Member Functions

Public Member Functions

Public member functions are the gateway into the class itself. These are member functions that are listed under the **public** section of a class and are accessible from any outside function of the program so long as it is through an object.

For example, suppose there is a Fruits class that counts the number of fruits sold and/or returned in a transaction and displays the results to the cashier and customer (See **Fruit.cpp** after this section for full implementation).

```
class Fruits
{
    Public:
        Fruits() : apple(0), orange(0), banana(0), strawberry(0) {}
        void fruitCount(); //Public Member Function
        void displayTotals(); //Public Member Function
    Private:
        int apple;
        int orange;
        int banana;
        int strawberry;
};
```

Suppose that an object of the Fruits class is created in main as such:

```
int main()
{
    Fruits Thomas;
}
```

From here, any public member function can be invoked with the **Thomas** object anywhere this object is in the program. In order to do this, you utilize the **dot operator**. Here is how to invoke both of the functions using the object **Thomas**:

```
int main()
{
    Fruits Thomas;
    Thomas.fruitCount(); //Executes fruitCount()
    Thomas.displayTotals(); //Executes displayTotals()
}
```

See the **Member Function Details** section for an understanding on how to define a member function, how you can access member variables, and how to operate on objects of the class using member functions.

Private Member Functions

Member functions that are **private** are known as **helper functions** and can only be used within other member functions of the class. That means there cannot be a direct function call from outside of the class via an object just as you cannot access private member variables directly from outside of the class.

Suppose a function is created to display a message to the cashier to input an integer value that corresponds to one of the fruits and to place the casher's input into a variable of type **int**. This function serves no purpose to be accessed outside of the Fruits class as this function alters a variable local to a public member function. Therefore, this function is a **helper function**. It aids in the readability of the code, and since the message is generic, this function could be used elsewhere in the class if the class ever grows.

Here is what the **private** portion of the class looks like with the addition of the helper function:

```
private:
    int apple;
    int orange;
    int banana;
    int strawberry;
    //Helper Function
    void message(int &choice);
```

For example, the **Thomas** object invokes the **fruitCount()** function in main as shown on the previous page. Once inside the **fruitCount()** function, the **message(int &choice)** function is able to be invoked just like any regular function you have seen. There is no dot operator required as you have already invoked a member function via an object and are conveniently inside the class.

If you try, for example, in main to do **Thomas.message(choice)** where choice is a variable of type **int**, the compiler will error and tell you it is a private member of the Fruits class.

Member Function Details

When defining (implementing) member functions (public and private), the general formula is:

```
ReturnType ClassName::MemberFunctionName(Formal Parameters)
{ Body of Function }
```

Compared to a function that does not belong to a class, everything is the same except there is the addition of the **ClassName** (type qualifier) and **scope resolution operator**. This is present to denote

that the function being defined is indeed a part of the class at hand and is not some global function or a function from another class.

For example, consider the two public member functions and one helper function of the Fruits class. Here is how to define each function in the implementation section (omission of function body for brevity):

```
void Fruits::fruitCount()
{ Function Body }
void Fruits::displayTotals()
{ Function Body}
void Fruits::message(int &choice)
{ Function Body }
```

No matter if the member function is public or private it gets implemented the same way in the same section.

The calling object of the function will have all of its private data members (variables and functions) available without the dot operator. For example, since the **Thomas** object has invoked the **displayTotals()** function in main, you can use the private data just as if you declared/defined an **int** or **char** in a function. You will see all the member variables used in this manner below.

```
void Fruits::displayTotals()
{ //Displaying the quantity of each of the fruits via member variables
    cout << "Here are the total number of fruits customer bought. \n"
         << "Negative number means customer returned the fruit. \n";
    cout << "\tApples:       " << apple << endl
         << "\tOranges:      " << orange << endl
         << "\tBananas:      " << banana << endl
         << "\tStrawberries: " << strawberry << "\n\n";
}
```

MFD: Multiple Objects In One Member Function

There exists member functions that take objects of the same class as arguments. In this case, you need to use the **dot operator** to distinguish between the calling object's private data and the argument object's private data. It still holds true that the calling object need not use the dot operator, but the argument object(s) do.

Suppose another object of the Fruits class is created named **Emily**. Since there are now two customers buying fruits, a public member function to compare the number of apples between the two is created. The public member function declaration looks as such:

```
void compareApples(const Fruits &Emily);
```

The Fruits object will be called by constant reference to use less memory resources, prevent having to use a copy constructor, and because the object is not being modified.

In main, let's say there is a function invocation that looks like: **Thomas.compareApples(Emily);** Therefore, the calling object is **Thomas** and the argument object is **Emily**. This means that **Thomas** may use the private member data as seen in the private section but **Emily** needs to use the dot operator for her private member data. Here is the implementation below:

```
//This function displays how many apples Emily bought vs Thomas
void Fruits::compareApples(const Fruits &Emily)
{
    cout << "Emily purchased " << Emily.apple << " apple(s) \n";
    cout << "Thomas purchased "  << apple << " apple(s) \n";
    if((Emily.apple + apple) == 0) {
        cout << "No apples sold. Cannot do comparison. \n\n";
        return;
    }
    cout << "\tTherefore, Emily purchased "
         << static_cast<double>(Emily.apple)/(Emily.apple + apple)
            * 100 << '%'
         << " of the apples! \n\n";
}
```

MFD: Constant Member Functions/Formal Parameters

As you've seen above, it is useful to have an object passed by **const** reference. Member functions in a class are also able to be marked as **const** meaning that nothing that goes on within said member function may alter the contents of the object at hand.

It is very important to mark every object and function that does not alter the object's contents as **const** due to the fact that constant objects can only invoke constant member functions and constant member functions can only invoke other constant member functions. This can be thought of as a safety net to keep the integrity of the data.

Keep in mind that non-constant member functions may invoke constant member functions, but not the other way around.

The keyword **const** goes at the end of the function declaration. The function header during the function definition must follow suit. For example, the **void compareApples(const Fruits &Emily)** function is more properly labeled as: **void compareApples(const Fruits &Emily) const** since all the function does is display to the user information. In **Fruit.cpp**, all functions/objects that do not modify the private member variables are labeled with **const**. See that program for examples.