Classes

Review and Additional Features

Terminology

A class:

- is simply a user-defined data type—one that can be used just like int or string
- defines a template for a given concept (a car, a robot, a player), specifying both the properties and behaviors it should have

An object (or instance):

- is a variable with a class as its data type
- has its own copies of the properties declared in its class
- has all the behaviors provided by its class

Terminology

class Human

- describes something that has a name, hair color, and a smart phone
- defines the ability to walk upright and use tools (a.k.a. smart phones)

There are roughly seven billion Human *objects* in the world

- each is an *instance* of class Human
- each has its own name, hair color, and smart phone (individual properties)
- each has the ability to walk upright and use tools (shared behavior)

So...

- the class is the <u>template</u> for a human
- an object is an actual human being—an *instance* of the class

Declaring a Class

Classes are generally declared using two files...

A header file:

- has a `.h` extension (e.g., `Point.h`)
- contains the class declaration
- `#includes` any needed libraries
- uses `#pragma once` to prevent multiple inclusions

An implementation file:

- has a `.cpp` extension (e.g., `Point.cpp`)
- contains implementations of class methods and static properties
- `#includes` the class header file

Declaring a Class

Declare a class to represent a 2-dimensional point in space:

```
// header file (Point.h)

#pragma once

<class declaration goes here>


// implementation file (Point.cpp)

#include "Point.h"

<method implementations go here>
```

Declaring a Class

Declare a class to represent a 2-dimensional point in space:

```
// header file (Point.h)

class Point {


};
```

Visibility

Visibility modifiers:

- public properties and methods are accessible everywhere in your program
- private properties and methods are only available from within the class itself

Example:

```
// header file (Point.h)
class Point {
    public:
        // accessible everywhere in the program
    private:
        // accessible only within the class
};
```

Member Variables (Properties)

Member variables are the properties that describe objects of the class

- every object will get its own copy of these variables
- member variables are—by convention—placed in the private section of a class
- these variables cannot be initialized in the class declaration (use constructors instead)

Example:

```
// header file (Point.h)
class Point {
    private:
        double x, y; // member variables
};
```

Member Functions (Methods)

Classes also let us define behaviors that their objects should have

- these behaviors are implemented with functions, called ‘methods’ or ‘member functions’

Methods are called <u>on a specific object</u> using the dot operator:

```
p1.distance_to_origin(); // call method on p1
p2.happy_dance();        // call method on p2
```

the calling object

The object on which a method is called is the <u>calling object</u>

- the method will automagically use that object’s values when executing

Member Functions (Methods)

Example:

```
// header file (Point.h)
class Point {
    public:
        // negates x and y coordinates
        void negate();
    private:
        double x, y;
};
```

To declare the method:

- place the method’s prototype in the class declaration

Member Functions (Methods)

Example:

```
// implementation file (Point.cpp)

void Point::negate() {

    x = -x;

    y = -y;

}
```

To implement the method:

- add the method’s implementation to the implementation file
- prefix the name of the method with the name of the class to which it belongs and the scope-resolution operator (::)

Getters and Setters

If member variables are private, how can they be accessed in a program?

- declare public methods to get and set their values
- because the methods are part of the same class, they have access to private properties
- because the methods are public, they can be used from anywhere in your program

By convention:

- a getter for a property called ‘`prop`’ is named `get_prop` (or `getProp`)
- a setter for the same property is named `set_prop` (or `setProp`)

Getters

Getters allow public, read-only access to private properties:

```
// header file (Point.h)
class Point {
    public:
        // getters for x and y
        double get_x() const;
        double get_y() const;

    private:
        double x, y;
};
```

Getters

Getters allow public, read-only access to private properties:

```
// implementation file (Point.cpp)

// getter for x
double Point::get_x() const {
    return x;
}

// getter for y
double Point::get_y() const {
    return y;
}
```

Setters

Setters allow public modification of private properties:

```
// header file (Point.h)
class Point {
    public:
        // setters for x and y
        void set_x(double new_x);
        void set_y(double new_y);

    private:
        double x, y;
};
```

Setters

Setters allow public modification of private properties:

```
// implementation file (Point.cpp)


// setter for x
void set_x(double new_x) {
    x = new_x;
}


// setter for y
void set_y(double new_y) {
    y = new_y;
}
```

const methods

Take a look at this getter prototype:

```
double get_x() const;
```

Notice the const keyword at the end of it...

- this is simply a promise that this method will not modify the calling object in any way

Simple example:

```
Point p1;

cout << p1.get_x() << endl; // p1 will NOT be modified!
```

the calling object

const methods

Take a look at this getter prototype:

```
double get_x() const;
```

Notice the const keyword at the end of it...

- this is simply a promise that this method will not modify the calling object in any way

A matching const must also appear in the method implementation:

```
double Point::get_x() const {
    return x;
}
```

const methods

Take a look at this getter prototype:

```
double get_x() const;
```

Notice the const keyword at the end of it...

- this is simply a promise that this method will not modify the calling object in any way

...and prevents modifications to the calling object:

```
double Point::get_x() const {
    return x = 42; // compiler error
}
```

Constructors

Constructors are responsible for initializing objects

- think of constructors as object factories that set each property on a new object

Constructors are "special" methods...

- they must have the exact same name as the class to which they belong
- they are implicitly called whenever we create a new object of the class
- unlike other functions, constructors don't have a return type (not even void)

Terminology:

- a default constructor is one that accepts zero arguments
- a parameterized constructor is one that accepts one or more arguments

Constructors

Example constructors for the Point class:

```
// header file (Point.h)
class Point {
    public:
        Point(); // default constructor
        Point(double, double);

    private:
        double x, y;
};
```

Constructors

Example constructors for the Point class:

```
// implementation file (Point.cpp)


// default constructor creates a point at (0,0)
Point::Point() {
    x = y = 0;
}
```

Constructors

Example constructors for the Point class:

```
// implementation file (Point.cpp)


// 2-argument constructor creates a point at (X,Y)
Point::Point(double X, double Y) {
    x = X;
    y = Y;
}
```

Constructors

Given these constructors, you can now create Points like this:

```
// create a Point at (0,0) using default constructor
Point p1; // no ()’s

// create a point at (3,5) using 2-argument constructor
Point p2(3, 5);
```

These are simply variable declarations that use Point as the type!

New Concept:

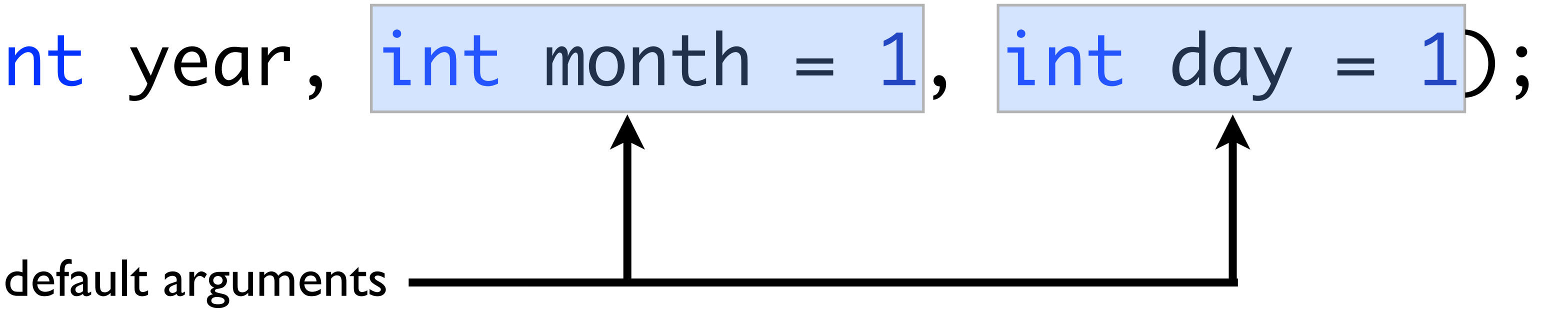
Default arguments

Default Arguments

A function can define defaults for some or all of its arguments

- this essentially makes the argument “optional” when calling the function
- specify default arguments in the prototype; no changes are needed in the implementation

Example:



```
int date_check(int year, int month = 1, int day = 1);
```

default arguments

```
// no defaults specified in the implementation!
int date_check(int year, int month, int day) {
    // code to validate
}
```

Default Arguments

A function can define defaults for some or all of its arguments

- this essentially makes the argument “optional” when calling the function
- specify default arguments in the prototype; no changes are needed in the implementation

Example:

```
int date_check(int year, int month = 1, int day = 1);
```

```
date_check();        // invalid (year isn’t optional)
date_check(2000);    // uses default for month and day
date_check(2002, 7); // uses default for day
date_check(2002, 8, 20);
```

Default Arguments

A function can define defaults for some or all of its arguments

- this essentially makes the argument "optional" when calling the function
- specify default arguments in the prototype; no changes are needed in the implementation

Default arguments must occur AFTER arguments without defaults

```
int date_check(int year = 1, int month, int day = 1);
```

non-default arguments must occur first!

Default Arguments

Revised declaration of constructor that takes default arguments:

```
// header file (Point.h)
class Point {
    public:
        // specify default values for each argument
        Point(double X = 0, double Y = 0);

    private:
        double x, y;
};
```

Default Arguments

Revised* implementation of constructor that takes default arguments:

```
// implementation file (Point.cpp)


// default values are specified in the prototype ONLY
Point::Point(double X, double Y) {
    x = X;
    y = Y;
}
```

* no changes to the implementation to use default arguments!

- default arguments are specified in the prototype alone

Default Arguments

With this one constructor, all three of these statements are valid:

```
// create a Point at (0,0); uses default for x and y
Point p1; // no ()’s


// create a point at (3,0); uses default for y
Point p2(3);


// create a point at (3,5)
Point p2(3, 5);
```

New Concept:

Inline methods

Inline Methods

An inline method is fully implemented inside the class declaration:

```
class MyClass {
    int inline_function() const { return 42; }
};
```

Inline methods are treated slightly differently than normal methods:

- the compiler replaces calls to an inline method with a copy of that function's code
- this eliminates the overhead associated with normal function calls
- however, because the code for the function is duplicated every time the function is called, it also increases the size of the resulting executable

Inline Methods

An inline method is fully implemented inside the class declaration:

```
class MyClass {
    int inline_function() const { return 42; }
};
```

Why use inline methods?

- they generally result in faster execution (yay!)
- however, they *can* slow your program down if the size increases too much (cache misses)
- only use inline methods for SHORT functions (one-liners)

Inline Methods

An inline method is fully implemented inside the class declaration:

```
class Point {
    public:
        double get_x() const { return x; } // inline
        double get_y() const;              // NOT inline
    private:
        double x, y;
};
```

Compare the two getters (x and y):

- get_x (inline) is a complete, fully implemented function
- get_y (not inline) is just a prototype; the implementation is elsewhere

Inline Methods

You could also implement the constructor as an inline method:

```
class Point {
    public:
        // an inline constructor (wow!)
        Point(double X = 0, double Y = 0) {
            x = X;
            y = Y;
        }
    private:
        double x, y;
};
```

New Concept:

Initialization lists

Initialization Lists

Inline constructors can make use of initialization lists:

```
// header file (Point.h)
class Point {
    public:
        Point(double X, double Y) : x(X), y(Y) { }
    private:
        double x, y;
};
```

initialization list

What this does:

- just like before, this sets the new object's x and y properties to X and Y, respectively

Initialization Lists

Inline constructors can make use of initialization lists:

```
Point(double X = 0, double Y = 0) : x(X), y(Y) { }
```

More about this syntax:

- use a colon, followed by a comma-separated list in the form of: `p(v)`, which initializes property `p` to the value `v`
- initialization lists are required to initialize member variables that are const or references
- notice that this is an inline method; you must include the curly braces after the list!

You can still put code inside the method body, too:

```
Point(double X = 0, double Y = 0) : x(X) {
    y = Y;
}
```

New Concept:

Yay for penguins!