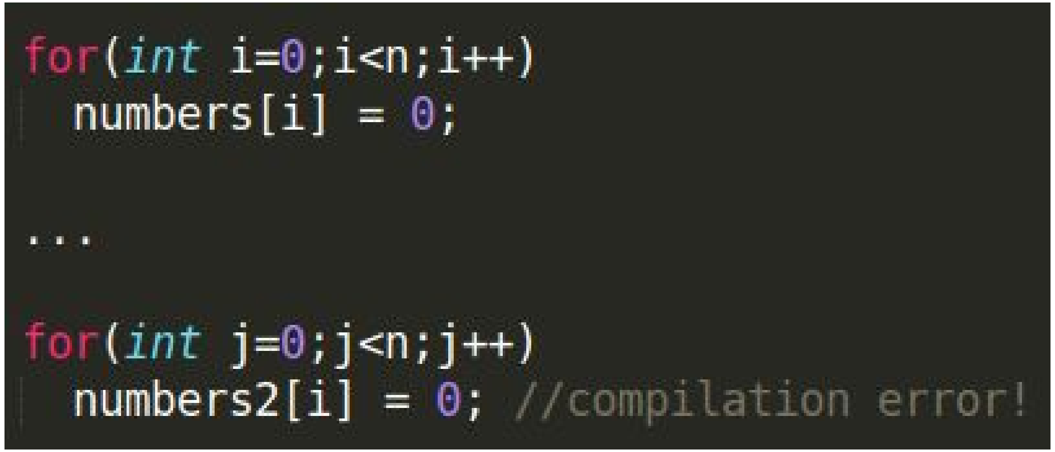
Ok now?



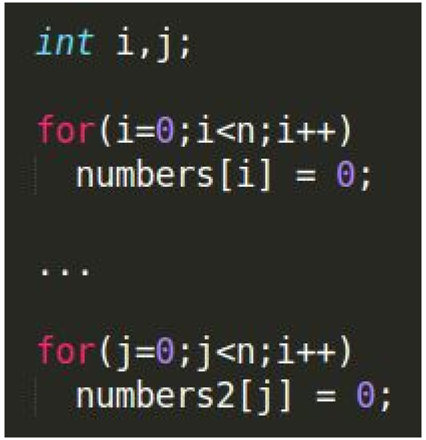
```
int i,j;

for(i=0;i<n;i++)
  numbers[i] = 0;

...

for(j=0;j<n;i++)
  numbers2[j] = 0;
```



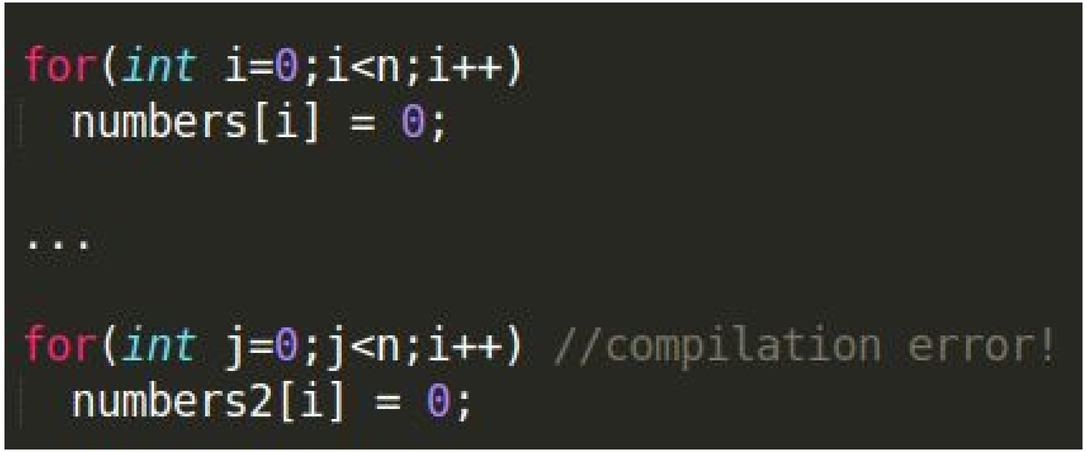
```
int i,j;

for(i=0;i<n;i++)
  numbers[i] = 0;


...


for(j=0;j<n;j++)
  numbers2[i] = 0;
```



```
for(int i=0;i<n;i++)
  numbers[i] = 0;


...


for(int j=0;j<n;j++)
  numbers2[i] = 0; //compilation error!
```

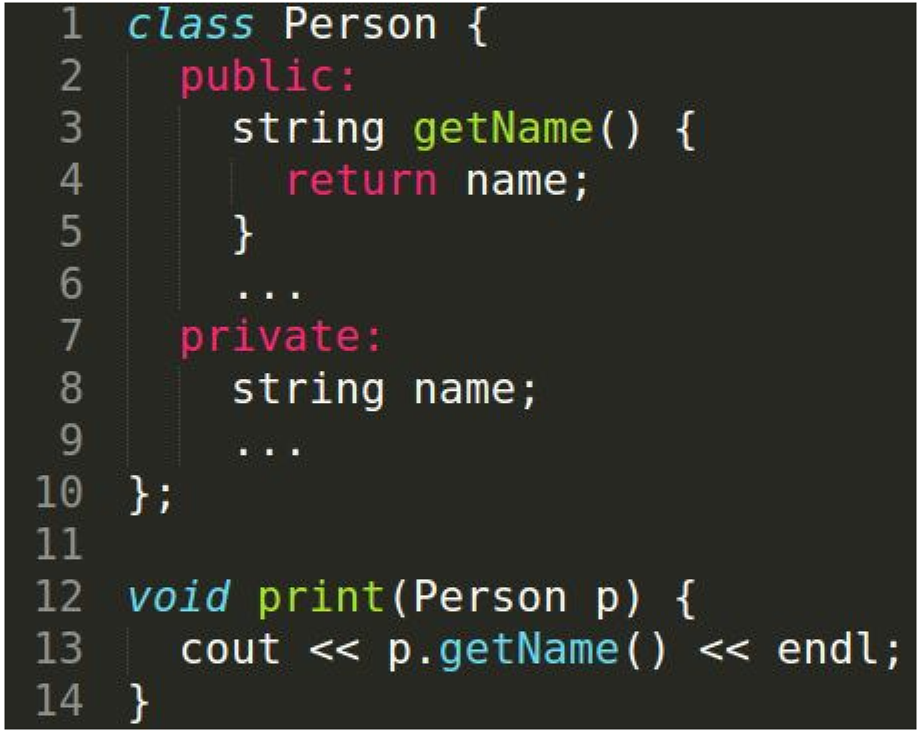


```
int i,j;

for(i=0;i<n;i++)
  numbers[i] = 0;


...


for(j=0;j<n;i++)
  numbers2[j] = 0;
```



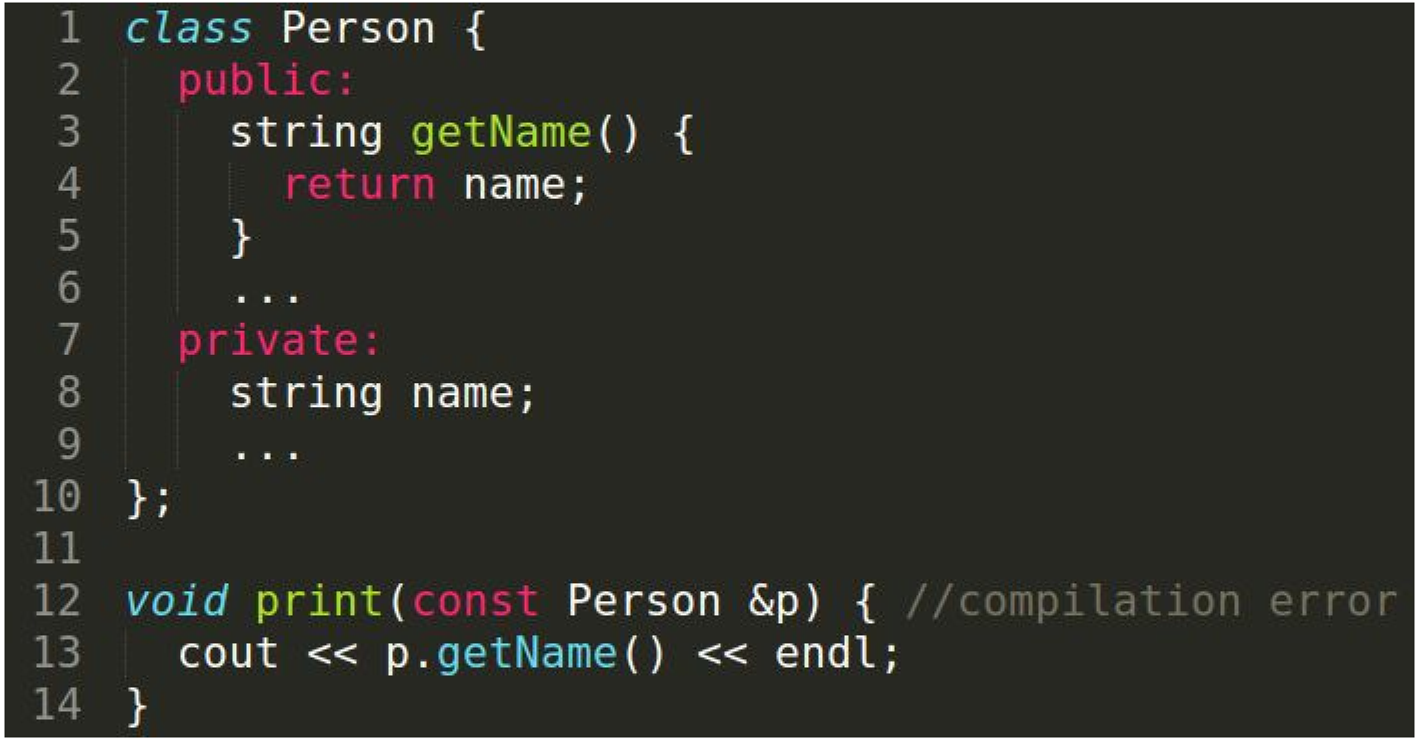
```
for(int i=0;i<n;i++)
  numbers[i] = 0;


...


for(int j=0;j<n;i++) //compilation error!
  numbers2[i] = 0;
```

Another common mistake...



```cpp
class Person {
  public:
    string getName() {
      return name;
    }
    ...
  private:
    string name;
    ...
};

void print(Person p) {
  cout << p.getName() << endl;
}
```

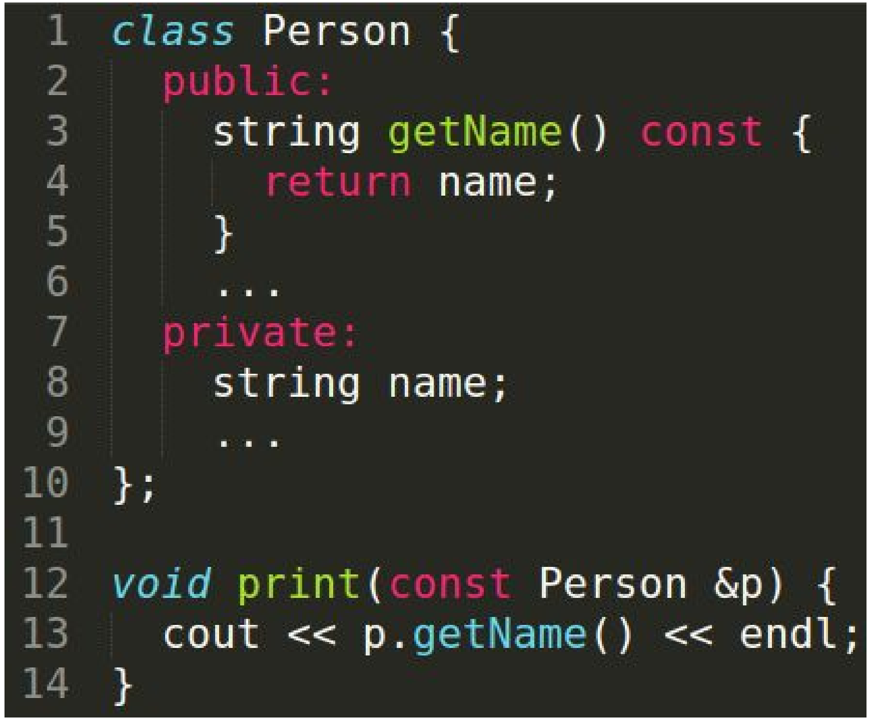
Another common mistake...



```cpp
class Person {
  public:
    string getName() {
      return name;
    }
    ...
  private:
    string name;
    ...
};

void print(const Person &p) { //compilation error
  cout << p.getName() << endl;
}
```

Const/reference

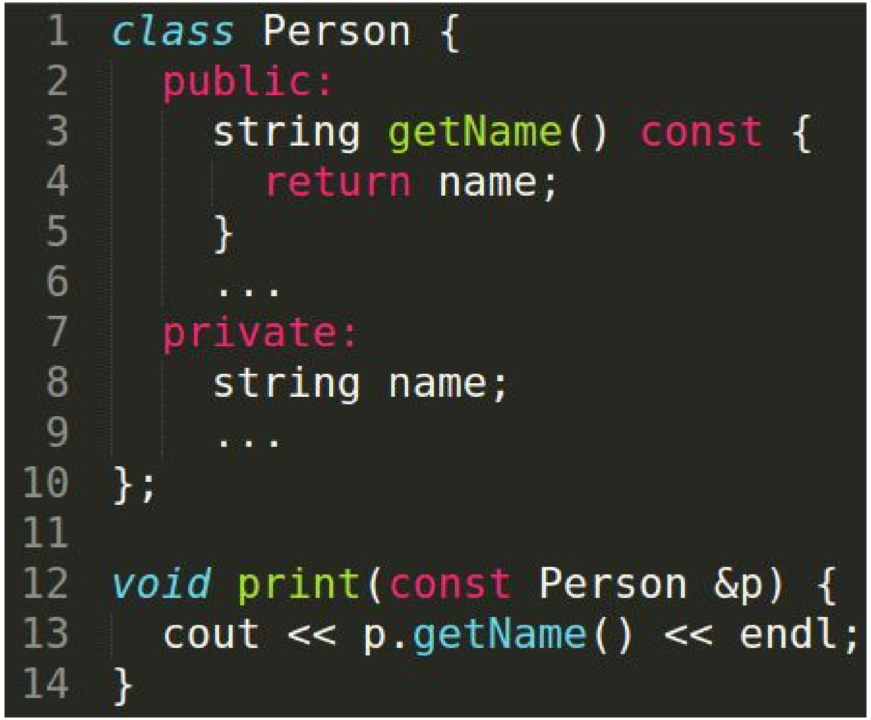
Ok now?



```cpp
class Person {
  public:
    string getName() const {
      return name;
    }
    ...
  private:
    string name;
    ...
};

void print(const Person &p) {
  cout << p.getName() << endl;
}
```

Const/reference

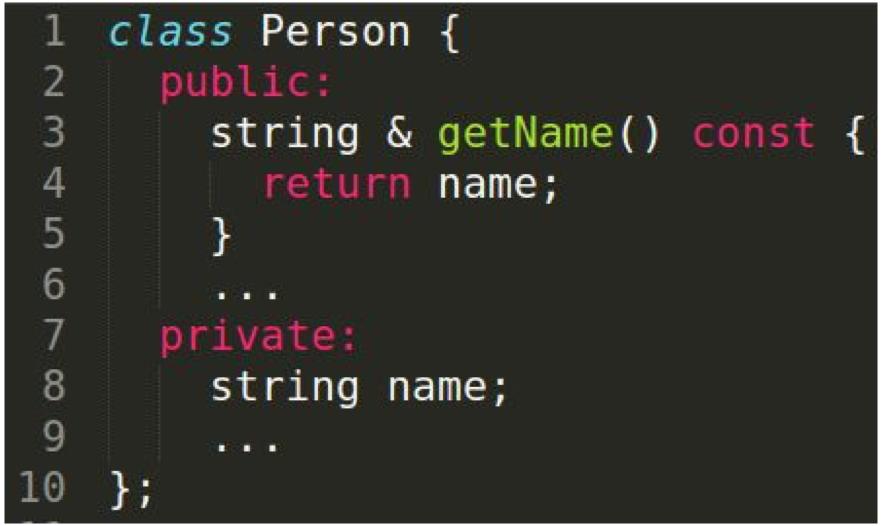
getName() → always returns a COPY of name…



```
class Person {
  public:
    string getName() const {
      return name;
    }
    ...
  private:
    string name;
    ...
};

void print(const Person &p) {
  cout << p.getName() << endl;
}
```

Const/reference

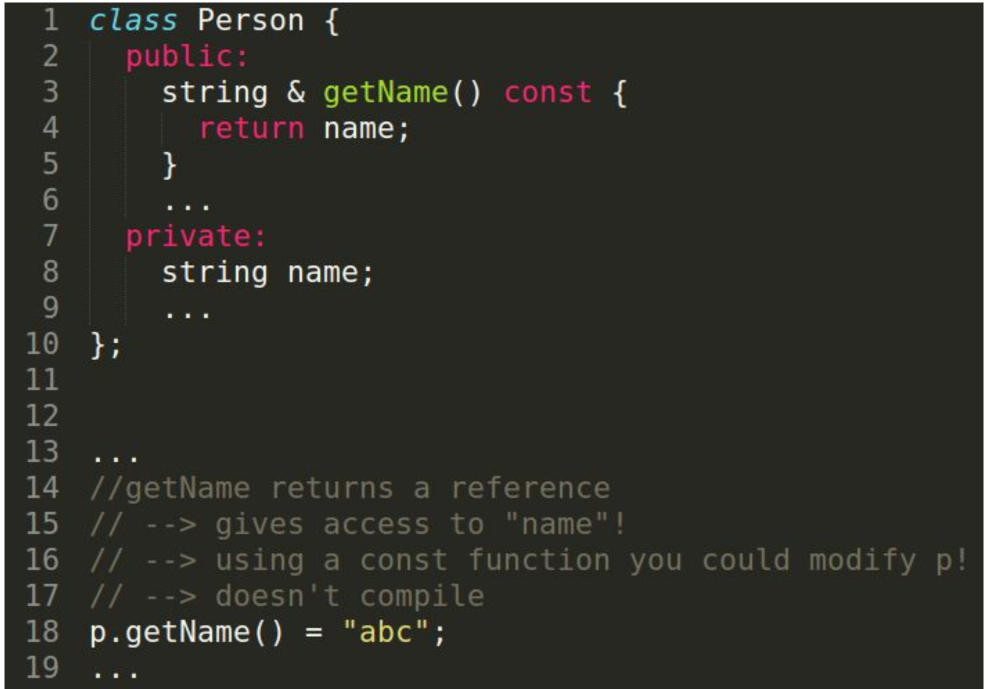
Ok now?



```
class Person {
  public:
    string & getName() const {
      return name;
    }
    ...
  private:
    string name;
    ...
};
```

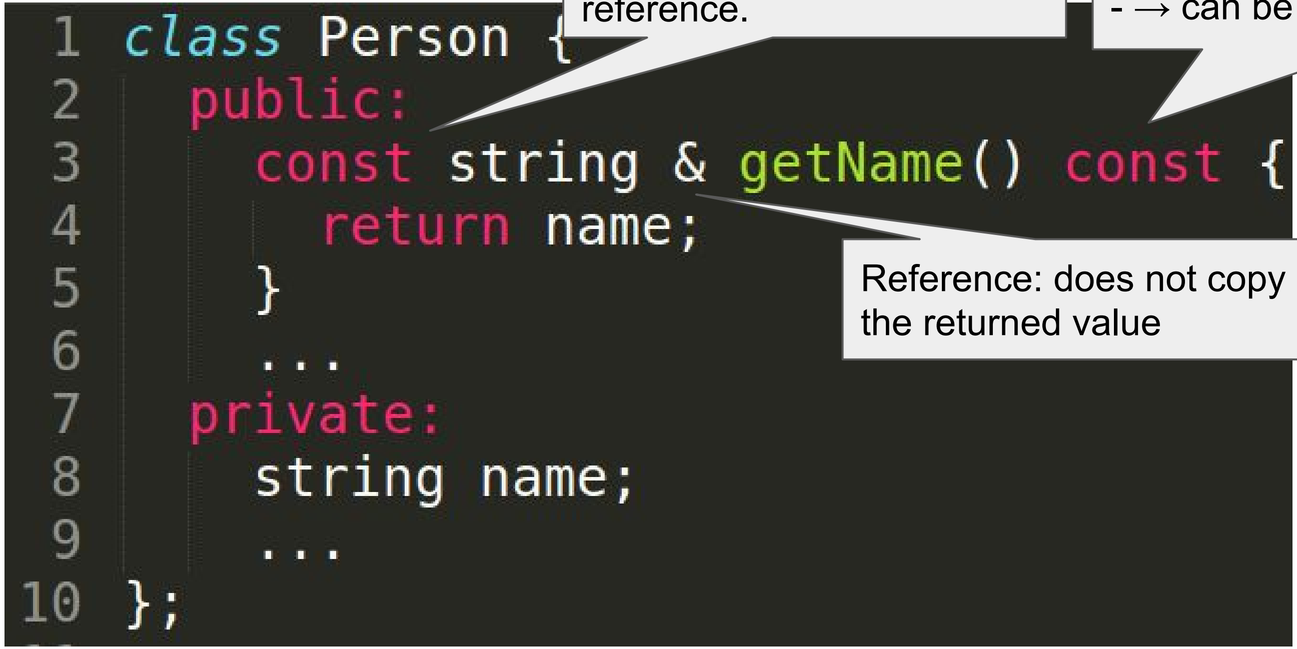
Const/reference

Where does the compilation error happen? What if getName() was not const?



```cpp
class Person {
  public:
    string & getName() const {
      return name;
    }
    ...
  private:
    string name;
    ...
};


...
//getName returns a reference
// --> gives access to "name"!
// --> using a const function you could modify p!
// --> doesn't compile
p.getName() = "abc";
...
```

Const/reference



```
class Person {
  public:
    const string & getName() const {
      return name;
    }
    ...
  private:
    string name;
    ...
};
```

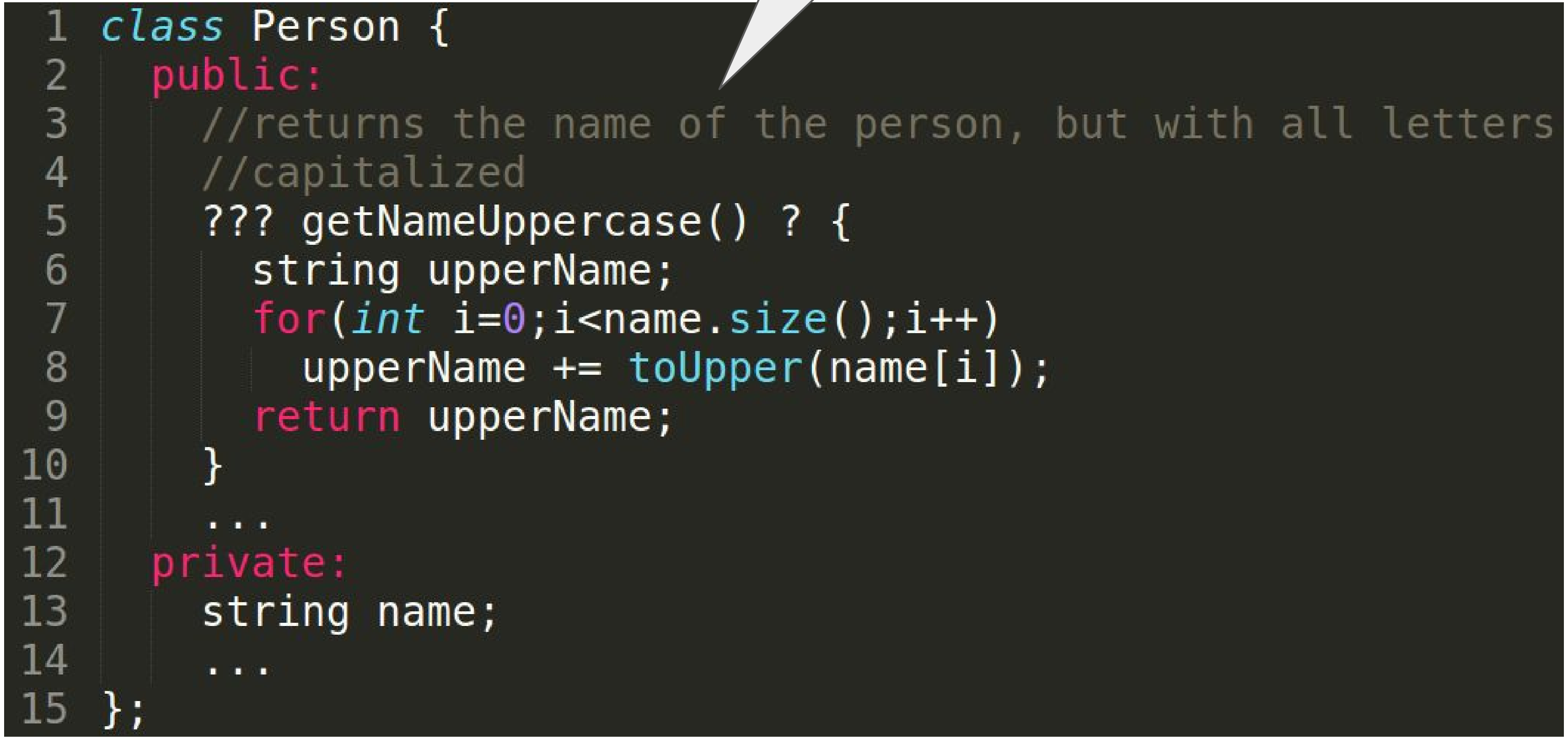
Const returned value: you can't modify the returned reference.

Const function:
- Can't modify object.
- → can be applied to a const object

Reference: does not copy the returned value

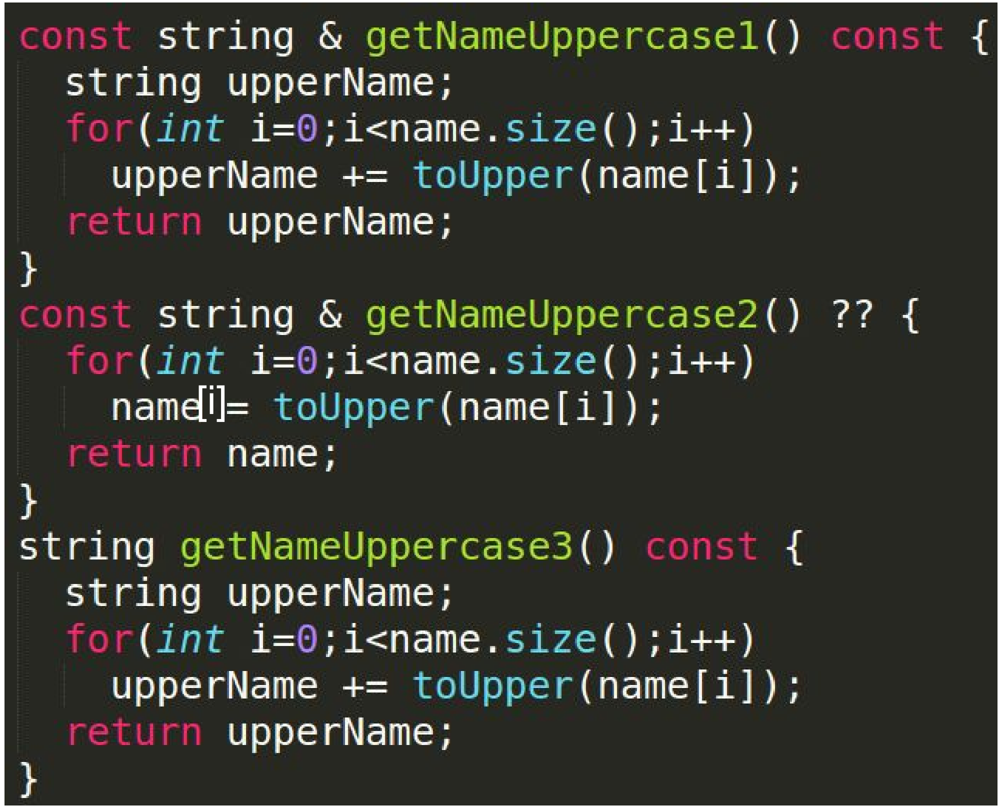
Const/reference

This function should be in the .cpp file !



```
class Person {
  public:
    //returns the name of the person, but with all letters
    //capitalized
    ??? getNameUppercase() ? {
      string upperName;
      for(int i=0;i<name.size();i++)
        upperName += toUpper(name[i]);
      return upperName;
    }
    ...
  private:
    string name;
    ...
};
```

Const/reference



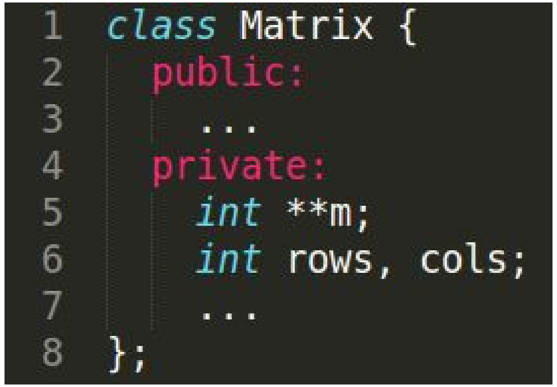
```
const string & getNameUppercase1() const {
  string upperName;
  for(int i=0;i<name.size();i++)
    upperName += toUpper(name[i]);
  return upperName;
}
const string & getNameUppercase2() ?? {
  for(int i=0;i<name.size();i++)
    name[i]= toUpper(name[i]);
  return name;
}
string getNameUppercase3() const {
  string upperName;
  for(int i=0;i<name.size();i++)
    upperName += toUpper(name[i]);
  return upperName;
}
```

Code reuse

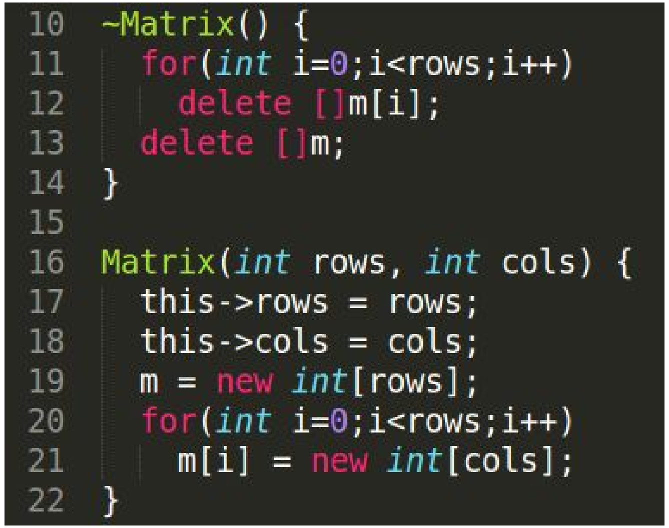
Avoid having duplicate code

- More code → more bugs (usually)
- More code → more things to fix
- More code → more time to implement
- More code → lower grade!

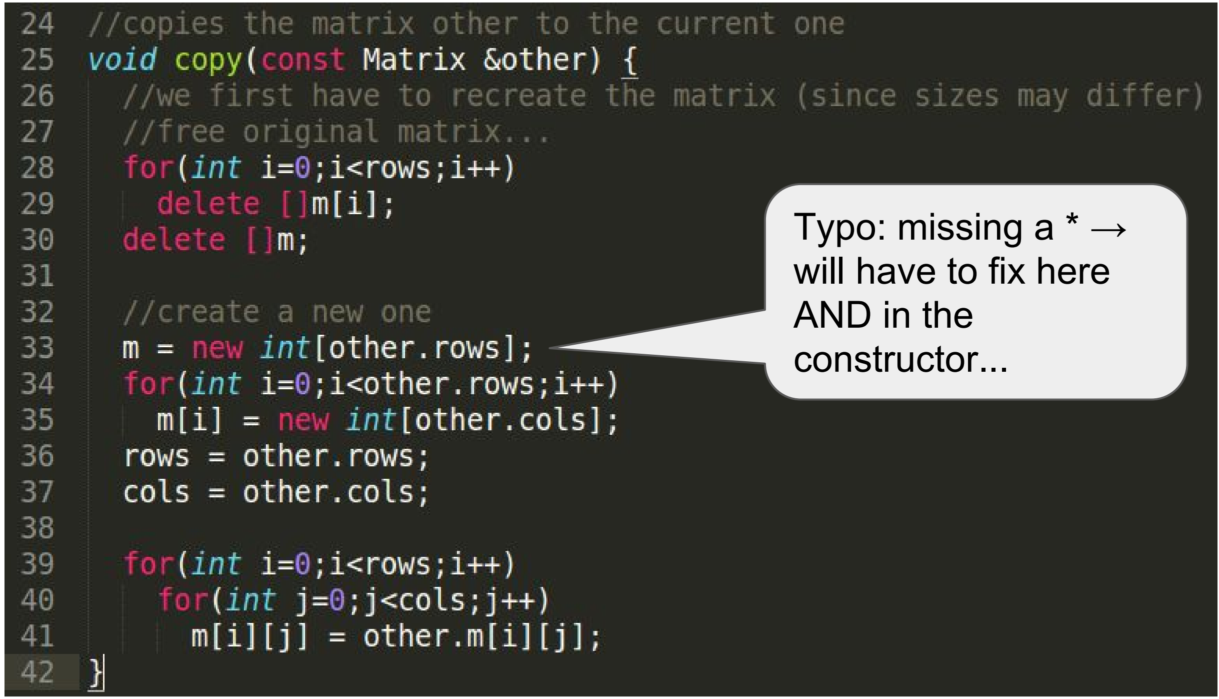
What is the problem here?



```cpp
class Matrix {
  public:
    ...
  private:
    int **m;
    int rows, cols;
    ...
};
```



```cpp
~Matrix() {
  for(int i=0;i<rows;i++)
    delete []m[i];
  delete []m;
}

Matrix(int rows, int cols) {
  this->rows = rows;
  this->cols = cols;
  m = new int[rows];
  for(int i=0;i<rows;i++)
    m[i] = new int[cols];
}
```

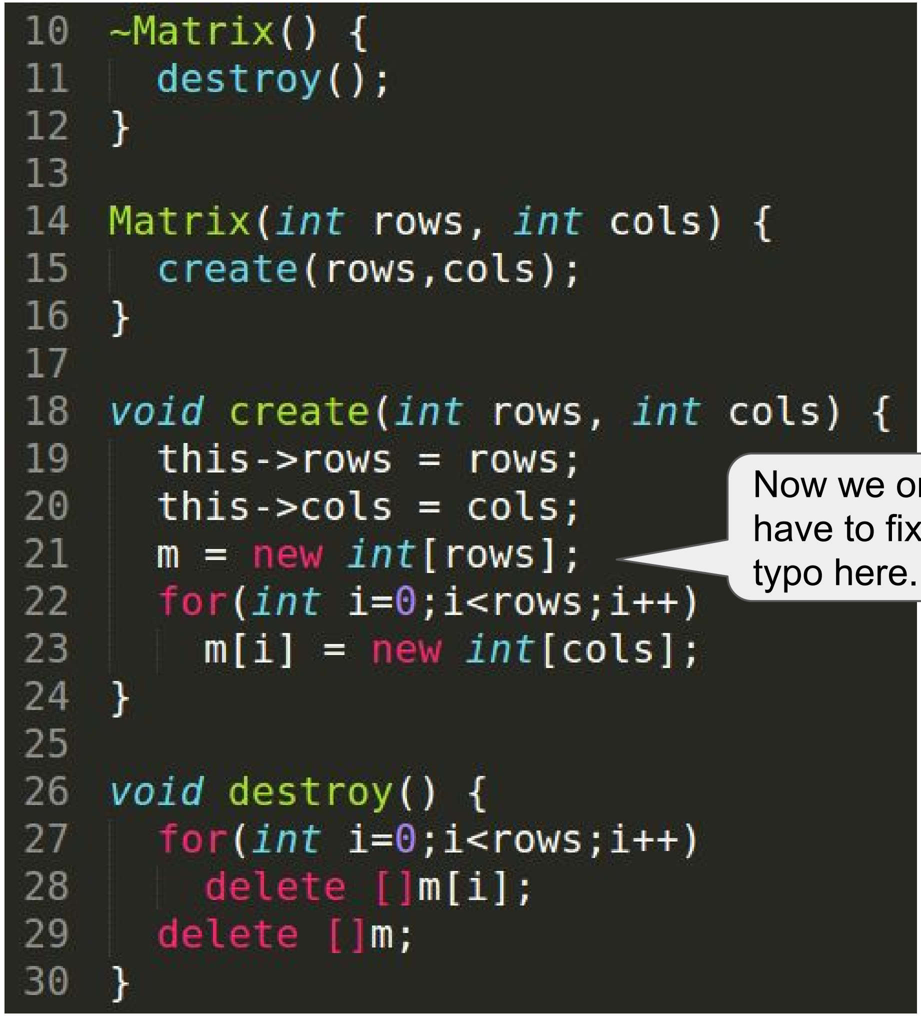


```cpp
//copies the matrix other to the current one
void copy(const Matrix &other) {
  //we first have to recreate the matrix (since sizes may differ)
  //free original matrix...
  for(int i=0;i<rows;i++)
    delete []m[i];
  delete []m;

  //create a new one
  m = new int[other.rows];
  for(int i=0;i<other.rows;i++)
    m[i] = new int[other.cols];
  rows = other.rows;
  cols = other.cols;

  for(int i=0;i<rows;i++)
    for(int j=0;j<cols;j++)
      m[i][j] = other.m[i][j];
}
```

Typo: missing a * → will have to fix here AND in the constructor...

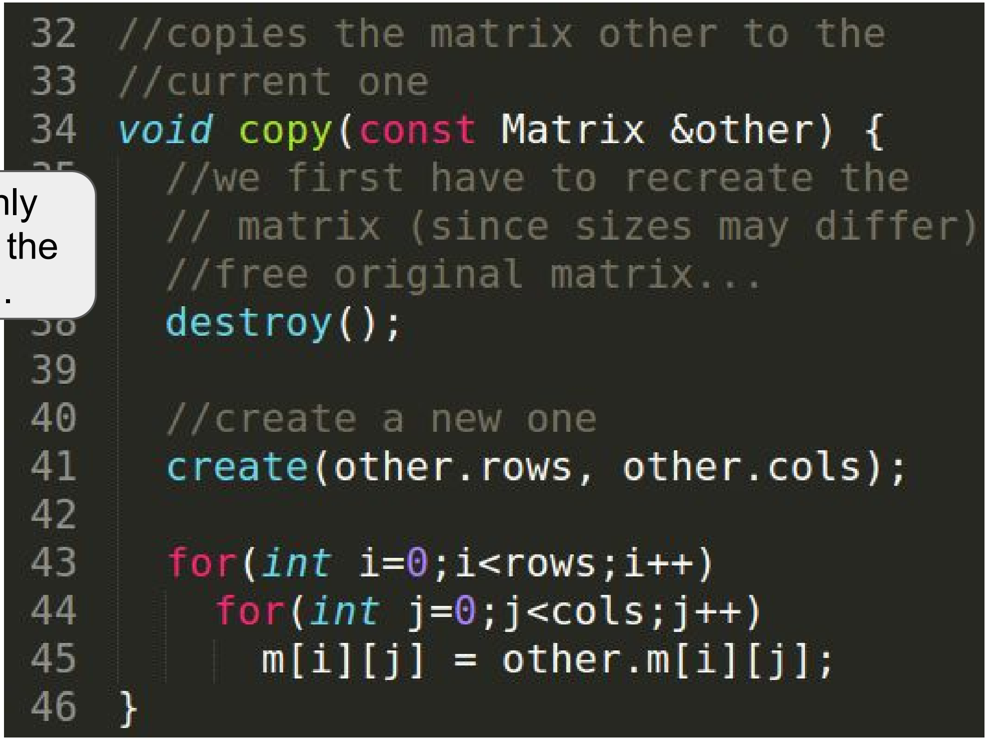
```
Matrix a(5,2);
Matrix b(3,3);

a = b; //the operator will call a.copy(b)
```

- Smaller code
- Easier to read



```
~Matrix() {
  destroy();
}

Matrix(int rows, int cols) {
  create(rows,cols);
}

void create(int rows, int cols) {
  this->rows = rows;
  this->cols = cols;
  m = new int[rows];
  for(int i=0;i<rows;i++)
    m[i] = new int[cols];
}

void destroy() {
  for(int i=0;i<rows;i++)
    delete []m[i];
  delete []m;
}
```

Now we only have to fix the typo here...



```
//copies the matrix other to the
//current one
void copy(const Matrix &other) {
  //we first have to recreate the
  // matrix (since sizes may differ)
  //free original matrix...
  destroy();

  //create a new one
  create(other.rows, other.cols);

  for(int i=0;i<rows;i++)
    for(int j=0;j<cols;j++)
      m[i][j] = other.m[i][j];
}
```

References

PEP 8 -- Style Guide for Python Code:
https://www.python.org/dev/peps/pep-0008/

The Best of the Best Practices (BOBP) Guide for Python:
https://gist.github.com/sloria/7001839

Coding Techniques and Programming Practices:
https://msdn.microsoft.com/en-us/library/aa260844(v=vs.60).aspx

Code Style http://docs.python-guide.org/en/latest/writing/style/