

Lecture 8: const and Classes

We've seen several examples of **const** used on normal variables to prevent them from being modified. Now that we know about classes, we can introduce some other uses of **const**: on member functions, on member function arguments, and on objects. These concepts work together to provide some surprising benefits.

const Member Functions

A **const** member function guarantees that it will never modify any of its class's member data.

The CONSTFU program shows how this works.

```
//constfu.cpp
//demonstrates const member functions
/
class aClass
{
private:
    int alpha;
public:
    void nonFunc()           //non-const member function
        { alpha = 99; }     //OK

    void conFunc() const     //const member function
        { alpha = 99; }     //ERROR: can't modify a member
};
```

The non-const function **nonFunc()** can modify member data **alpha**, but the constant function **conFunc()** can't. If it tries to, a compiler error results.

A function is made into a constant function by placing the keyword **const** after the declaratory but before the function body. If there is a

separate function declaration, **const** must be used in both declaration and definition. Member functions that do nothing but acquire data from an object are obvious candidates for being made **const**, because they don't need to modify any data.

A Distance Example

To avoid raising too many subjects at once we have, up to now, avoided using **const** member functions in the example programs. However, there are many places where **const** member functions should be used. For example, in the Distance class, shown in several programs, the **showdist()** member function could be made **const** because it doesn't (or certainly shouldn't!) modify any of the data in the object for which it was called. It should simply display the data.

Here's the listing for ENGCONST. Note that **const** is used in both the declaration and the definition of **add_dist()**:

```
// engConst.cpp
// const member functions and const arguments to member functions
#include <iostream>
using namespace std;
/////////////////////////////////////////////////////////////////
class Distance                                     //English Distance class
{
private:
    int feet;
    float inches;
public:
    //constructor (no args)
    Distance() : feet(0), inches(0.0)
    { }
    //constructor (two args)
    Distance(int ft, float in) : feet(ft), inches(in)
    { }

    void getdist()                                //get length from user
    {
        cout << "\nEnter feet: "; cin >> feet;
        cout << "Enter inches: "; cin >> inches;
    }
    void showdist() const                          //display distance
    { cout << feet << "'-' " << inches << "'"; }

    Distance add_dist(const Distance&) const;      //add
};
//-----
//add this distance to d2, return the sum
Distance Distance::add_dist(const Distance& d2) const
{
    Distance temp;                                //temporary variable

    // feet = 0;                                //ERROR: can't modify this
    // d2.feet = 0;                             //ERROR: can't modify d2
    temp.inches = inches + d2.inches;             //add the inches
    if(temp.inches >= 12.0)                       //if total exceeds 12.0,
    {                                              //then decrease inches
        temp.inches -= 12.0;                     //by 12.0 and
        temp.feet = 1;                           //increase feet
    }                                              //by 1
    temp.feet += feet + d2.feet;                 //add the feet
    return temp;
}
/////////////////////////////////////////////////////////////////
```

```
int main()
{
    Distance dist1, dist3;           //define two lengths
    Distance dist2(11, 6.25);       //define, initialize dist2

    dist1.getdist();                //get dist1 from user
    dist3 = dist1.add_dist(dist2);  //dist3 = dist1 + dist2

                                   //display all lengths
    cout << "\ndist1 = "; dist1.showdist();
    cout << "\ndist2 = "; dist2.showdist();
    cout << "\ndist3 = "; dist3.showdist();
    cout << endl;
    return 0;
}
```

Here, `showdist()` and `add_dist()` are both constant member functions. In `add_dist()` we show in the first commented statement, `feet = 0`, that a compiler error is generated if you attempt to modify any of the data in the object for which this constant function was called.

const Member Function Arguments

The argument should be made `const` in the function declaration (and definition). This is true of member functions as well.

In `ENGCONST` the argument to `add_dist()` is passed by reference, and we want to make sure that `ENGCONST` won't modify this variable, which is `dist2` in `main()`. Therefore we make the argument `d2` to `add_dist()` `const` in both declaration and definition. The second commented statement shows that the compiler will flag as an error any attempt by `add_dist()` to modify any member data of its argument `dist2`.

const Objects

In several example programs, we've seen that we can apply **const** to variables of basic types such as **int** to keep them from being modified. In a similar way, we can apply **const** to objects of classes. When an object is declared as **const**, you can't modify it. It follows that you can use only **const** member functions with it, because they're the only ones that guarantee not to modify it. The CONSTOBJ program shows an example.

```
// constObj.cpp
// constant Distance objects
#include <iostream>
using namespace std;
/////////////////////////////////////////////////////////////////
class Distance //English Distance class
{
private:
    int feet;
    float inches;
public:
    //2-arg constructor
    Distance(int ft, float in) : feet(ft), inches(in)
    { }
    void getdist() //user input; non-const func
    {
        cout << "\nEnter feet: "; cin >> feet;
        cout << "Enter inches: "; cin >> inches;
    }
    void showdist() const //display distance; const func
    { cout << feet << "\'-' " << inches << "\'"; }
};
/////////////////////////////////////////////////////////////////
int main()
{
    const Distance football(300, 0);

    // football.getdist(); //ERROR: getdist() not const
    cout << "football = ";
    football.showdist(); //OK
    cout << endl;
    return 0;
}
```

A football field (for American-style football) is exactly 300 feet long. If we were to use the length of a football field in a program, it would make sense to make it **const**, because changing it would represent the end of the world for football fans. The CONSTOBJ program makes football a **const** variable. Now only **const** functions, such as **showdist()**, can be called for this object.

Non-const functions, such as **getdist()**, which gives the object a new value obtained from the user, are illegal. In this way the compiler enforces the **const** value of football.

When you're designing classes it's a good idea to make **const** any function that does not modify any of the data in its object. This allows the user of the class to create **const** objects. These objects can use any **const** function, but cannot use any non-const function. Remember, using **const** helps the compiler to help you.