

Classes

The (not so) definitive guide

Class Declarations

Syntax for a class declaration (in a .h file):

```
class SuperSexRobot {  
  
};  
  
// remember the semicolon!
```

This creates a **SuperSexRobot** class (a user-defined data type)

- you're essentially teaching the computer about the existence of super sex-robots
- this means you can now create super sex-robots *objects* in your C++ program

public and private

Syntax for defining visibility:

```
class SuperSexRobot {  
    public:  
        // public 'stuff' is visible everywhere in your program  
    private:  
        // private 'stuff' is only visible to this class  
};
```

public and **private** determine the 'visibility' of methods and properties

- **public** means accessible by all parts of your program
- **private** means only accessible to methods belonging to the same class

Properties (data members)

Real-world objects & concepts can be described by certain properties

- classes encapsulate such properties, allowing us to think more abstractly
- each instance of the class gets its own copies of these variables
- we encourage the practice of making all non-constant properties **private**
- data members cannot be initialized in the class declaration (use constructors instead)

Examples:

```
class Cylinder {  
    private:  
        double height; // every cylinder object has its own  
        double radius; // height and radius values  
};
```

Class constants (**static**)

Objects of a class frequently share certain constant values

- having multiple copies of this data would be wasteful
- **static** means that all objects of a class share the exact same variable (no copies)
- since constants cannot be changed, we encourage you to make them **public**
- only integer-type data (**int** and **char**) can be initialized like you see below

Examples:

```
class Triangle {  
    public:  
        // one constant value shared by all Triangles  
        static const int NUM_SIDES = 3;  
};
```

Class constants (**static**)

For non-integer type constants, you have to do a bit more work:

```
class Circle {  
    public:  
        // constant shared by all Circle objects  
        static const double PI; // no value here...  
};
```

Then, in your implementation file (not inside any function):

```
// the name of the constant is prefixed with Circle::  
const double Circle::PI = acos(-1);
```

↑
No **static** keyword here!

↑
name prefixed w/ class

Class constants (**static**)

Same thing for array-type constants:

```
class Car {  
    public:  
        static const int NUM_COLORS = 3; // value okay for ints  
        static const string COLORS[NUM_COLORS];  
};
```

Then, in your implementation file (not inside any function):

```
// the name of both constants is prefixed with Car::  
const string Car::COLORS[Car::NUM_COLORS] = {  
    "red", "green", "penguin"  
};
```

Methods (class functions)

Classes also enable us to define behaviors (class functions):

- we call functions that belong to a class 'methods' or 'member functions'

Creating class methods involves two steps:

- specify the method's prototype in the class declaration
- implement the method in the class implementation file, remembering to prefix the method name with the name of the class and the scope-resolution operator (::)

Methods are called using the dot operator:

```
sally.get_last_name(); // sally, what's your last name?
```

```
bobby.say_hi(sally); // bobby, say hi to sally
```

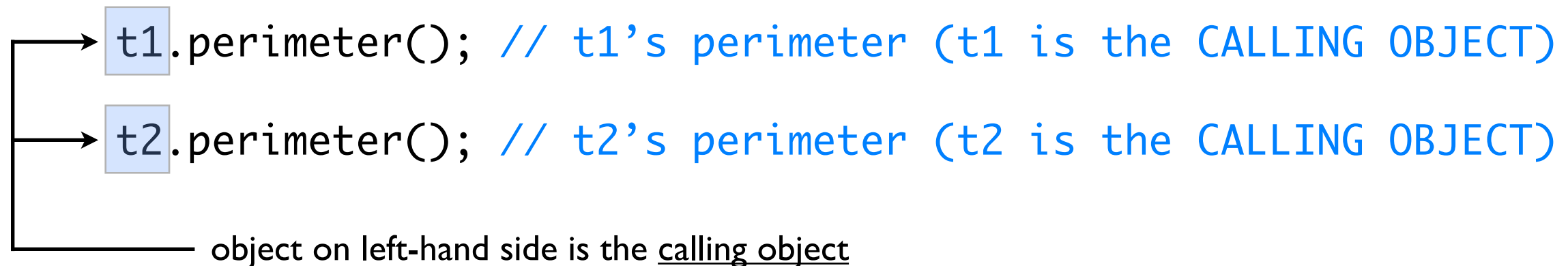

Methods (class functions)

Let's say we have these objects (variables w/ class data type):

```
Triangle t1(3, 4, 5);
```

```
Triangle t2(6, 8, 10);
```

Also, assume the `Triangle` class has a method called `perimeter`:



```
t1.perimeter(); // t1's perimeter (t1 is the CALLING OBJECT)
```

```
t2.perimeter(); // t2's perimeter (t2 is the CALLING OBJECT)
```

object on left-hand side is the calling object

The `perimeter` method operates on the calling object:

- all methods like `perimeter` operate on a specific object and use *that object's values*
- whatever is on the LHS of the dot operator is the object that will be used

Constructors

Constructors are special class methods:

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

Constructors are responsible for initializing objects

- think of constructors as object factories
- constructors set all the relevant properties for each new object

Terminology:

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

Constructors

Place the constructor prototypes in the class declaration (.h file):

```
class Car {  
    public:  
        Car();    // default constructor prototype  
        Car(int); // one-parameter constructor prototype  
    private:  
        int horsepower;  
};
```

C++ *implicitly* calls constructors when creating objects:

```
Car my_car;           // calls the default constructor - NO ()'s!  
Car your_car(200);    // calls the one-parameter constructor
```

Constructors

Implement constructors in the class implementation file:

```
// implementation of the default constructor
```

```
Car::Car() {
```

```
    horsepower = 100; // a reasonable default value
```

```
}
```

```
// implementation of the parameterized constructor
```

```
Car::Car(int hp) {
```

```
    horsepower = hp; // use the user-supplied argument
```

```
}
```

Notice the `Car::` prefix before the function names!

- this tells C++ that these functions are part of the `Car` class
- accordingly, this also allows them access to the `private` horsepower property

Getters and Setters

If data members are **private**, how can they be changed or viewed?

- 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
- getters and setters help you protect the integrity of your class objects (validation)

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`)

Sometimes getters/setters are inappropriate (but follow the rubrics!)

- some properties are only used within the class, so public access isn't necessary
- some, like gender, are static (though I suppose you could allow sex-changes)

Getters and Setters

Prototypes for horsepower getter and setter:

```
class Car {  
    public:  
        Car(); // default constructor  
        int get_horsepower() const; // getter for horsepower  
        void set_horsepower(int); // setter for horsepower  
    private:  
        int horsepower;  
};
```

Accessor methods for the **private int** property horsepower:

- get_horsepower should return an **int**, accept zero arguments, and be **const**
- set_horsepower should be **void** and accept a single **int** argument

Getters and Setters

Prototypes for horsepower getter and setter:

```
class Car {  
    public:  
        Car(); // default constructor  
        int get_horsepower() const; // getter for horsepower  
        void set_horsepower(int);    // setter for horsepower  
    private:  
        int horsepower;  
};
```

Pretty easy, right?

- the getter and setter must occur in the **public** section
- note the similarity of the method names to the data member they serve

Getters and Setters

Implementation of getter/setter:

```
// getter for horsepower
```

```
int Car::get_horsepower() const {
```

```
    return horsepower; // simply return the property's value
```

```
}
```

```
// setter for horsepower
```

```
void Car::set_horsepower(int h) {
```

```
    horsepower = h; // set the new value if it's valid
```

```
}
```

Again, note the `Car::` prefix (these methods are part of the `Car` class)

- it occurs before the name of the function, not before the return type

const methods

Take a look at this getter prototype:

```
int get_horsepower() const;
```

Notice the `const` keyword at the end of it

- this is simply a promise that this method will not change the calling object in any way
- a getter simply returns the value of a property; it won't actually change anything
- a setter, on the other hand, exists to change the object; it cannot be `const`

A matching `const` must also appear in the method implementation:

```
int Car::get_horsepower() const { // const here, too!  
    return horsepower;  
}
```

Helper Functions

Helper functions are **private** class methods

- they often contain logic or code that would otherwise be repeated in **public** methods

Example (helper function for validation):

```
class Penguin {  
    public:  
        Penguin(int how_awesome);  
        void set_awesome(int how_awesome);  
    private:  
        int awesomeness;  
        // helper method used by both constructor and setter  
        bool is_valid(int proposed_awesome) const;  
};
```

Input and Output Methods

There are many ways to do I/O with objects

- one such way is with public class methods, which accept an appropriate stream argument (`istream` or `ostream`) by reference and return it (also by reference)

Output example:

```
class Pirate {  
    public:  
        Pirate(string); // default constructor  
        // silly output function to have pirate say something  
        ostream& say_something(ostream&) const;  
    private:  
        string name;  
};
```

Input and Output Methods

There are many ways to do I/O with objects

- one such way is with public class methods, which accept an appropriate stream argument (`istream` or `ostream`) by reference and return it (also by reference)

Output example:

```
ostream& Pirate::say_something(ostream& out) const {  
    out << name << " says Arrrrgh, matey!"; // no endl  
    return out; // return the stream object (by reference)  
}
```

Notice:

- the `ostream` object is both passed and returned by reference!
- established etiquette says that you *should not* include an `endl` at the end of your output

Input and Output Methods

Input is a bit more involved...

- read from the `istream` object into local variables
- validate the values (set the `istream`'s failbit if an error is detected)
- store the validated values into the object's data members

Example:

```
class Cone {  
    public:  
        // read the cone's dimensions from the given input stream  
        istream& get_dimensions(istream&);  
    private:  
        double height, radius;  
};
```

Input and Output Methods

Input implementation:

```
istream& Cone::get_dimensions(istream& in) {  
    double temp_h, temp_r; // temporary local variables  
  
    // attempt to read values and ensure that both are >= 0  
    if (!(in >> temp_r >> temp_h) || temp_r < 0 || temp_h < 0) {  
        in.setstate(ios::failbit); // set stream into fail mode  
    } else {  
        height = temp_h;           // otherwise, these are valid values  
        radius = temp_r;           // update the object's properties  
    }  
  
    return in; // always return the stream object  
}
```

Arrays of Objects

Creating an array with explicit size:

- each element in an array of objects gets initialized with the default constructor
- this means you **MUST** have a default constructor declared for your code to compile

Example of an array of objects:

```
// each element is created using the default constructor
```

```
Player my_team[100];
```

```
// same story here... doesn't work without default constructor
```

```
Player your_team[100] = {};
```

Arrays of Objects

You can use parameterized constructors with initialization lists

- each constructor has to be defined, and a default constructor is necessary if explicit size

This works if an appropriate 2-argument constructor is defined:

```
Player my_team[] = {  
    Player(10, 5), Player(10, 5), Player(10, 5), Player(10, 5)  
};
```

This requires that the default constructor be defined, too:

```
Player my_team[10] = {  
    Player(10, 5), Player(10, 5), Player(10, 5), Player(10, 5)  
};
```


Arrays of Objects

Assume that a Person class exists

- each person has a name (a `string`) and a corresponding getter method

Also assume we have this array of `Person` objects:

```
Person family[] = {  
    Person("Sven"), Person("Romeo"),  
    Person("Nero"), Person("Merlin")  
};
```

This will print the name of each person in the array:

```
// each element in array is a person object  
for (int i = 0; i < 4; i++) {  
    cout << family[i].get_name() << endl; // cout name of current Person  
}
```

Arrays as Data Members

You can declare arrays as data members belonging to a class

- each object of the class will have its own distinct array
- the values in the array should be initialized in the constructor, probably using a for loop

Assume this class declaration:

```
class GradeTracker {  
    public:  
        static const int NUM_ASSIGNMENTS = 40; // array size  
        GradeTracker();  
        GradeTracker(string grade_file);  
    private:  
        double grades[NUM_ASSIGNMENTS]; // array of 40 possible scores  
        int num_grades;                  // actual number of scores  
};
```

Arrays as Data Members

Example default constructor implementation:

```
// default constructor initializes all grades to 0
GradeTracker::GradeTracker() {
    // initialize all elements to 0
    for (int i = 0; i < NUM_ASSIGNMENTS; i++) {
        grades[i] = 0;
    }
    num_grades = NUM_ASSIGNMENTS;
}
```

The num_grades data member keeps track of actual number of grades

- in the case of the default constructor, this is the number of possible assignments
- though it could be whatever we decided the default values should be...

Arrays as Data Members

Example parameterized constructor implementation:

```
// parameterized constructor initializes all grades reading from the file
GradeTracker::GradeTracker(string file) {
    // open input file and check for error
    ifstream infile(file.c_str());
    if (!infile) {
        exit(1);
    }
    // keep count of how many actual grades were read
    num_grades = 0;
    // read elements from file initialize all elements to 0
    for (int i = 0; i < NUM_ASSIGNMENTS && infile >> grades[i]; i++) {
        num_grades++;
    }
}
```

Arrays as Data Members

Notice the format of the array getter and setter:

```
class GradeTracker {  
    public:  
        static const int NUM_ASSIGNMENTS = 40;  
        GradeTracker();  
        GradeTracker(string grade_file);  
        // setter and getter for 'grades' array (individual elements only)  
        double get_grade(int index) const;           // accepts an index argument  
        void set_grade(int index, double value); // accepts an index argument  
  
    private:  
        double grades[NUM_ASSIGNMENTS];  
        int num_grades;  
};
```

Arrays as Data Members

Notice the format of the array getter and setter:

```
// grade getter method (specify index of interest)
double GradeTracker::get_grade(int index) const {
    if (index >= 0 && index < num_grades) {
        return grades[index];    // only return values at valid indexes
    }
    return 0; // default value (or exit on error)
}

// grade setter method (specify index to set and the new value)
void GradeTracker::set_grade(int index, double value) {
    if (index >= 0 && index < num_grades) {
        grades[index] = value;    // only change value if at valid index
    }
}
```

Object Arguments of Same Class

Assume this class declaration:

```
class Point {  
    public:  
        Point(int, int);  
        Point reflect(const Point&) const;  
    private:  
        int x, y; // x and y coordinates  
};
```

Notice the reflect() function

- it accepts a Point object as a pass-by-reference argument
- the function creates a new point object (reflected across the origin) and returns a copy of that new object

Object Arguments of Same Class

reflect implementation:

```
Point Point::reflect(const Point& p) const {  
    // creates a new local Point object, which gets returned  
    return Point(  
        -(p.x), // negative of p's x coordinate  
        -(p.y)  // negative of p's y coordinate  
    );  
}
```

This function creates a new Point object using an existing one

- it accepts a Point object as a pass-by-reference argument
- the function creates a new point object (reflected across the origin), which it then returns
- because p is a Point object, this function (also part of the Point class) can directly access p's private properties using the dot operator