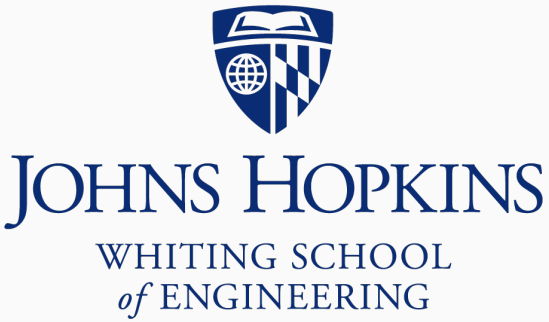
Classes & object-oriented programming

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Classes & object-oriented programming

We saw `structs`, which bundle variables that describe different aspects of the same thing:

```
struct Rectangle {
    double width;
    double height;
};
```

We might additionally define functions that work with `Rectangles`:

C++ classes & object-oriented programming

```cpp
#include <iostream>
using std::cout;  using std::endl;

struct Rectangle {
    double width;
    double height;
};

double area(Rectangle r) { return r.width * r.height; }

void print_rectangle(Rectangle r) {
    cout << "width=" << r.width
         << ", height=" << r.height
         << ", area=" << area(r) << endl;
}

int main() {
    Rectangle r = {30.0, 40.0};
    print_rectangle(r);
    return 0;
}
```

C++ classes & object-oriented programming

```
$ g++ -c rectangle1.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o rectangle1 rectangle1.o
$ ./rectangle1
width=30, height=40, area=1200
```

C++ classes & object-oriented programming

As "object-oriented" programmers, we prefer to have related functionality (`print_rectangle`, `area`) be *part of the object*

We couldn't do this in C, but C++ allows us to define `classes`...

C++ class

```cpp
#include <iostream>
using std::cout;  using std::endl;

class Rectangle {
public:
    void print() const {
        cout << "width=" << width
             << ", height=" << height
             << ", area=" << area() << endl;
    }
private:
    double width;
    double height;
    double area() const { return width*height; }
};

int main() {
    Rectangle r = {30.0, 40.0};
    r.print();
    return 0;
}
```

C++ class

```
#include <iostream>
using std::cout;  using std::endl;

class Rectangle {
public:
    void print() const { // print is a "member function"
        cout << "width=" << width
             << ", height=" << height
             << ", area=" << area() << endl;
    }
    void set(double w, double h) { width = w; height = h; }
private:
    double width;  // width is a field; every Rectangle has one
    double height; // height is another field
    double area() const { return width*height; } // another "member function"
}; // need a semicolon here, like with structs

int main() {
    Rectangle r;
    r.set(30.0, 40.0); // call r's "set" member function
    r.print();         // call r's "print" member function
    return 0;
}
```

C++: Object oriented programming

```
$ g++ -c rect_class.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o rect_class rect_class.o
$ ./rect_class
width=30, height=40, area=1200
```

C++ classes

C++ `classes` have *fields* (like `struct`) and *member functions* (unlike `struct`)

Member functions can access/modify fields, and can call other member functions

In previous example:

- `width` and `height` are fields
- `print`, `area` and `set` are member functions
  - `area` uses `width` and `height` in its calculation
  - `set` modifies `width` and `height`
  - `print` uses `width` & `height` and calls `area()`

C++ classes

In Java:

- What we call "member functions" Java calls "methods"
- What we call "fields" Java sometimes calls "instance variables"

We use the generic term "members" to refer to both fields and member functions

C++ classes

Like in Java, fields and member functions can be `public` or `private`

- (or `protected`, discussed later)

Labels `public:` or `private:` divide the class definition into sections according to whether members are `public` or `private`

- E.g. all members declared after the `public:` label are public until the end of the class or until `private:`

Members are `private` by default

C++ classes

Functions can be declared *and defined* inside `class { ... };`

- We only do this if it's *very short*
- Otherwise, we put a *prototype* in the class definition and we define the member function in a `.cpp` file

For example, this might appear in `grade_list.h`:

```
class GradeList {
    ...
    void add(double grade)
    {
        // definition inside class
        grades.push_back(grade);
    }
};
```

C++ classes

Or (more often) we put this in `grade_list.h`:

```
class GradeList {
    ...
    void add(double grade);
    ...
};
```

...and this in `grade_list.cpp`:

```
#include "grade_list.h"

// definition outside class
void GradeList::add(double grade) {
    grades.push_back(grade);
}
```

C++ classes

Note that when defining a member function, you must prefix the function name with the class name followed by `::`

```
void GradeList::add(double grade) {
    grades.push_back(grade);
}
```

C++ classes

A private member can be accessed from other member functions in the class, but *not* by the user

```
class GradeList {
public:
    ...
    void add(double grade) {
        grades.push_back(grade); // OK
    }
    ...
private:
    std::vector<double> grades;
};
```

C++ classes

```
class GradeList {
    ...
private:
    std::vector<double> grades;
};

int main() {
    GradeList gl;
    cout << gl.grades.size() << endl; // not OK!
    return 0;
}
```

C++ classes

public fields and member function can be accessed freely

```
class GradeList {
public:
    void add(double grade)
    {
        grades.push_back(grade);
    }
    void add7() { add(7.0); } // OK!
    ...
};

int main() {
    GradeList gl;
    gl.add(45.0); // also OK!
    return 0;
}
```

C++ classes

You might want to initialize a field when it's declared:

```
class GradeList {
    ...

    bool is_sorted = false;
};
```

This is common in Java but was not allowed in C++ before C++11

- Called a *default member initializer*
- We will avoid it, preferring to initialize using constructors & initializer lists, discussed later