EECS 402 Discussion 13!

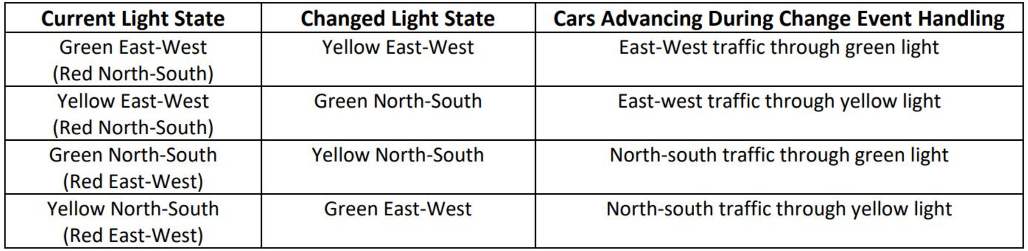
P5 overview, Polymorphism, .inl files, ternary operator, enum

Project 5

Overview

Creating a simulation of a traffic light

Handles car arrival and light changes



Current Light State	Changed Light State	Cars Advancing During Change Event Handling
Green East-West (Red North-South)	Yellow East-West	East-West traffic through green light
Yellow East-West (Red North-South)	Green North-South	East-west traffic through yellow light
Green North-South (Red East-West)	Yellow North-South	North-south traffic through green light
Yellow North-South (Red East-West)	Green East-West	North-south traffic through yellow light

Functions TODO

- Template FIFOQueue and SortedListClass from p4
- Overload operators needed in eventClass
- Function in intersection simulations labeled “THIS FUNCTION NEEDS TO BE IMPLEMENTED”
    - scheduleArrival
    - scheduleLightChange
    - handleNextEvent

Project Tips

Read the spec and given code *really well* before starting

Most of the code is done for you- you only have to implement a couple functions

Taking the time to understand what you are doing will save you debugging time

IO Redirection

Used to redirect input

Use a text file instead of typing into the command line

```
./programName {any arguments needed} < input.txt > output.txt
```

https://www.diffchecker.com/diff

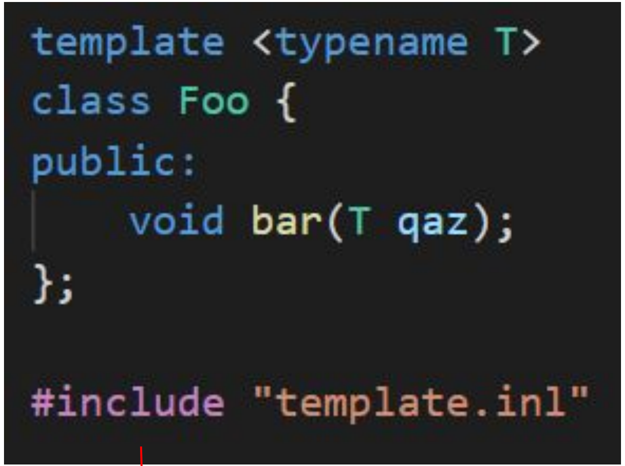
Compiling Templated Files

Compiling Templates

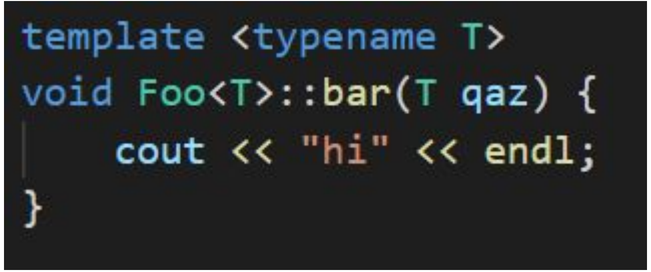
C++ requires template definitions be included with the declarations

- We still want to separate the interface from the implementations!

Solution: include the definitions at the bottom of your header file!



```cpp
template <typename T>
class Foo {
public:
    void bar(T qaz);
};

#include "template.inl"
```



```cpp
template <typename T>
void Foo<T>::bar(T qaz) {
    cout << "hi" << endl;
}
```

Polymorphism

Principles of Object Oriented Programming

- Encapsulation
  - We want data and the functions that edit that data to be in the same place
  - Classes!
- Inheritance
  - We want a way to relate sets of data and functionality to each other
  - Child / parent classes!
- **Polymorphism**
  - **We want to allow those sets of data to interact more “smoothly”**
  - **Generic base class pointers!**

Big idea #1

- I want a list of vehicles where each vehicle can be any derived class
- That way, I can iterate through them easily

Big idea #1

- I want a list of vehicles where each vehicle can be any derived class
- That way, I can iterate through them easily
- Solution: Just use pointers to a base class object

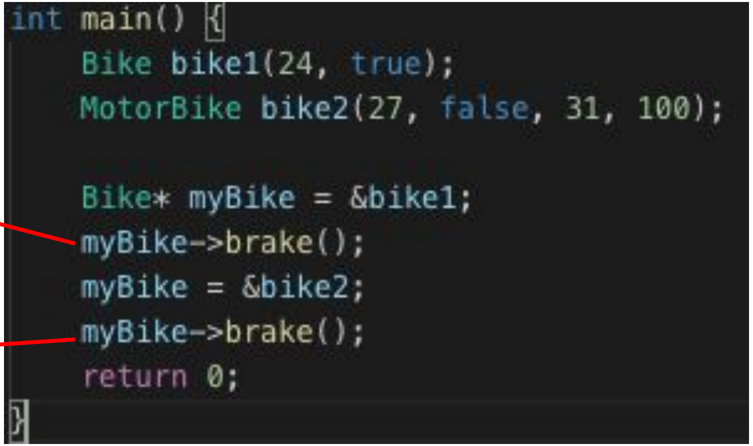
Static and Dynamic typing

Static type: Bike

Dynamic type: Bike

Static type: Bike

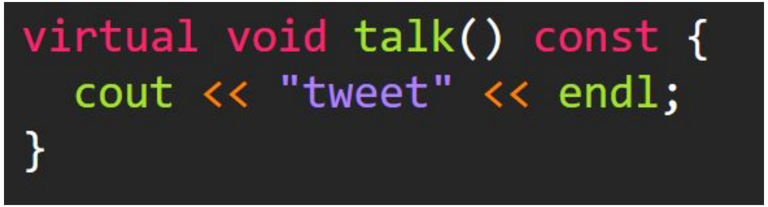
Dynamic type: MotorBike



```
int main() {
    Bike bike1(24, true);
    MotorBike bike2(27, false, 31, 100);

    Bike* myBike = &bike1;
    myBike->brake();
    myBike = &bike2;
    myBike->brake();
    return 0;
}
```

Virtual and Pure Virtual Functions

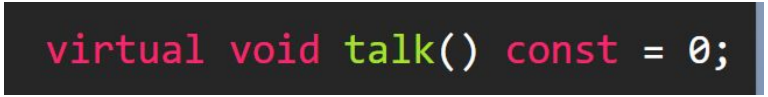
Virtual functions- can be overridden



```cpp
virtual void talk() const {
  cout << "tweet" << endl;
}
```

Pure Virtual Functions- MUST be overridden

Make a class abstract



```cpp
 virtual void talk() const = 0;
```

Types of Polymorphism

ad hoc Polymorphism

- Function overloading
- Operator overloading

Parametric Polymorphism

- Templates

Subtype Polymorphism

- Using derived-class objects when a base-class object is expected

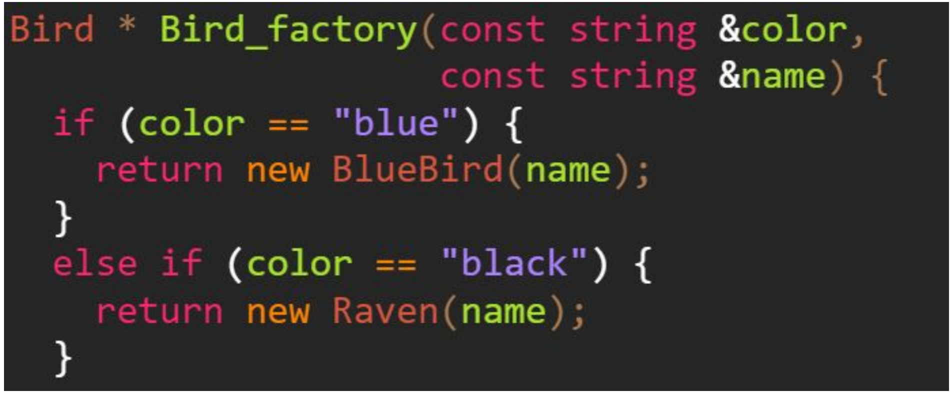
The Factory Pattern

A factory function is a function that creates and returns objects

- Uses subtype polymorphism

Helps separate interface from implementation even more!

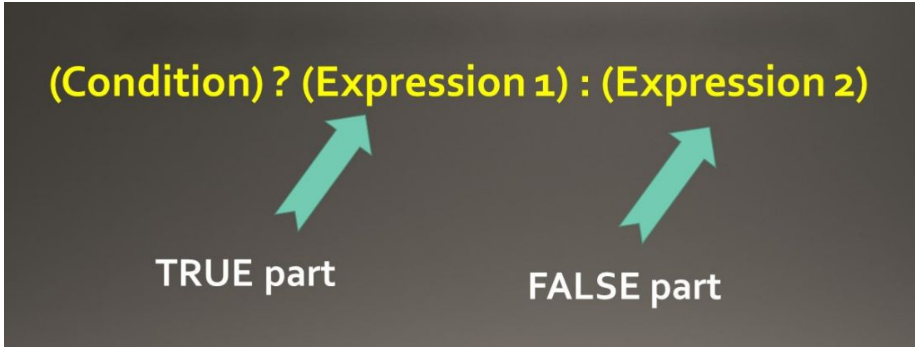
- User never needs to know that “Bluebirds” or “Ravens” exist, they just need to know they’re using a type of Bird!



```
Bird * Bird_factory(const string &color,
                    const string &name) {
  if (color == "blue") {
    return new BlueBird(name);
  }
  else if (color == "black") {
    return new Raven(name);
  }
```

Ternary Operator

Ternary Operator

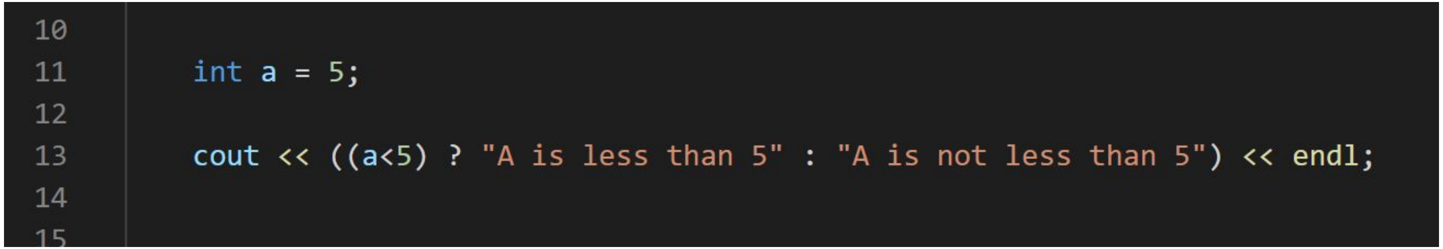


(Condition) ? (Expression 1) : (Expression 2)

TRUE part

FALSE part

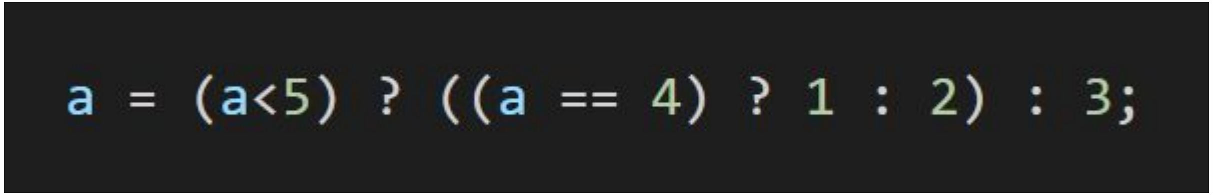
- Allows you do use conditionals in one line
- Often replaces if statements



```

    int a = 5;

    cout << ((a<5) ? "A is less than 5" : "A is not less than 5") << endl;


```

Example

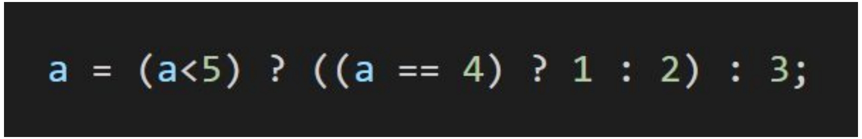
What would this be translated to if/else if statements?



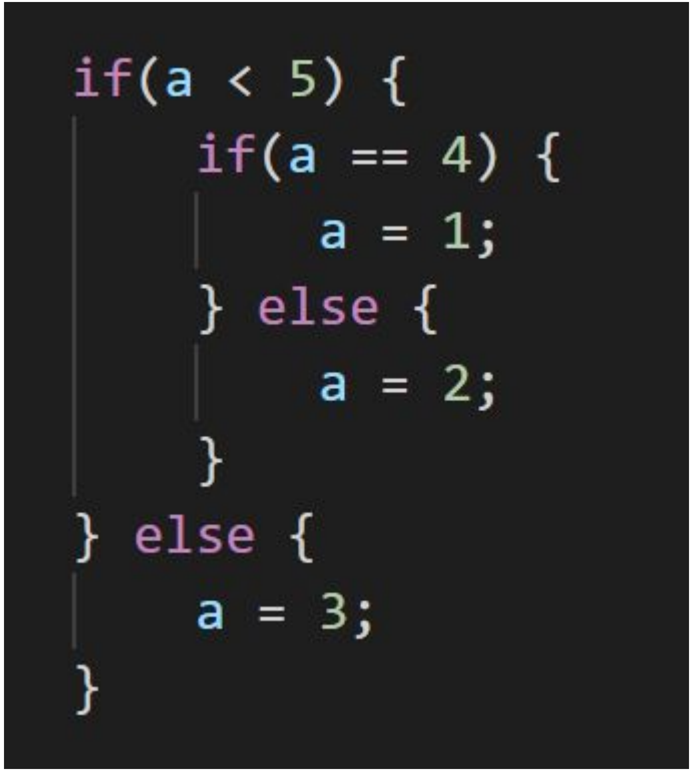
```
a = (a<5) ? ((a == 4) ? 1 : 2) : 3;
```

Example Solution

What would this be translated to if/else if statements?

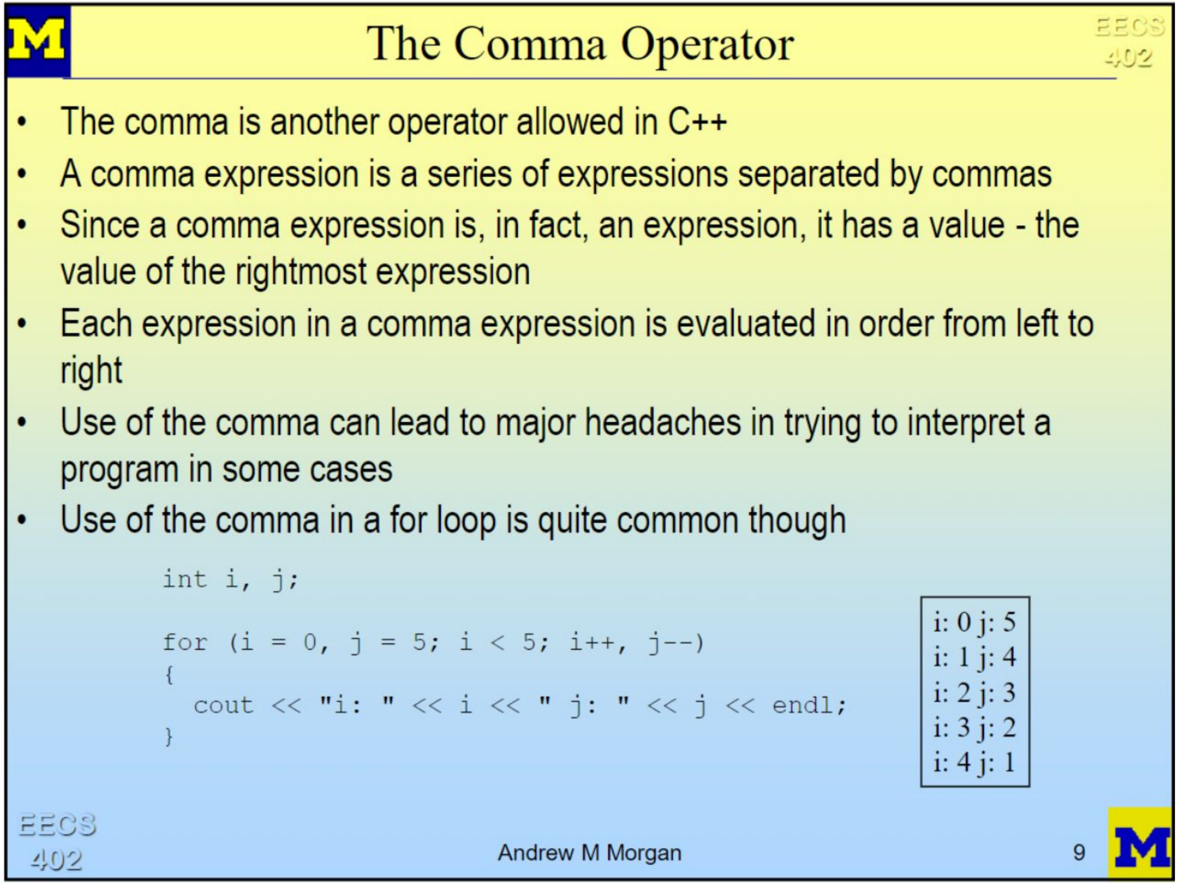


```
a = (a<5) ? ((a == 4) ? 1 : 2) : 3;
```



```
if(a < 5) {
    if(a == 4) {
        a = 1;
    } else {
        a = 2;
    }
} else {
    a = 3;
}
```

Comma Operator



The Comma Operator

- The comma is another operator allowed in C++
- A comma expression is a series of expressions separated by commas
- Since a comma expression is, in fact, an expression, it has a value - the value of the rightmost expression
- Each expression in a comma expression is evaluated in order from left to right
- Use of the comma can lead to major headaches in trying to interpret a program in some cases
- Use of the comma in a for loop is quite common though

```
int i, j;

for (i = 0, j = 5; i < 5; i++, j--)
{
  cout << "i: " << i << " j: " << j << endl;
}
```

```
i: 0 j: 5
i: 1 j: 4
i: 2 j: 3
i: 3 j: 2
i: 4 j: 1
```

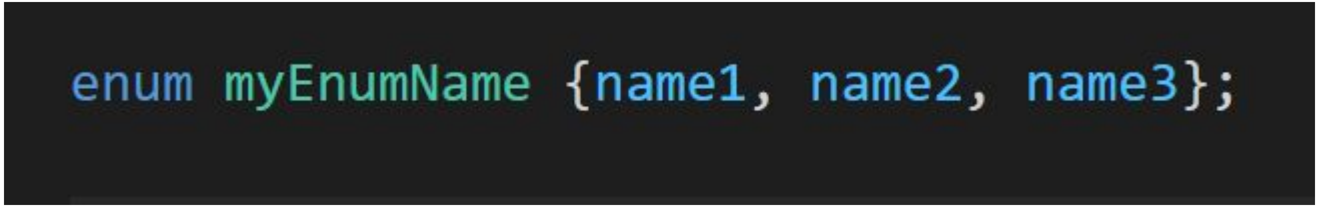
Enums

Enum Types

- Allows you to name your own variable types
- Declares a list of possible options for each variable of this type
- Can be used in switch statements (evaluates to an integer)

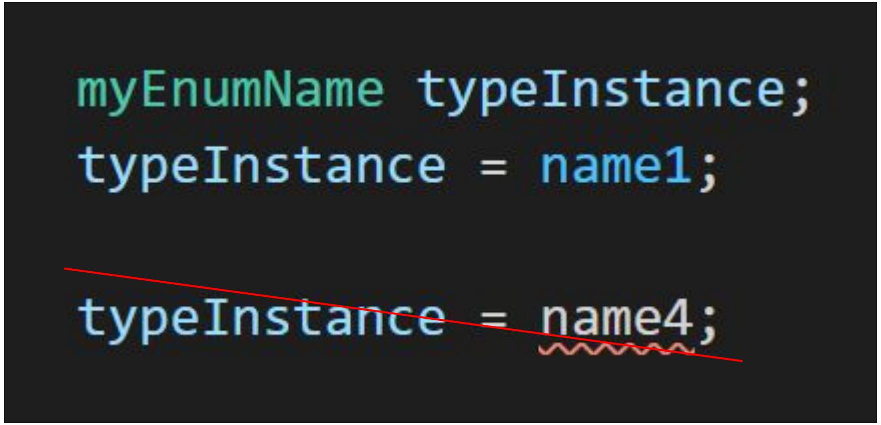
Enum Types

- To declare



```
enum myEnumName {name1, name2, name3};
```

- To use



```
myEnumName typeInstance;
typeInstance = name1;
```

~~typeInstance = name4;~~