601.220 Intermediate Programming

C++ classes

C++: non-object-oriented programming

We already saw `struct`s, which bring together several variables that collectively describe one ”thing”:

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

We might additionally define some functions that do things with rectangles, like print them or calculate their area. For example:

C++: non-object-oriented programming

```cpp
// class1.cpp

#include <iostream>

struct rectangle {
    double width;
    double height;
};

void print_rectangle(struct rectangle r) {
    std::cout << "width=" << r.width << ", height=" << r.height << std::endl;
}

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

int main() {
    rectangle r = {30.0, 40.0};
    print_rectangle(r);
    std::cout << "area=" << area(r) << std::endl;
    return 0;
}
```

```
$ g++ -o class1 class1.cpp -std=c++11 -pedantic -Wall -Wextra
```

```
$ ./class1
```

```
width=30, height=40
area=1200
```

C++: object-oriented programming

As good Java or Python programmers, though, we prefer to have the related functionality (`print_rectangle`, `area`) be **part of the object** (the `struct` in this case)

No simple way to do this in C. But in C++...

C++: object-oriented programming

```cpp
// class2.cpp
#include <iostream>

class Rectangle {
public:
    double width;
    double height;

    void print() const {
        std::cout << "width=" << width << ", height=" << height << std::endl;
    }

    double area() const {
        return width * height;
    }
};

int main() {
    Rectangle r = {30.0, 40.0};
    r.print();
    std::cout << "area=" << r.area() << std::endl;
    return 0;
}
```

```
$ g++ -o class2 class2.cpp -std=c++11 -pedantic -Wall -Wextra
```

```
$ ./class2
```

```
width=30, height=40
area=1200
```

C++: Object-oriented programming

- A class definition is like a **blueprint defining a type**
- Objects of that type are created from that **blueprint**
- Once we define a class, we have one blueprint from which we can create 0 or more objects
- Each of the objects is an **instance of the class and has its own copies of all instance variables**

C++: Object-oriented programming

```
void print() const {
  ...
}
```

- The use of `const` as modifier in method header indicates that the function will not modify any member fields
- The `rectangle` object on which we call `print` will not be modified by the call

C++ classes

Basic principles for writing C++ classes

- Class definition goes in a *.h* file
- Functions can be declared **and defined** inside `class{...};`
- Only define member function inside the class definition if it's **very** short
  - this is called "in-lining" the function definition
- Otherwise, put a prototype in the class definition and define the member function in a *.cpp* file
  - you'll need to qualify the function with the class scope such as `Classname::function(){}` in the *.cpp* file

C++ classes

```
// rectangle.h
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle {
    ...
    double area() const {
        // short definition inside class (in-lining)
        return width * height;
    }
    ...
};
#endif // RECTANGLE_H
```

C++ classes

```
// rectangle.h
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle {
    ...
        double area() const;
    ...
};
#endif // RECTANGLE_H

// rectangle.cpp
#include "rectangle.h"

double Rectangle::area() const {
    // define outside class
    return width * height;
}
```

C++ classes

- Fields and member functions can be `public` or `private`
    - (or `protected`, discussed later)
- We use `public:` and `private:` to divide class definition into sections according to whether members are `public` or `private`
- Everything is `private` by default
- A `public` field or member function can be accessed freely by any code with access to the class definition (code that includes the *.h* file)
- A `private` field or member function can be accessed from other member functions in the class, but **not** by a user of the class

C++ classes

```cpp
class Rectangle {
public:
    ...
    double area() const {
        // definition inside class
        return width * height;  // OK
    }
    ...
private:
    double width, height;
};
```

C++ classes

```
class Rectangle {
    ...
private:
    double width, height;
};

int main() {
    Rectangle r;
    std::cout << r.width << std::endl; // not OK!
    return 0;
}
```

C++ classes

Do **not** try to initialize class fields immediately when they are declared

```
class Rectangle {
    ...
    double width = 10;  // NO!
    ...
};
```

This kind of initialization is only allowed for static fields

C++ classes

```
// rectangle.h
#ifndef RECTANGLE_H
#define RECTANGLE_H
class Rectangle {
public:
    double area() const {
        return width * height;
    }

private:
    double width, height;
};
#endif // RECTANGLE_H
```