C++ Programming

A Few Other Things

A Few Other Things

- The this Pointer
- Static Members
- Default Parameters

The this Pointer

- Every object in C++ has access to its own address through the use of a pointer called `this`

- The `this` pointer is an implicit parameter to all member functions, and so inside a member function, the `this` pointer can be used to refer to the invoking object

The this Pointer

```cpp
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
public:
    Rectangle(int width, int height) {
        width = width;
        height = height;

    }
};

int main() {
    Rectangle r(10, 20);
}
```

What happens here?

The this Pointer

```cpp
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
public:
    Rectangle(int width, int height) {
        this->width = width;
        this->height = height;

    }
};

int main() {
    Rectangle r(10, 20);
}
```

Much better!

Static Members

- By declaring a class member as static, it is possible to have that member belong to the class itself, as opposed to individual objects of the class

- Such static members, in a way, are essentially shared among all objects of the class

- Static members can also be accessed without requiring the instantiation of an object in advance
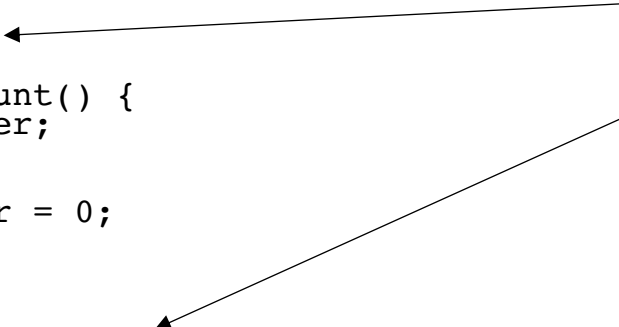
Static Members

```cpp
#include <iostream>
using namespace std;

class Person {
private:
    static int counter;
public:
    Person() {
        counter++;
    }
    int getPersonCount() {
        return counter;
    }
};
int Person::counter = 0;

int main() {
    Person p1;
    Person p2;
    cout << p2.getPersonCount() << endl;
}
```

Our counter variable goes up every time we create an object of the class and we can access it to see how many we have made so far!

Static Members

- When we declare a member variable as `static`, we are simply telling the class that it exists

- As it is not attached to any object, no storage is allocated for it from only its class declaration

- As a result, `static` member variables need to also be declared separately to instantiate them and, optionally, initialize them

Static Members

- We can also have `static` member functions as part of our classes

- Again, these can be used without having to instantiate objects

- As these functions are not attached to particular objects, they do not have a `this` pointer

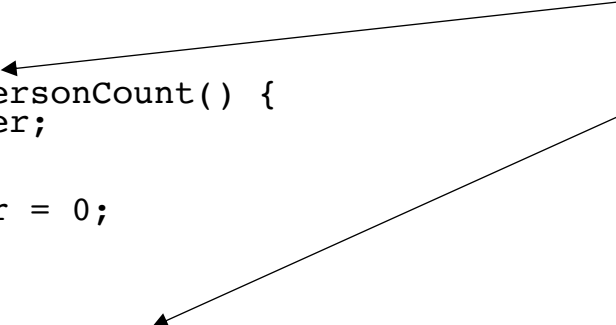# Static Members

```cpp
#include <iostream>
using namespace std;

class Person {
private:
    static int counter;
public:
    Person() {
        counter++;
    }
    static int getPersonCount() {
        return counter;
    }
};
int Person::counter = 0;

int main() {
    Person p1;
    Person p2;
    cout << Person::getPersonCount() << endl;
}
```

Because our accessor method to retrieve the count of objects created is now `static`, we no longer need to use one of our instances to access it …

Static Members

- There are some caveats to creating static members that should be kept in mind whenever using them
    - Static member variables are more-or-less the same as having global variables, albeit potentially with some access restrictions
    - Because static member variables are essentially shared between all objects of a class, they represent a threat to thread-safeness and access to them needs to be regulated like other globally shared data

Default Parameters

- Default parameters allow functions to be called without providing one or more trailing parameters

- Instead of explicitly providing such a parameter, a default value is substituted in its place

- This default value is specified in the declaration of the function

- Both regular functions and member functions of classes can use default parameters

Default Parameters

```cpp
#include <iostream>
using namespace std;

void printNumber(int i=0) {
    cout << i << endl;
}

int main() {
    printNumber(2);
    printNumber(1);
    printNumber();
}
```

As no parameter is given to our `printNumber()` function on the last call to it, it will use the default value of 0 for that call of the function.

Default Parameters

- Once a default parameter is specified in a parameter list for a function, all subsequent parameters must also have default values

- As another important note, default parameters can cause some attempts to overload a function to fail because it creates ambiguity between the two functions …

Default Parameters

```cpp
#include <iostream>
using namespace std;

void printNumber(int i=0) {
    cout << i << endl;
}

void printNumber() {
    cout << "?" << endl;
}

int main() {
    printNumber(2);
    printNumber(1);
    printNumber();
}
```

How would the compiler know which version of `printNumber()` to use? The one with no parameter or the one with a default parameter? It wouldn't know what to do, so this would not be allowed ...