


Learning Objectives: Advanced Topics

- Include and create an object defined in a separate file
- Understand what the scope resolution operator does 
- Manipulate class variables not attached to a class object
- Create structs and enums
- Compare objects

Using #include

Objects Defined in Other Files

Now that you're familiar with creating, manipulating, and inheriting objects, we'll discuss how these concepts all come together when programs are created. You may have noticed that a lot of the classes that you've previously worked with is fairly lengthy. In fact, it's not typical to include class definitions within the same file as the main program. Usually, classes are defined in other files but they can be accessed by the main program if you set up the correct encapsulation.

"#include" in the Header

Ever noticed how `#include <iostream>` and `using namespace std;` are usually present in the header of the file? The `#include` directs the system to look for the file or library `iostream`. Then `using namespace std;` enables the system to access functions from the class `std` without having to use the **scope resolution operator** `::`.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5
6      //add code below this line
7
8
9
10     //add code above this line
11
12     return 0;
13
14 }
```

.guides/img/advanced/Include

Enter the following into main and Try It.

```
cout << "Hello world" << endl;
```

Next, comment the header line using `namespace std;` and run the program again.

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

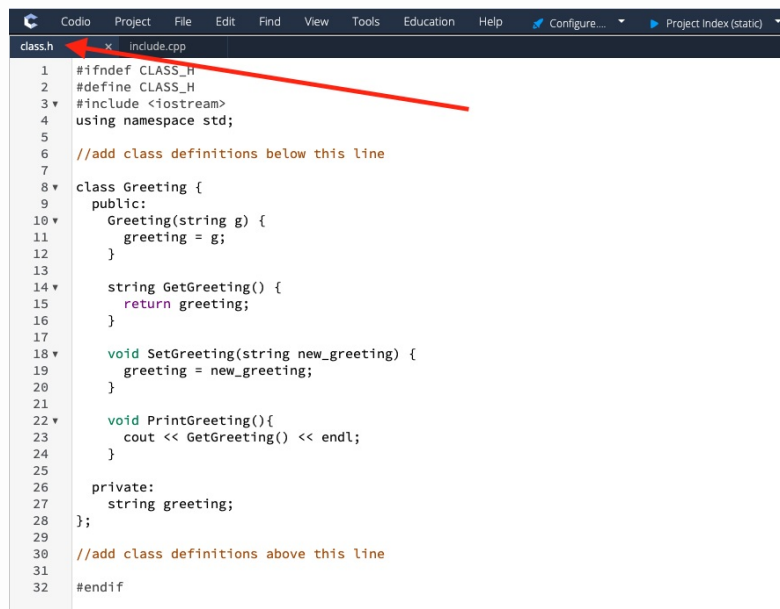
Without using `namespace std;`, the system has no clue which class to look into to use `cout`. Now revise the code in `main` to look like this:

```
std::cout << "Hello world" << std::endl;
```

The scope resolution operator `::` does the same job as using `namespace std;`. Note that you will need to use `std::` in front of every function you decide to call.

Calling Functions From a Separate File

In the upper left of the text editor window, you will notice a tab that says `class.h`. Click on this tab to see the content of the file.



[.guides/img/advanced/HeaderFile](#)

This is a header file which you can use to store your class definitions. There is some syntax (at the beginning and the end) that you have to include in order to specify that `class.h` is a header file, but ultimately you can reduce the number of code in your `main` program by creating header files. All header files should have `#ifndef` and `#define` (followed by their file name and then an underscore `_` and `H`) in their header and `#endif` towards the end.

Now, go back into `include.cpp`, copy the entire code below and TRY IT.

```
//#include <iostream>
//using namespace std;
#include "class.h"

int main() {

    //add code below this line

    Greeting g("Hello world");
    cout << g.GetGreeting() << endl;
    g.SetGreeting("Hi world");
    cout << g.GetGreeting() << endl;

    //add code above this line

    return 0;

}
```

Notice how we used `#include "class.h"` in the header of our file. This enables a connection between the main file and the header file `class.h` where the class definitions are found. Also note that because `#include <iostream>` and `using namespace std;` are already included in `class.h`, you can comment out or remove them in the main program.

Static Variables & Functions

Manipulating Non-Object Variables

Throughout this course, we've been creating class objects through the use of a constructor, which is required in order for an object to access certain class functions. However, if we want to manipulate a particular class attribute without having to instantiate that particular object, you can use a combination of the `static` keyword plus the scope resolution operator `::` to manipulate attributes. Copy the code below and TRY IT.

```

#include <iostream>
using namespace std;

//add class definitions below this line

class Player {
public:
    Player() {
        health = 100;
        score = 0;
        level = 1;
    }
    void PrintLevel() {
        cout << level << endl;
    }
    static int ChangeLevel(int change) { //define static
        function
        level = change;
        return level;
    }

private:
    int health;
    int score;
    static int level; //declare static variable
};

//add class definitions above this line

int Player::level = 0; //italize static variable globally

int main() {

    //add code below this line

    Player mario;
    mario.PrintLevel(); //calling class function, object required
    cout << Player::ChangeLevel(5) << endl; //calling static
        function, object not needed

    //add code above this line

    return 0;

}

```

Notice how when calling the ChangeLevel function, we needed the class name Player followed by the scope resolution operator :: followed by the

function name `ChangeLevel` and any parameter arguments. Calling a static function enabled us to change the attribute `level` to 5 without calling the function on the object itself.

When defining and calling static functions, keep the following in mind:

- * Within a class, static functions can only access other static members; for example, if `level` was not a static variable in the example above, `ChangeLevel` will not be able to access it.
- * A static variable should be defined or initialized globally *outside* of any class or function. In other words, a static variable should **not** be initialized within a class or within the `main` function.
- * Use the scope resolution operator `::` as defined above to access static functions that modify static variables.

Static Variables

It's important to note that static variables are variables that are created only once. They cannot be created again for the duration of the program. The following code will showcase how the static variable `level` works throughout the program.

```

#include <iostream>
using namespace std;

//add class definitions below this line

class Player {
public:
    Player() {
        health = 100;
        score = 0;
    }
    void PrintLevel() {
        cout << level << endl;
    }
    static int ChangeLevel(int change) {
        level = change;
        return level;
    }

private:
    int health;
    int score;
    static int level;
};

//add class definitions above this line

int Player::level = 5; //initialize static variable level to 5

int main() {


    //add code below this line


    Player mario;
    mario.PrintLevel();
    cout << Player::ChangeLevel(6) << endl;
    Player luigi;
    luigi.PrintLevel();


    //add code above this line


    return 0;
}

```









Again, note that static variables are initialized **outside** of a class globally. This is to prevent the variable from being duplicated. By definition, static variables are only created once. This is why when the Player object luigi is created, its level attribute is the same as mario's even though its level was never changed. Essentially, both objects *share* the same static variable level. Changes made to level will be reflected in all objects that have that attribute.

Structs & Enums

What are Structs?

Structs are like classes, except everything inside a struct is **public**. Therefore, anything external to the struct can alter its members. For example:

```
//add class definitions below this line

struct Person {
    string name;
    int age;
    double salary;
};

//add class definitions above this line

int main() {

    //add code below this line

    Person p;
    p.age = 50;
    cout << p.age << endl;

    //add code above this line

    return 0;

}
```

In the example above, after creating the struct called `Person`, we are able to access the `age` attribute by simply using dot notation and specifying the attribute. If security is not an issue, structs are useful at putting together a collection of attributes that are highly modifiable and accessible.

Setting Values with Enums

Enums are similar to **switch-case** statements where particular values are assigned to particular cases. See example below:

```

//add class definitions below this line

enum grades {A = 90, B = 80, C = 70, D = 60};

//add class definitions above this line

int main() {

    //add code below this line

    grades grade;

    grade = A;
    cout << "Grade = " << grade << endl;

    //add code above this line

    return 0;

}

```

Cases (variables) of the alphabet from A through D have been assigned to variable integers from 90 backwards to 60 in increments of 10. When you create a grades enum, you can assign it to any of the cases specified in the enum definition. This will signal to the system to assign the grades enum to appropriate integer equivalent.

NOTE: Enum values must be **integers**. For example, you can't create an enum with *string* values.

challenge

Try this variation:

- Change enum grades {A = 90, B = 80, C = 70, D = 60}; in the code above to enum grades {A, B, C, D};. → *Grade = 0*
- Change grade = A; to grade = B;. → *Grade = 1*
- Change enum grades {A, B, C, D}; to enum grades {A = 90, B, C, D};. → *Grade = 91*
- Change enum grades {A = 90, B, C, D}; to enum grades {A, B, C = 5, D};. → *Grade = 1*
- Change grade = B; to grade = D;. → *Grade = 6*

NOTE: If the enum variables are not assigned values, the first variable will be 0 by default. The second variable will be 1 and so on. Additionally, the incrementation of 1 happens following the first value of the initialized enum variable. For example, `enum fruit {apple = 5, banana, orange};` causes apple to have a value of 5 and each variable after will have a value incremented by 1. This means banana will have a value of 6 and orange will have a value of 7. On the other hand, `enum fruit {apple, banana = 5, orange};` causes apple to be 0, banana to be 5, and orange to be 6.

Object Equality

Comparing Object Types

You can compare object types by implementing the `typeid` function. For example, the code below showcases the comparison between two `Player` objects called `mario` and `luigi` using `typeid`.

```
//add class definitions below this line

class Player {
public:
    Player() {
        health = 100;
        score = 0;
        level = 1;
    }

private:
    int health;
    int score;
    int level;
};

//add class definitions above this line


int main() {

    //add code below this line

    Player mario;
    Player luigi;
    cout << boolalpha;
    cout << (typeid(mario) == typeid(luigi)) << endl;

    //add code above this line

    return 0;

} 
```

Since both `mario` and `luigi` are of the class `Player`, their `typeid` will be the same. This is why `cout << (typeid(mario) == typeid(luigi)) << endl;` returns `true`. Unfortunately, `typeid` does not check to ensure that both objects contain the same exact attribute values. However, you can create a user-defined class to check for that.

Comparing Same-Class Objects

Let's create a static member function called `ComparePlayers`. This function takes in two `Player` objects as parameters, then checks to see if each of their attributes is equal to the other. If their attributes are equal, `true` is returned. Else, `false` is returned. Another member function called `NextLevel` is also created.

```
//add class definitions below this line

class Player {
public:
    Player() {
        health = 100;
        score = 0;
        level = 1;
    }

    static bool ComparePlayers(Player p1, Player p2) {
        if ((p1.health == p2.health) &&
            (p1.score == p2.score) &&
            (p1.level == p2.level)) {
            return true;
        }
        else {
            return false;
        }
    }

    void NextLevel() {
        level++;
    }

private:
    int health;
    int score;
    int level;
};

//add class definitions above this line
```

```

int main() {

    //add code below this line

    Player mario;
    Player luigi;
    cout << boolalpha;
    cout << Player::ComparePlayers(mario, luigi) << endl;

    //add code above this line

    return 0;

}

```

→ true

challenge

Try this variation:

- Replace the code in main with:

```

//add code below this line

Player mario;
Player luigi;
cout << boolalpha;
mario.NextLevel();
cout << Player::ComparePlayers(mario, luigi) << endl;

//add code above this line

```

→ false

Notice how when mario's level changed, the ComparePlayers function returns false when mario and luigi are compared.