Member Functions

- *Member functions* are functions that are included within a class.
- Usually the data within a class is private and the functions are public
- Data is hidden so it will be safe from accidental manipulation, while the functions that operate on the data are public so they can be accessed from outside the class.

Methods/functions

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

class MyClass {        // The class
  public:              // Access specifier
    void myMethod() {   // Method/function
      cout << "Hello World!“<<endl;
    }
};

int main() {
  MyClass myObj;     // Create an object of MyClass
  myObj.myMethod();  // Call the method
  return 0;
}
```

Member Functions

- Remember that the definition of the class does not create any objects. It only describes how they will look when they are created,
- just as a structure definition describes how a structure will look but doesn't create any structure variables.
- Defining an object is similar to defining a variable of any data type: Space is set aside for it in memory.

Methods/functions

- Methods are functions that belongs to the class.
- There are two ways to define functions that belongs to a class:
  - Inside class definition
  - Outside class definition

```
#include<iostream>
using namespace std;
class car
{
private:
	int sc;
	float milage;
public:
	void setData(int m, float n)
	{
		sc = m;
		milage = n;
	}
	void showData()
	{
		cout << "Seating Capcity: " << sc << endl <<" Milage: " << milage <<
endl;
	}

};
```

```
int main()
{
	car c1;
	c1.setData(5,8.9);
	c1.showData();
}
```

- See Example on Visual Studio:
  - The setdata() function accepts a value as a parameter and sets the data members to this value.
  - The showdata() function displays the value stored in data members .

Accessors (getters) and **Mutators** (setters)

```
class Person
{
private://access control
  string firstName;
  string lastName;
  int year;
protected:
  int phoneNumber;
  int salary;
};
public:
  string getFname()
  {
if (!firstName.empty())
    return firstName;
Else
        cout<<“name is empty”<<endl;
  }
```

```
String getLname()
  {
    return lastName;
  }
};
```

Accessors (getters) and **Mutators** (setters)

Mutators (setters) are used to set values of private data members.

```
class Person
{
private://access control
  string firstName;
  string lastName;
  int year;
protected:
  int phoneNumber;
  int salary;
};
public:
Void setFandLname()
{
        cin>>firstName>>lastName;
}
```

Utilities

- utility function (also called a helper function). A utility function is not part of a class's public interface; rather, it is a private member function that supports the operation of the class's public member functions. Utility functions are not intended to be used by clients of a class

Constructors

- A constructor is a member function that is executed automatically whenever an object is created.
- However, it's convenient if an object can initialize itself when it's first created, without requiring a separate call to a member function.
- To create a constructor, use the same name as the class, followed by parentheses ()

```
class MyClass {     // The class
  public:           // Access specifier
    MyClass() {     // Constructor
      cout << "Hello World!";
    }
};

int main() {
  MyClass myObj;    // Create an object of MyClass (this will call the constructor)
  return 0;
}
```

The constructor has the same name as the class, it is always public, and it does not have any return value.

Why we need constructors

- We write set count() function to do initialize the data members by creating objects and call it with an argument of 0, or we could provide a some function, which would always set count to 0.
- However, such functions would need to be executed every time we created a Counter obj

  Counter c1; //every time we do this,

  c1.zer0_count(); //we must do this too
- This is mistake prone, because the programmer may forget to initialize the object after creating it.

Two methods to create constructors

```
count()
{ count = 0; }
```

```
count() :count(0)
    {    }
```

- If there are two objects of the class then constructor will be called twice, one for first object and second time for 2nd object.

Types of Constructors

- Default constructor
- Parametrized constructor
- Copy constructor

Default Constructor

- Default constructor is the constructor which doesn't take any argument. It has no parameters.
- Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.

```
using namespace std;
class construct {
public:
    int a, b;
// Default Constructor
    construct()
    {
        a = 10;
        b = 20;
    }
};
int main()
{
    // Default constructor called automatically
    // when the object is created
    construct c;
    cout << "a: " << c.a << endl << "b: " << c.b;
    return 0;
}
```

Parametrized Constructor

- It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

Parametrized Constructor

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

class Point {
private:
    int x, y;

public:
Point(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    int getX()
    {
        return x;
    }
    int getY()
    {
        return y;
    }
};
```

```
int main()
{
    // Constructor called
    Point p1(10, 15);

    // Access values assigned by constructor
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

    return 0;
}
```

Output:

```
p1.x = 10, p1.y = 15
```

```
using namespace std;

class construct
{

public:
    float area;

    // Constructor with no parameters
    construct()
    {
        area = 0;
    }

    // Constructor with two parameters
    construct(int a, int b)
    {
        area = a * b;
    }

    void disp()
    {
        cout<< area<< endl;
    }
};

int main()
{
    // Constructor Overloading
    // with two different constructors
    // of class name
    construct obj;
    construct obj2( 10, 20);

    obj.disp();
    obj2.disp();
    return 0;
}
```

```
0
200
```

Uses of PC

- It is used to initialize the various data elements of different objects with different values when they are created.
- **What is Constructor Overloading?**
  - have more than one constructors in a class
  - Constructors can be overloaded in a similar way as function overloading. Overloaded constructors have the same name (name of the class) but the different number of arguments. Depending upon the number and type of arguments passed, the corresponding constructor is called.
  - E.g you have both default and parametrized constructors in a class, it means you have more than one constructor and one has no parameter and other have parameters.

```
using namespace std;

class Person {
   private:
    int age;
   public:
    // 1. Constructor with no arguments
    Person() {
        age = 20;
    }
    // 2. Constructor with an argument
    Person(int a) {
        age = a;
    }
    int getAge() {
        return age;
    }
};
int main() {
    Person person1, person2(45);
    cout << "Person1 Age = " << person1.getAge() << endl;
    cout << "Person2 Age = " << person2.getAge() << endl;
    return 0;
}
```

Constructors with Default Arguments in C++

- Default arguments of the constructor are those which are provided in the constructor declaration. If the values are not provided when calling the constructor the constructor uses the default arguments automatically.

```
#include<iostream>
using namespace std;
class Simple{
    int data1;
    int data2;
    int data3;

    public:
        Simple(int a=1, int b=9, int c=8){
            data1 = a;
            data2 = b;
            data3 = c;
        }
        void printData()
       {cout<<"The value of data1, data2 and data3 is "<<data1<<",
       "<< data2<<" and "<< data3<<endl;}
};
```

```
int main(){
    Simple s(12, 13);
    s.printData();
    return 0;
}
```

C++ Function Overloading

In C++, two functions can have the same name if the number and/or type of arguments passed is different.

These functions having the same name but different arguments are known as overloaded functions. For example:

```
// same name different arguments
int test() { }
int test(int a) { }
float test(double a) { }
int test(int a, double b) { }
```

Notice that the return types of all these 4 functions are not the same. Overloaded functions may or may not have different return types but they must have different arguments. For example,

```
// Error code
int test(int a) { }
double test(int b){ }
```

Here, both functions have the same name, the same type, and the same number of arguments. Hence, the compiler will throw an error.

Diff types of parameter

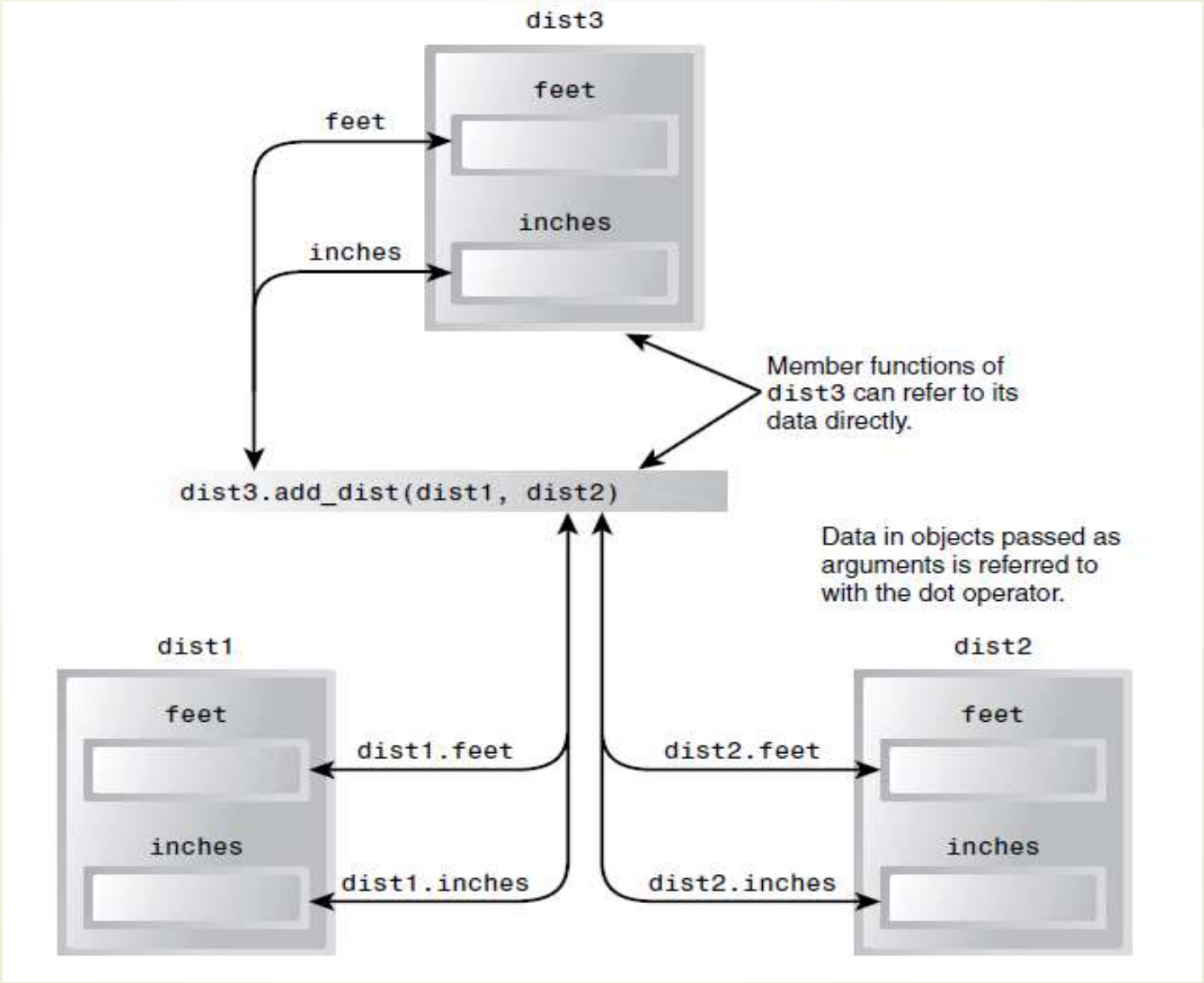
```
// function with float type parameter
float absolute(float var){
    if (var < 0.0)
        var = -var;
    return var;
}

// function with int type parameter
int absolute(int var) {
     if (var < 0)
         var = -var;
    return var;
}

int main() {

    // call function with int type parameter
    cout << "Absolute value of -5 = " << absolute(-5) << endl;

    // call function with float type parameter
    cout << "Absolute value of 5.5 = " << absolute(5.5f) <<
endl;
    return 0;
}
```

Diff number of parameters

Code in notes

Objects as function arguments

- Pass objects as parameters in a function



Returning objects from the function