```
}

int main() {
    Circle circle;
    Square square;

    displayShape(&circle);  // Drawing Circle
    displayShape(&square);  // Drawing Square

    return 0;
}
```

Here, the draw() function call is dynamically bound to the correct derived class implementation at runtime, demonstrating polymorphic behavior.

3.2 Virtual Functions

In C++, a virtual function is a member function of a class that is declared with the virtual keyword. Virtual functions enable polymorphic behavior, allowing derived classes to provide their own implementation of the function by method overriding.

Declaration and Syntax:

- The virtual keyword is used to declare a function as virtual in the base class.

  ```
  class Base {
  public:
      virtual void myVirtualFunction() {
          // Base class implementation
      }
  };
  ```

- In the derived class, the override keyword is used to explicitly indicate that the function is overriding a virtual function from the base class.

  ```
  class Derived : public Base {
  public:
      void myVirtualFunction() override {
          // Derived class implementation
      }
  };
  ```

Purpose and Benefits:

1. Polymorphism: Virtual functions enable polymorphism, allowing different classes to provide different implementations of the same function.
2. Dynamic Binding: Virtual functions are resolved at runtime based on the actual type of the object, enabling dynamic method dispatch.
3. Base Class Pointers: Virtual functions are often used with base class pointers to achieve runtime polymorphism.

Example:

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

// Base class
class Animal {
public:
    // Virtual function
    virtual void makeSound() {
        cout << "Animal makes a sound" << endl;
    }
};

// Derived class
class Dog : public Animal {
public:
    // Override virtual function
    void makeSound() override {
        cout << "Dog barks" << endl;
    }
};

// Derived class
class Cat : public Animal {
public:
    // Override virtual function
    void makeSound() override {
        cout << "Cat meows" << endl;
    }
};

int main() {
    Animal* animal1 = new Dog(); // Creating a Dog object
    Animal* animal2 = new Cat(); // Creating a Cat object

    // Calling virtual function on Dog object
    animal1->makeSound(); // Output: Dog barks

    // Calling virtual function on Cat object
    animal2->makeSound(); // Output: Cat meows

    return 0;
}
```

In this example:

- We have a base class Animal with a virtual function makeSound().
- There are two derived classes Dog and Cat, each overriding the makeSound() function with their specific implementation.
- In the main() function, we create objects of type Dog and Cat using base class pointers.
- When we call the makeSound() function on these objects, the appropriate version of the function is invoked based on the actual type of the object, demonstrating polymorphic behavior through dynamic dispatch.

3.3 Pure Virtual Functions

In C++, a pure virtual function (or abstract function) is a virtual function declared in a base class that has no implementation. It is declared by assigning 0 in the base class, must be overridden in derived classes. It serves as a placeholder for derived classes to override and provide their own implementation. A class containing at least one pure virtual function is known as abstract class and cannot be instantiated directly.

Declaration and Syntax:

- Pure virtual functions are declared with = 0 at the end of their declaration.

```
class AbstractBase {
public:
    virtual void pureVirtualFunction() = 0; // Pure virtual function
};
```

Example:

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

// Abstract base class
class Animal {
public:
    // Pure virtual function
    virtual void makeSound() = 0;
};

// Derived class
class Dog : public Animal {
public:
    // Override pure virtual function
    void makeSound() override {
        cout << "Dog barks" << endl;
    }
};

// Derived class
class Cat : public Animal {
```

```
public:
    // Override pure virtual function
    void makeSound() override {
        cout << "Cat meows" << endl;
    }
};

int main() {
    Animal* animal1 = new Dog(); // Creating a Dog object
    Animal* animal2 = new Cat(); // Creating a Cat object

    // Calling pure virtual function on Dog object
    animal1->makeSound(); // Output: Dog barks

    // Calling pure virtual function on Cat object
    animal2->makeSound(); // Output: Cat meows

    return 0;
}
```

In this example:

- We have an abstract base class Animal with a pure virtual function makeSound().
- There are two derived classes Dog and Cat, each overriding the makeSound() function with their specific implementation.
- In the main() function, we create objects of type Dog and Cat using base class pointers.
- When we call the makeSound() function on these objects, the appropriate version of the function is invoked based on the actual type of the object, demonstrating polymorphic behavior through dynamic dispatch.

Abstract Classes and Interfaces:

- **Abstract Class:** A class that contains at least one pure virtual function. It cannot be instantiated and serves as a blueprint for derived classes to implement common behavior while allowing specific implementations for their own unique features.
- **Interface:** C++ does not have a built-in concept of interfaces like Java or C#. However, interfaces can be simulated using abstract classes. An interface in C++ is an abstract class that has only pure virtual functions and no data members or non-virtual member functions. This ensures that the derived classes implement the specific methods defined by the interface.

Differences between Abstract Classes and Interfaces:

Feature	Abstract Classes	Interfaces
Definition	A class containing at least one pure virtual function	An abstract class containing only pure virtual functions