# Constructors

Please keep in mind that constructors have **no** return type, not even void. Also, if you do not declare/define a constructor at all, an automatic constructor will be created by the compiler. However, this is not to be considered reliable when dealing with pointers and dynamically allocated data.

```
class Produce
{
    Public:
        //Constructors Have NO return type/Have same name as class
        Produce(); //Default Constructor
        Produce(string name, int plu, double price); //Constructor
        Produce(const Produce &Object); //Copy Constructor
    Private:
        //Member variables
        string name;
        int plu;
        double price;
};
```

The constructor name has to be the **same** name as the **class**. There are three ways to go about the implementation of a constructor. One way is to initialize the member variables **within curly brackets**, another method is to initialize them using an **initializer list**, and the last method is to use an initializer list/curly brackets hybrid.

Please note that the initializer list works for all types of constructors. However, if you need to copy over/initialize arrays and dynamically allocated data, you must do so within curly brackets. This is where the hybrid could be utilized to copy over arrays and dynamically allocated data within curly brackets and the rest of the data can be copied using the initializer list. See the **MultiArray** program at the end of the **Class** section to see an example of this.

Sometimes the constructors are initialized on the same line as the declaration of the constructor (inside the class interface) or are initialized alongside the other member functions in the implementation section (see **Member Functions** for more information on implementation). **Choose one area to initialize data.** If the initialization process spans more than one line, use the latter implementation. The former implementation will only have examples pertaining to the default constructor in this section.

# Default Constructor

It is important to first create a **default constructor**. This initializes all of the member variables to some default value.

The **first example** you will see is placed in the class interface and the **second example** you will see is placed alongside other member functions in the implementation section.

```
Public: //First example (public portion of class interface)
    Produce() : name(“”), plu(0), price(0.0) {}
```

---

```
Produce::Produce() : name(“”), plu(0), price(0.0) {} //Second example
```

**Initializer list route** (above): A colon is placed after the constructor function and each variable is initialized with a default value in parentheses. It is important to note that each variable is separated by a comma and the **variables must be in the same order as defined in the private section**. Additionally, there must be curly brackets afterward which signify an empty function body.

```
public:  //First example (public portion of class interface)
    Produce()  //Not the recommended as it spans more than one line
    {          //but still legal to do
        name = “”;
        plu = 0;
        price = 0.0;
    }
```

---

```
Produce::Produce() //Second example
{
    name = “”;
    plu = 0;
    price = 0.0;
}
```

**Normal function definition route** (above): The same syntax as any function you have come across. The member variables are within curly brackets and are set using the assignment operator (=).

To invoke the default constructor, declare the class object as such:

```
Produce Carrot; //Example declaration one may see in main()
```

This will set the member variables to the default values. Notice that there are no parentheses appended onto the object **Carrot**. Those are

not required to invoke the default constructor and will not compile if you try to use them. The compiler will think you are trying to invoke a regular function if you include them.

## Constructor With Formal Parameters

The second constructor is a **constructor with formal parameters**. This works just like a normal function invocation, but it is done at the same time as an object declaration. For example:

```
Produce RedPepper("Red Pepper", 4088, 1);
```

This time, the parentheses are required with the appropriate arguments. Just like function arguments you have seen, variables are allowed to be used an as argument.

With a constructor like this, there are a couple of important notes to make. If you make the formal parameter variables a different name as the member variables, there is nothing additional to worry about. However, if the names are the same (as they are in this example), caution has to be made when copying data over inside the curly bracket method. The initializer list method works fine as is.

```
Produce::Produce(string name, int plu, double price) : name(name),
                  //Method works just fine               plu(plu),
                                                         price(price) {}
```

---

```
Produce::Produce(string name, int plu, double price) //Logic error
{
    name = name; //Self assignment
    plu = plu; //Self assignment
    price = price; //Self assignment
}
```

What this is actually doing is assigning the member variable to itself and ignoring the variables of the formal parameters. The way to circumvent this is to use the **this** operator. The **this** operator is to denote the member functions and member variables of the object at hand (calling object of the member functions or constructor). Keep in mind that **this** is a pointer that must be dereferenced. Therefore, utilization of the **arrow operator** is used as such:

```
Produce::Produce(string name, int plu, double price) //No logic error
{
    this->name = name; //No ambiguity on any assignment
    this->plu = plu;
    this->price = price;
}
```

# Copy Constructor

The third constructor is the **copy constructor**. This constructor copies over all contents from one object to another. To invoke the copy constructor, you must have an object of the class already declared and initialized.

For example, let's use the defined **RedPepper** object from the previous page. In order to copy over its contents to another object, say **RedPepperOhio**, you would do as such (at the same time as declaring the object):

```
Produce RedPepperOhio(RedPepper);
```

Now **RedPepper** and **RedPepperOhio** are completely different objects but contain the exact same data. Maybe this was done to increase sales as the name **RedPepperOhio** indicates local produce. Here is the definition of the copy constructor using both methods:

```
Produce::Produce(const Produce &Obj) : name(Obj.name),
                                       plu(Obj.plu),
                                       price(Obj.price) {}
```

---

```
Produce::Produce(const Produce &Obj)
{
    name = Obj.name;
    plu = Obj.plu;
    price = Obj.price;
}
```

In order to access the unique data of the object named Obj, you must use the dot operator.

The dot operator is not needed for any calling object's member variables/functions. In this case, the calling object is the object being initialized (**RedPepperOhio**).

Take note that the formal parameter is a constant reference. It cannot be a call-by-value formal parameter, because this **is** the function to copy over objects and would result in an infinite recursive call to said function..

# Special Remarks

**If a constructor with formal parameters is created and an object using the default constructor is declared, this is illegal (need both).**

**You may reset values of an object using a formal parameter constructor such as: RedPepper = Produce("Red Pepper", 4677, 1.5)**