Astroinformatics II (Semester 2 2024)

Object Oriented Programming

Nina Hernitschek
Centro de Astronomía CITEVA
Universidad de Antofagasta

November 12, 2024

Motivation

We have seen how to write programs in C++.

So far, we have used **functional programming** (which involves functions).

In this lecture, we will extend this to the concept of **object-oriented programming**.

Object Oriented Programming

Object-oriented programming (OOP) expands the concept of a data structure to a **class** which can hold both data and functions.
Classes are what separate the capabilities of C from that of C++. Classes are the object-oriented aspect of programming.

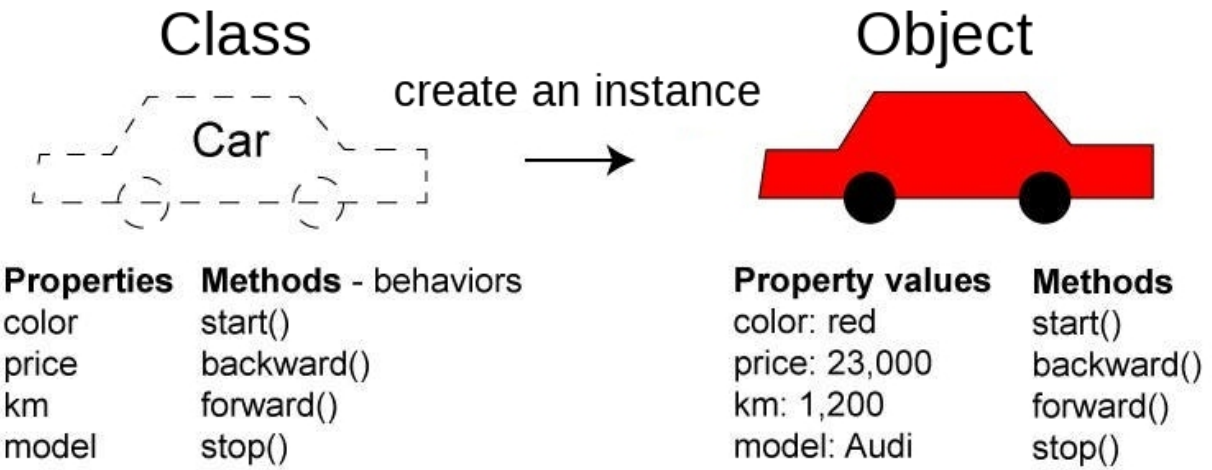
An **object** is an instantiation of a class, so a class would be the type, and an object would be the variable.

Object Oriented Programming

Object-oriented programming (OOP) expands the concept of a data structure to a **class** which can hold both data and functions.
Classes are what separate the capabilities of C from that of C++. Classes are the object-oriented aspect of programming.

An **object** is an instantiation of a class, so a class would be the type, and an object would be the variable.

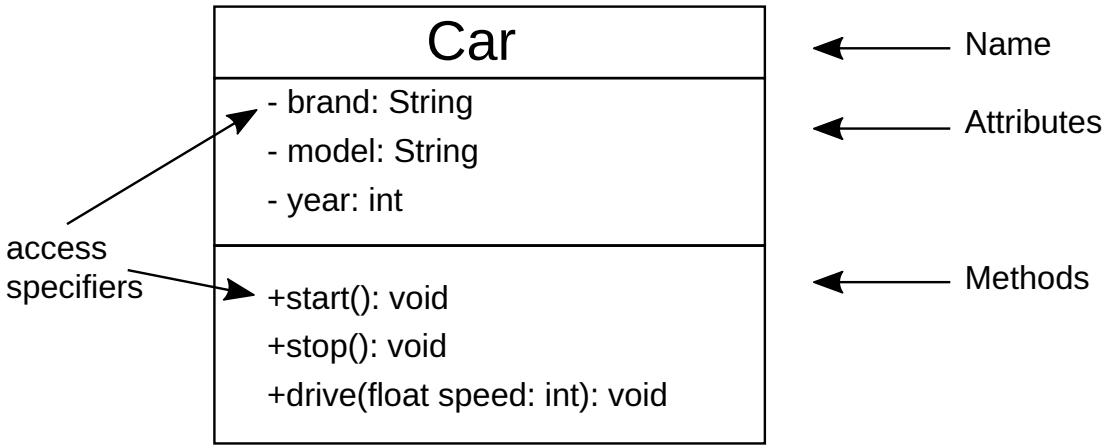
This is motivated by real-world objects which typically have both properties and functionality:



Object Oriented Programming

Along with descriptions of concepts, we will use the **UML (Unified Modeling Language)** description that is widely used in OOP.

UML is a standardized modeling language that helps in designing and documenting software systems.



The Basic Ideas of OOP

Classic **procedural** programming languages before C++ (such as C) often focused on the question "What should the program do next?"

The way you structure a program in these languages is:

1. Split it up into a set of tasks and subtasks.
2. Make functions for the tasks.
3. Instruct the computer to perform them in sequence.

The Basic Ideas of OOP

Classic **procedural** programming languages before C++ (such as C) often focused on the question "What should the program do next?"

The way you structure a program in these languages is:

1. Split it up into a set of tasks and subtasks.
2. Make functions for the tasks.
3. Instruct the computer to perform them in sequence.

With large amounts of data and/or large numbers of tasks, this leads to overly complex and **unmaintainable programs**.

The Basic Ideas of OOP

Classic **procedural** programming languages before C++ (such as C) often focused on the question "What should the program do next?"

The way you structure a program in these languages is:

1. Split it up into a set of tasks and subtasks.
2. Make functions for the tasks.
3. Instruct the computer to perform them in sequence.

With large amounts of data and/or large numbers of tasks, this leads to overly complex and **unmaintainable programs**.

To manage this complexity, it is easier to develop self-sufficient, modular pieces of code.
Similarly to people thinking of the world in terms of interacting objects, OOP allows programmers to pack away details into neat, self-contained boxes (objects) so that they can think of the objects more **abstractly** and focus on the **interactions** between them.

The Basic Ideas of OOP

Three primary features of OOP are:

Encapsulation: grouping related data and functions together as classes and defining an interface to those classes.

The Basic Ideas of OOP

Three primary features of OOP are:

Encapsulation: grouping related data and functions together as classes and defining an interface to those classes.

Inheritance: allowing code to be reused between related classes.

The Basic Ideas of OOP

Three primary features of OOP are:

Encapsulation: grouping related data and functions together as classes and defining an interface to those classes.

Inheritance: allowing code to be reused between related classes.

Polymorphism: allowing an object to be one of several classes, and determining at runtime which functions to call on it based on its type.

The Basic Ideas of OOP

Three primary features of OOP are:

Encapsulation: grouping related data and functions together as classes and defining an interface to those classes.

Inheritance: allowing code to be reused between related classes.

Polymorphism: allowing an object to be one of several classes, and determining at runtime which functions to call on it based on its type.

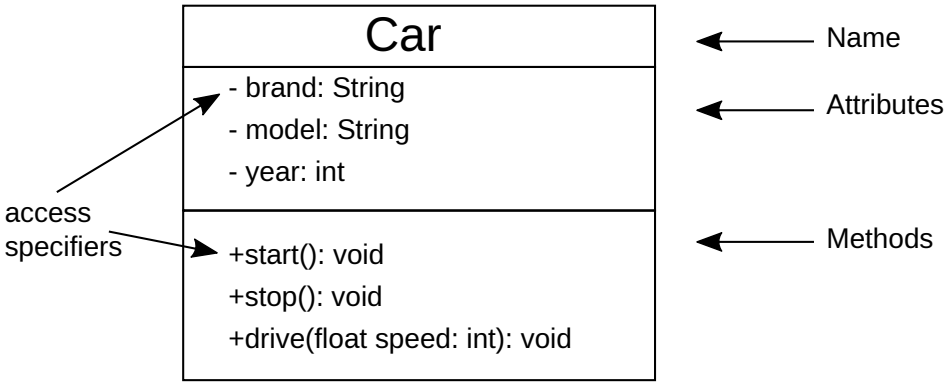
Associations: between objects.

Encapsulation

Encapsulation package up data (methods) and related functionality (methods) into classes.

When working with a class (e.g. from a library), to use it we do not need to know how it works internally. All we need to know is its **interface**: methods and how they can be accessed.

In UML:



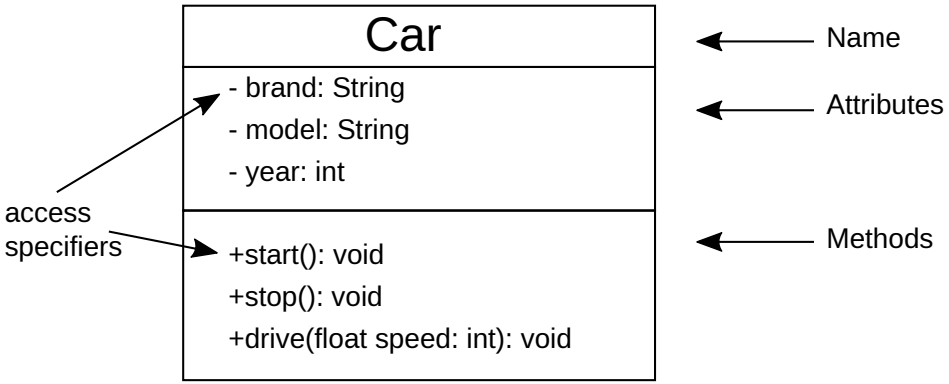
Class Name: The name of the class is typically written in the top compartment of the class box and is centered and bold.

Encapsulation

Encapsulation package up data (methods) and related functionality (methods) into classes.

When working with a class (e.g. from a library), to use it we do not need to know how it works internally. All we need to know is its **interface**: methods and how they can be accessed.

In UML:



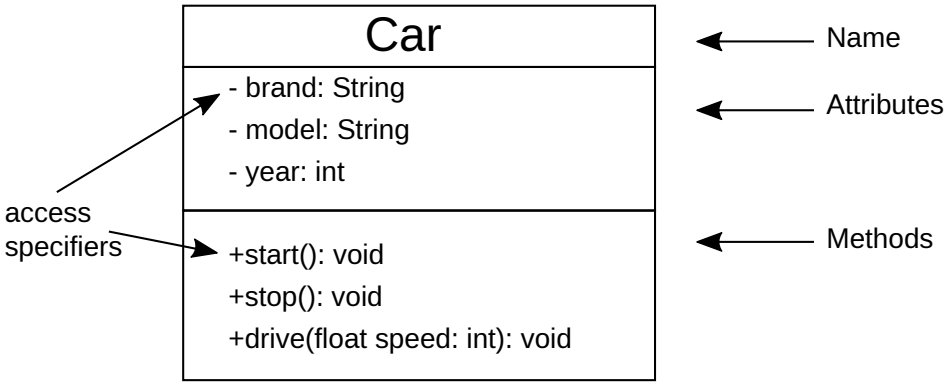
Attributes: Attributes, also known as properties or fields, represent the data members of the class. They are listed in the second compartment of the class box and often include the visibility (e.g., public, private) and the data type of each attribute.

Encapsulation

Encapsulation package up data (methods) and related functionality (methods) into classes.

When working with a class (e.g. from a library), to use it we do not need to know how it works internally. All we need to know is its **interface**: methods and how they can be accessed.

In UML:



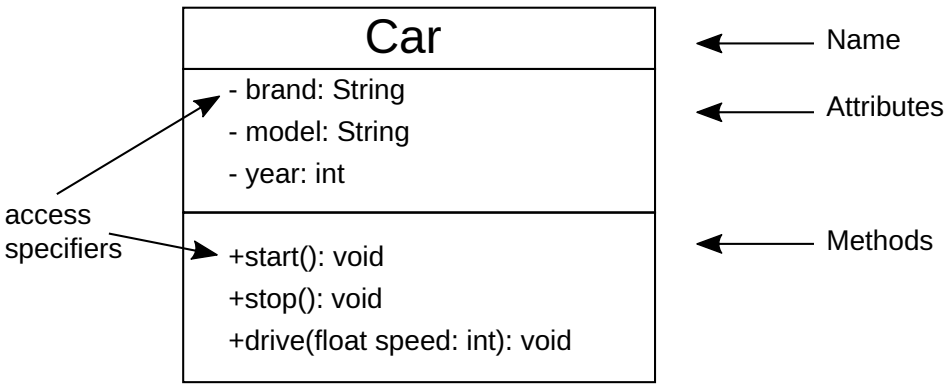
Methods: Methods represent the behavior or functionality of the class. They are listed in the third compartment of the class box and include the visibility (e.g., public, private), return type, and parameters of each method.

Encapsulation

Encapsulation package up data (methods) and related functionality (methods) into classes.

When working with a class (e.g. from a library), to use it we do not need to know how it works internally. All we need to know is its **interface**: methods and how they can be accessed.

In UML:



Access Specifiers: They indicate the access level of attributes and methods. Common access specifiers are:

+ for public (visible to all classes)
- for private (visible only within the class)
for protected (visible to subclasses)
˜for package visibility (visible to classes in the same package)

Encapsulation

This is often compared to operating a car: when you drive, you don't care how the steering wheel makes the wheels turn; you just care that the **interface** the car presents (the steering wheel) allows you to accomplish your goal.

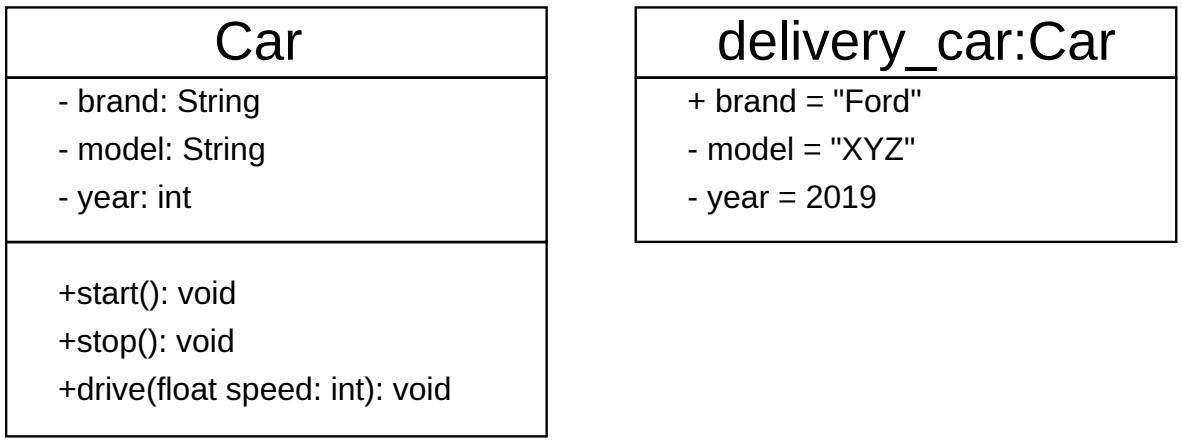
This is why C++ makes you specify public and private access specifiers: by default, it assumes that the things you define in a class are internal details which someone using your code should not have to worry about. The practice of hiding away these details from client code is often described as "black box".

Encapsulation

Class diagrams are primarily used for modeling the static structure of a software system. They depict the classes, their attributes, methods, and the relationships between classes.

Object diagrams, on the other hand, focus on capturing a snapshot of the runtime instances of classes and the relationships between them. They represent a set of objects and their associations. In an object diagram, you might depict instances like `myCar` (an instance of the `Car` class).

class diagram vs. object diagram

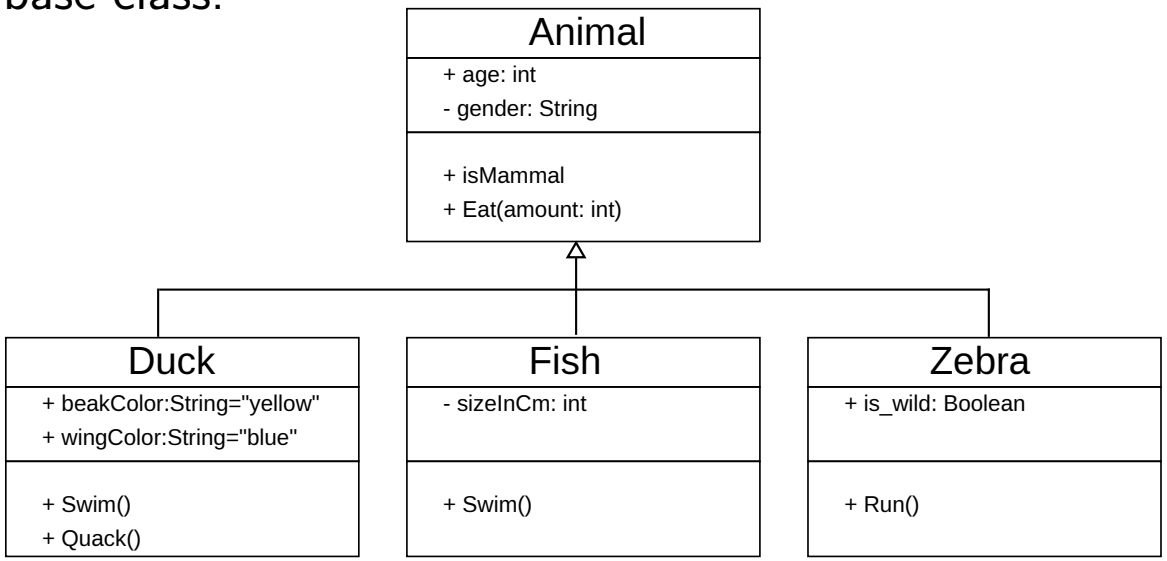


Inheritance

Inheritance allows us to define hierarchies of related classes. Inheritance represents an *is-a* relationship between classes, where the subclass inherits the properties and behaviors the superclass.

In UML, inheritance is depicted by a solid line with a closed, hollow arrowhead pointing from the subclass to the superclass.

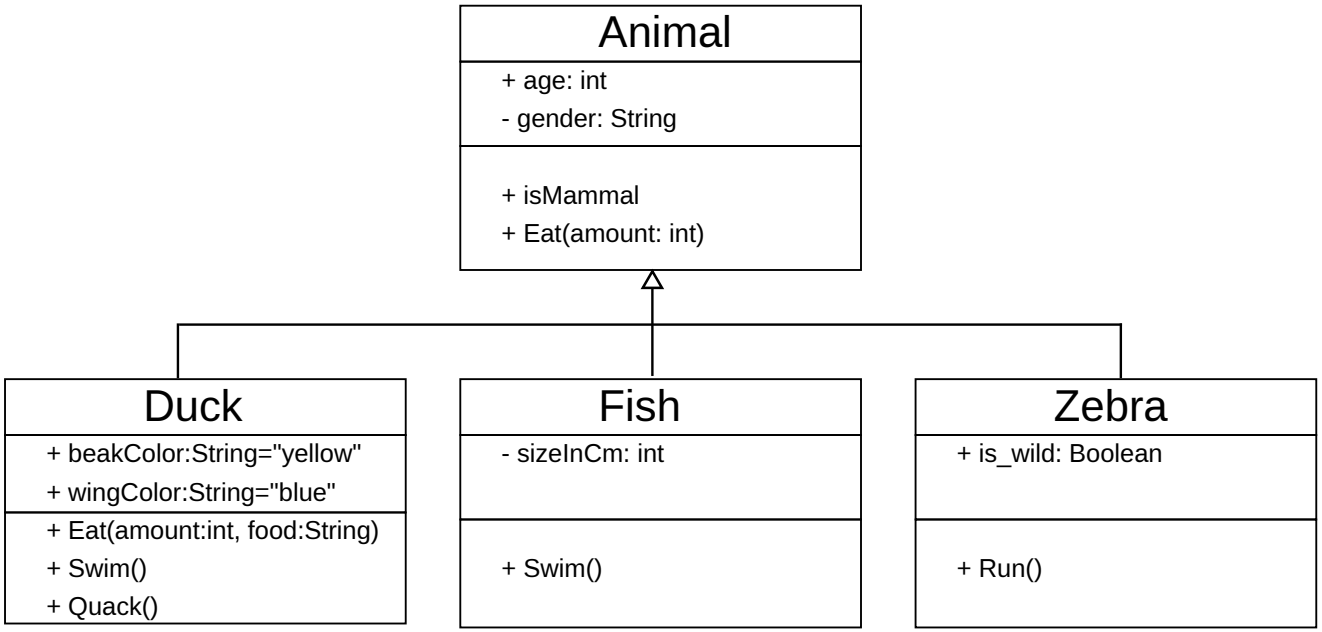
In the following **example**, the class `Animal` serves as the base class generalizing animal species (e.g. in a computer game), which are themselvse represented as classes that inherit and extend the functionality of the base class.



Inheritance

In the example class diagram, we have seen that some methods differ.

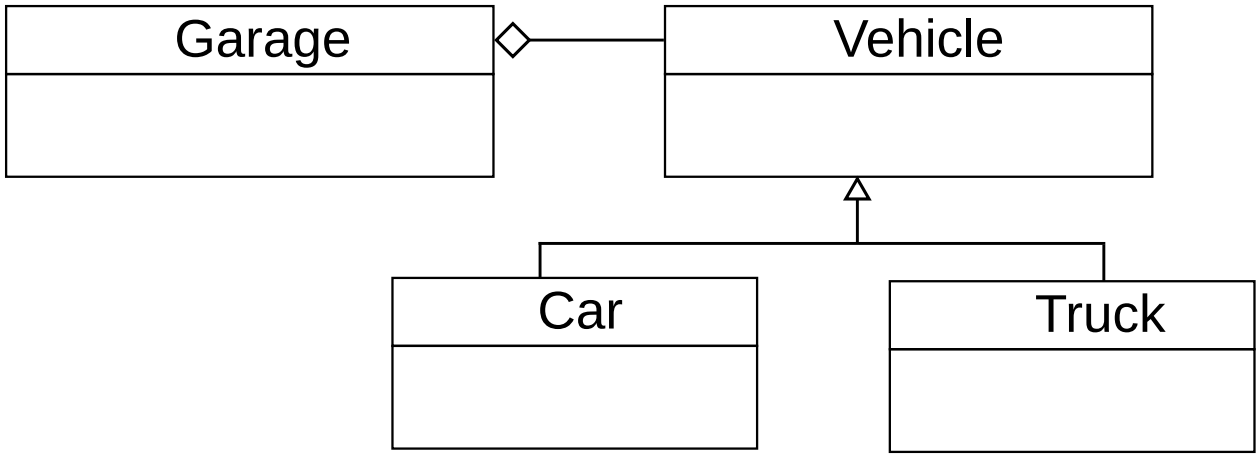
Generally, we might want to generate the methods for subclasses in a different way than for generic superclasses.
To accomplish this, we can simply **redefine (override) methods**. Below, we override the `eat()` method.



Polymorphism

Polymorphism means "many shapes". It refers to the ability of one object to have many types. If we have a function that expects a `Vehicle` object, we can safely pass it a `Car` object, because every `Car` is also a `Vehicle`. Likewise for references and pointers: anywhere you can use a `Vehicle*`, you can use a `Car*`, like in a parking garage.

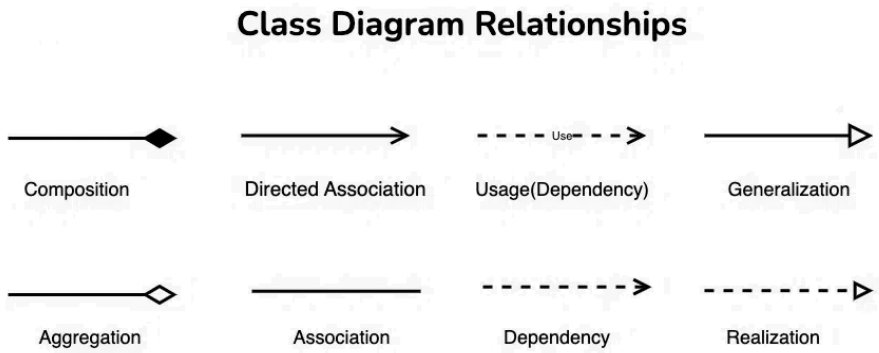
We represent polymorphism in UML in the following way:



Relationships

In class diagrams, **relationships** between classes describe how classes are connected or interact with each other within a system.

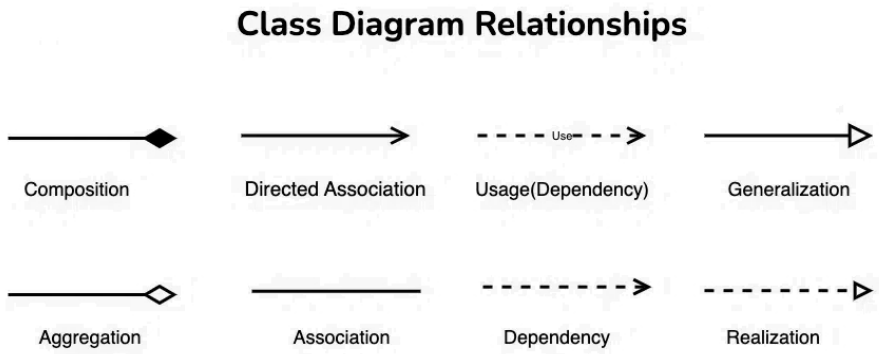
We have already seen one: inheritance. But there are several more types of relationships in object-oriented modeling, each serving a specific purpose. Here are some common types of relationships in class diagrams:



Relationships

In class diagrams, **relationships** between classes describe how classes are connected or interact with each other within a system.

We have already seen one: inheritance. But there are several more types of relationships in object-oriented modeling, each serving a specific purpose. Here are some common types of relationships in class diagrams:

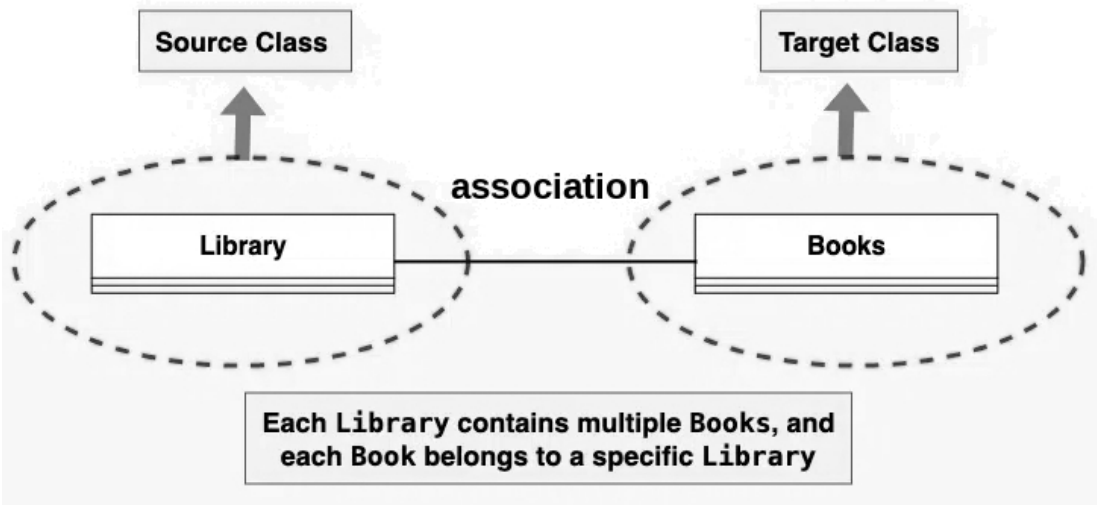


We only look into the most common one: **association**.

Relationships

An **association** represents a bi-directional relationship between two classes. It indicates that instances of one class are connected to instances of another class.
Associations are typically depicted as a solid line connecting the classes, with optional arrows indicating the direction of the relationship.

Let's understand association using an **example**:



The `Library` class can be considered the source class because it contains a reference to multiple instances of the `Book` class. The `Book` class would be considered the target class because it belongs to a specific `Library` object.

OOP in C++

We will now look into those concepts in detail, with application to coding in C++.

Objects and Classes

We saw: An **object** is the basic unit of object oriented programming. A class serves as a blueprint for an **object**, combining data representation and methods for manipulating the data.

In Python, we have already sometimes worked with classes and objects, e.g. when using `ndarray` (a multi-dimensional array of items of the same type and size).

Class Definition

In C++, a class definition starts with the keyword `class` followed by the class name; and the class body, enclosed by a pair of curly brackets.

For **example**, we define the `Box` data type using the keyword `class` as follows:

```
1  class Box {
2     public:
3        double length;   // Length of a box
4        double width;  // Width of a box
5        double height;   // Height of a box
6  };
```

A class definition must be followed either by a semicolon or a list of declarations.

The keyword `public` determines the access attributes of the members of the class that follows it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as private or protected which we will discuss in a sub-section.

C++ Objects

A class provides the blueprints for objects, so basically an object is created from a class.

We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class `Box`:

```
Box Box1;          // Declare Box1 of type Box
Box Box2;          // Declare Box2 of type Box
```

Both of the objects `Box1` and `Box2` will have their own copy of data members.

C++ Objects

A class provides the blueprints for objects, so basically an object is created from a class.

We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class `Box`:

```
Box Box1;          // Declare Box1 of type Box
Box Box2;          // Declare Box2 of type Box
```

Both of the objects `Box1` and `Box2` will have their own copy of data members.

The **public** data members of objects of a class can be **accessed** using the direct member access operator (`.`). Let us try the following example to make the things clear:

C++ Objects


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

class Box {
   public:
      double length;   // Length of a box
      double width;  // Width of a box
      double height;   // Height of a box
};

int main() {
   Box Box1;        // Declare Box1 of type Box
   Box Box2;        // Declare Box2 of type Box
   double volume = 0.0;     // Store the volume of a box here

   // box 1 specification
   Box1.height = 5.0;
   Box1.length = 6.0;
   Box1.width = 7.0;

   // box 2 specification
   Box2.height = 10.0;
   Box2.length = 12.0;
   Box2.width = 13.0;

   // volume of box 1
   volume = Box1.height * Box1.length * Box1.width;
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.height * Box2.length * Box2.width;
   cout << "Volume of Box2 : " << volume <<endl;
   return 0;
}
```

C++ Objects

When the above code is compiled and executed, it produces the following result:

C++ Objects

When the above code is compiled and executed, it produces the following result:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

It is important to note that private and protected members can not be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

C++ Objects

When the above code is compiled and executed, it produces the following result:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

It is important to note that private and protected members can not be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

So far, you have got very basic idea about C++ Classes and Objects. There are further interesting concepts related to C++ Classes and Objects from which we will just discuss some:

class member functions, class access modifiers, constructors and destructors, and static class members.

Class Member Functions

A **member function** of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Let us take the previously defined class and extend it with a member function:

```
1  class Box {
2     public:
3        double length;      // Length of a box
4        double width;     // Width of a box
5        double height;      // Height of a box
6
7        double getVolume(void) {
8           return length * width * height;
9        }
10 };
```

Class Member Functions

Alternatively, you can define a member function outside of the class using the scope resolution operator (::) as follows:

```
double Box::getVolume(void) {
   return length * width * height;
}
```

Here, the only important point is that you would have to use class name just before the :: operator.

Class Member Functions

A member function will be **called** using a dot operator (`.`) on a object where it will manipulate data related to that object only as follows:

```
1  Box myBox;           // Create an object
2
3  myBox.getVolume();   // Call member function for the object
```

Class Member Functions

Let us put together above concepts:

```
1  #include <iostream>
2  using namespace std;
3
4  class Box {
5     public:
6        double length;          // Length of a box
7        double width;         // Width of a box
8        double height;          // Height of a box
9
10       // Member functions declaration
11       double getVolume(void);
12       void setLength(double len);
13       void setWidth(double wit);
14       void setHeight(double hei);
15 };
16
17 // Member functions definitions
18 double Box::getVolume(void) {
19    return length * width * height;
20 }
21
22 void Box::setLength(double len) {
23    length = len;
24 }
25 void Box::setWidth(double wit) {
26    width = wit;
27 }
28 void Box::setHeight(double hei) {
29    height = hei;
30 }
```

Class Member Functions

continuation of source code:

```
31  // Main function for the program
32  int main() {
33     Box Box1;                // Declare Box1 of type Box
34     Box Box2;                // Declare Box2 of type Box
35     double volume = 0.0;     // Store the volume of a box here
36
37     // box 1 specification
38     Box1.setLength(6.0);
39     Box1.setWidth(7.0);
40     Box1.setHeight(5.0);
41
42     // box 2 specification
43     Box2.setLength(12.0);
44     Box2.setWidth(13.0);
45     Box2.setHeight(10.0);
46
47     cout << "Volume of Box1 : " << Box1.getVolume() <<endl;
48     cout << "Volume of Box2 : " << Box2.getVolume() <<endl;
49
50     return 0;
51  }
```

Class Member Functions

continuation of source code:

```
31  // Main function for the program
32  int main() {
33     Box Box1;                // Declare Box1 of type Box
34     Box Box2;                // Declare Box2 of type Box
35     double volume = 0.0;     // Store the volume of a box here
36
37     // box 1 specification
38     Box1.setLength(6.0);
39     Box1.setWidth(7.0);
40     Box1.setHeight(5.0);
41
42     // box 2 specification
43     Box2.setLength(12.0);
44     Box2.setWidth(13.0);
45     Box2.setHeight(10.0);
46
47     cout << "Volume of Box1 : " << Box1.getVolume() <<endl;
48     cout << "Volume of Box2 : " << Box2.getVolume() <<endl;
49
50     return 0;
51  }
```

Compiling and executing the above code produces the following result:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

Class Access Modifiers

Access modifiers are used for data hiding implementation, which is an important feature of object-oriented programming, allowing the functions of a program to access directly the internal representation of a class type.

In C++, a class member can be defined as `public`, `private` or `protected`. By default, C++ assumes members to be private.

A class can have multiple `public`, `protected`, or `private` labeled sections. Each section remains in effect until either another section label or the closing right brace of the class body is seen. The default access for members and classes is `private`.
See the following **example**:

```
1  class Base {
2     public:
3        // public members go here
4     protected:
5        // protected members go here
6     private:
7        // private members go here
8  };
```

Class Access Modifiers

Public members are accessible from anywhere outside the class but within a program. You can set and get the value of public variables without any member function.

The following **example** demonstrates the use of `public` access modifier:

Class Access Modifiers


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

class Line {
   public:
      double length;
      void setLength(double len);
      double getLength(void);
};

// Member functions definitions
double Line::getLength(void) {
   return length;
}

void Line::setLength(double len) {
   length = len;
}

// Main function for the program
int main() {
   Line line;

   // set line length
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   // set line length without member function
   line.length = 10.0; // OK: because length is public
   cout << "Length of line : " << line.length <<endl;

   return 0;
}
```

Class Access Modifiers

Private members cannot be accessed, or even viewed from outside the class. Only the class (and `friend` functions) can access private members.

By default, all class members are private.

The following **example** demonstrates the use of the `private` access modifier:

```
1 class Box {
2    double width; //this is a private member
3
4    public:
5       double length;
6       void setWidth(double wid);
7       double getWidth(void);
8 };
```

Class Access Modifiers

Private members cannot be accessed, or even viewed from outside the class. Only the class (and `friend` functions) can access private members.

By default, all class members are private.

The following **example** demonstrates the use of the `private` access modifier:

```
1 class Box {
2    double width; //this is a private member
3
4    public:
5       double length;
6       void setWidth(double wid);
7       double getWidth(void);
8 };
```

Practically, we define data in the `private` section and related functions in the `public` section so that they can be called from outside of the class.

Class Access Modifiers

The **protected** access modifier defines class members that are very similar to `private` members, but it provides one additional benefit that they can be accessed in derived (inherited) classes.

We will see later on in detail how **inheritance** works. For now you can check following **example** where I have derived one child class `SmallBox` from a parent class `Box`:

Class Access Modifiers


```
1  #include <iostream>
2  using namespace std;
3
4  class Box {
5     protected:
6        double width;
7  };
8
9  class SmallBox:Box { // SmallBox is the derived class.
10    public:
11       void setSmallWidth( double wid );
12       double getSmallWidth( void );
13 };
14
15 // Member functions of child class
16 double SmallBox::getSmallWidth(void) {
17    return width ;
18 }
19
20 void SmallBox::setSmallWidth( double wid ) {
21    width = wid;
22 }
23
24 // Main function for the program
25 int main() {
26    SmallBox box;
27
28    // set box width using member function
29    box.setSmallWidth(5.0);
30    cout << "Width of box : "<< box.getSmallWidth() << endl;
31
32    return 0;
33 }
```

Class Access Modifiers


```
1  #include <iostream>
2  using namespace std;
3
4  class Box {
5     protected:
6        double width;
7  };
8
9  class SmallBox:Box { // SmallBox is the derived class.
10    public:
11       void setSmallWidth( double wid );
12       double getSmallWidth( void );
13 };
14
15 // Member functions of child class
16 double SmallBox::getSmallWidth(void) {
17    return width ;
18 }
19
20 void SmallBox::setSmallWidth( double wid ) {
21    width = wid;
22 }
23
24 // Main function for the program
25 int main() {
26    SmallBox box;
27
28    // set box width using member function
29    box.setSmallWidth(5.0);
30    cout << "Width of box : "<< box.getSmallWidth() << endl;
31
32    return 0;
33 }
```

Constructor

A class **constructor** is a special function in a class that is called when a new object of the class is created.

A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.

Following **example** explains the concept of a constructor:

Constructor


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

class Line {
   public:
      void setLength( double len );
      double getLength( void );
      Line();  // This is the constructor
   private:
      double length;
};

// Member functions definitions including constructor
Line::Line(void) {
   cout << "Object is being created" << endl;
}
void Line::setLength( double len ) {
    length = len;
}
double Line::getLength( void ) {
   return length;
}

// Main function for the program
int main() {
   Line line;

   // set line length
   line.setLength(6.0);
   cout << "Length of line: " << line.getLength() <<endl;

   return 0;
}
```

Parameterized Constructor

A default constructor does not have any parameter, but if necessary, a constructor can have parameters. This can e.g. be used to assign an initial value to an object.

The following **example** shows this:

```
1  class Line {
2     public:
3        void setLength(double len);
4        double getLength(void);
5        Line(double len);  // This is the constructor
6
7     private:
8        double length;
9  };
10
11 // Member functions definitions including constructor
12 Line::Line( double len) {
13    cout << "Object is being created, length = " << len << endl;
14    length = len;
15 }
```

Parameterized Constructor

We call the constructor in the main function of the program:

```
1  int main() {
2     Line line(10.0);
3
4     // get initially set length.
5     cout << "Length of line : " << line.getLength() <<endl;
6
7     // set line length again
8     line.setLength(6.0);
9     cout << "Length of line : " << line.getLength() <<endl;
10
11    return 0;
12 }
```

Parameterized Constructor

We call the constructor in the main function of the program:

```
1  int main() {
2     Line line(10.0);
3
4     // get initially set length.
5     cout << "Length of line : " << line.getLength() <<endl;
6
7     // set line length again
8     line.setLength(6.0);
9     cout << "Length of line : " << line.getLength() <<endl;
10
11    return 0;
12 }
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

Class Destructor

Similarly to the constructor, the **destructor** is a special function which is called when created object is deleted.

More specifically, the destructor is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

A destructor has the exact same name as the class prefixed with a tilde ($\sim$) and it can neither return a value nor can it take any parameters.

The following **example** explains the concept of destructor:

Class Destructor


```
1  #include <iostream>
2  using namespace std;
3
4  class Line {
5     public:
6        void setLength(double len);
7        double getLength(void);
8        Line();   // This is the constructor declaration
9        ~Line();  // This is the destructor: declaration
10
11    private:
12       double length;
13 };
14
15 // Member functions definitions including constructor
16 Line::Line(void) {
17    cout << "Object is being created" << endl;
18 }
19 Line::~Line(void) {
20    cout << "Object is being deleted" << endl;
21 }
22 void Line::setLength(double len) {
23    length = len;
24 }
25 double Line::getLength(void) {
26    return length;
27 }
28
```

Parameterized Constructor

We call the constructor in the main function of the program:

```
1      int main() {
2      Line line;
3
4      // set line length
5      line.setLength(6.0);
6      cout << "Length of line: " << line.getLength() <<endl;
7
8      return 0;
9  }
```

Parameterized Constructor

We call the constructor in the main function of the program:

```
1      int main() {
2      Line line;
3
4      // set line length
5      line.setLength(6.0);
6      cout << "Length of line: " << line.getLength() <<endl;
7
8      return 0;
9   }
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created
Length of line: 6
Object is being deleted
```

Static Class Members

A **static** member is shared by all objects of the class.
When we declare a member of a class as `static` it means no matter how many objects of the class are created, there is only one copy of the `static` member.

Both data members and function members of a class can be declared as `static`.

All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class by redeclaring the static variable, using the scope resolution operator :: to identify which class it belongs to.

The following **example** illustrates the concept of static data members:

Static Class Members


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

class Box {
   public:
      static int objectCount;

      // Constructor definition
      Box(double l = 2.0, double b = 2.0, double h = 2.0) {
         cout <<"Constructor called." << endl;
         length = l;
         width = b;
         height = h;

         // Increase every time object is created
         objectCount++;
      }
      double Volume() {
         return length * width * height;
      }

   private:
      double length;     // Length of a box
      double width;     // Width of a box
      double height;     // Height of a box
};

// Initialize static member of class Box
int Box::objectCount = 0;
```

Parameterized Constructor

We call the constructor in the main function of the program:

```
1  int main(void) {
2     Box Box1(3.3, 1.2, 1.5);    // Declare box1
3     Box Box2(8.5, 6.0, 2.0);    // Declare box2
4
5     // Print total number of objects.
6     cout << "Total objects: " << Box::objectCount << endl;
7
8     return 0;
9  }
```

Parameterized Constructor

We call the constructor in the main function of the program:

```
1  int main(void) {
2     Box Box1(3.3, 1.2, 1.5);    // Declare box1
3     Box Box2(8.5, 6.0, 2.0);    // Declare box2
4
5     // Print total number of objects.
6     cout << "Total objects: " << Box::objectCount << endl;
7
8     return 0;
9  }
```

When the above code is compiled and executed, it produces the following result:

```
Constructor called.
Constructor called.
Total objects: 2
```

Static Class Members

By declaring a function member as `static`, you make it independent of any particular object of the class. A `static` member function can be called even if no objects of the class exist and the static functions are accessed using only the class name and the scope resolution operator `::`.

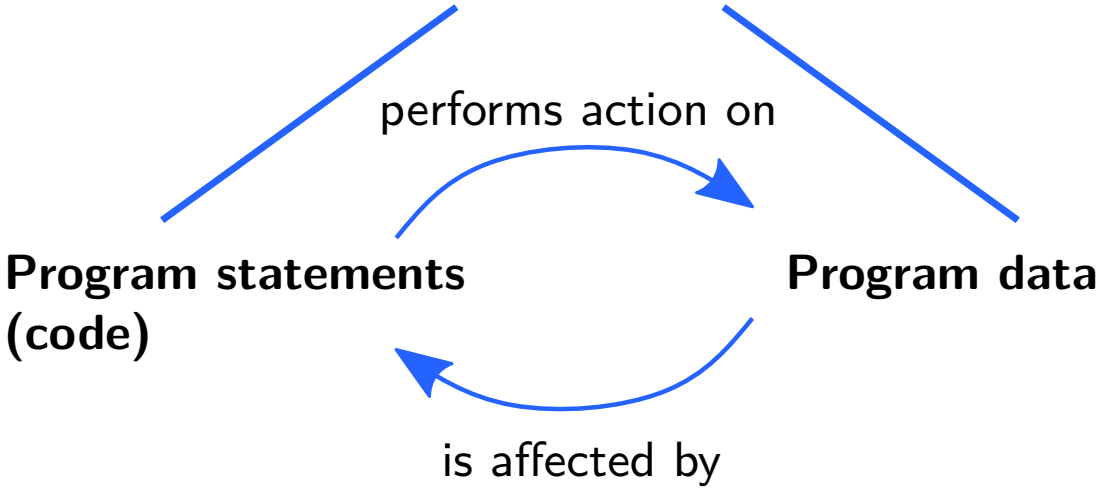
A `static` member function can only access static data member, other static member functions and any other functions from outside the class.

For **example**, we could supplement the `Box` class by the following static member function:

```
1   static int getCount() {
2   return objectCount;
3   }
```

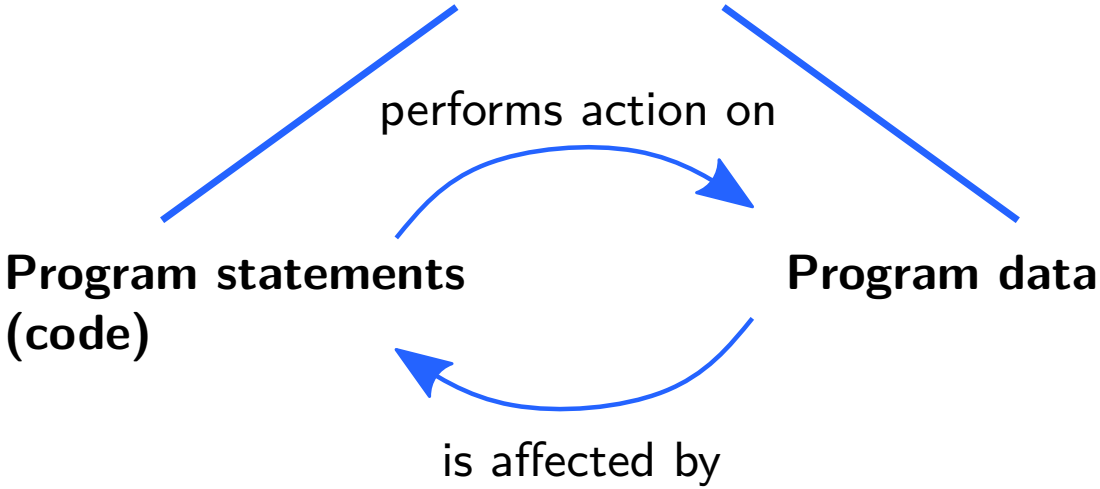
Encapsulation in C++

Like generally most programs, all C++ programs are composed of the following two fundamental elements:



Encapsulation in C++

Like generally most programs, all C++ programs are composed of the following two fundamental elements:



Encapsulation is an OOP concept that binds together the data and functions that manipulate the data.

Data encapsulation is a mechanism of bundling the data, and the functions that use them and data abstraction is a mechanism of exposing only the **interfaces** and hiding the implementation details from the user.

Encapsulation in C++

C++ supports the properties of encapsulation and data hiding through the creation of classes.

We already saw that a class can contain private, protected and public members. By default, all items defined in a class are private. For example:

```
1  class Box {
2     public:
3        double getVolume(void) {
4           return length * width * height;
5        }
6
7     private:
8        double length;     // Length of a box
9        double width;      // Width of a box
10       double height;     // Height of a box
11 };
```

The variables `length`, `witdth`, and `height` are private. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved.

Encapsulation in C++

Making one class a `friend` of another exposes the implementation details and reduces encapsulation.

A `friend` class can access private and protected members of other classes in which it is declared as a `friend`. It is sometimes useful to allow a particular class to access private and protected members of other classes.

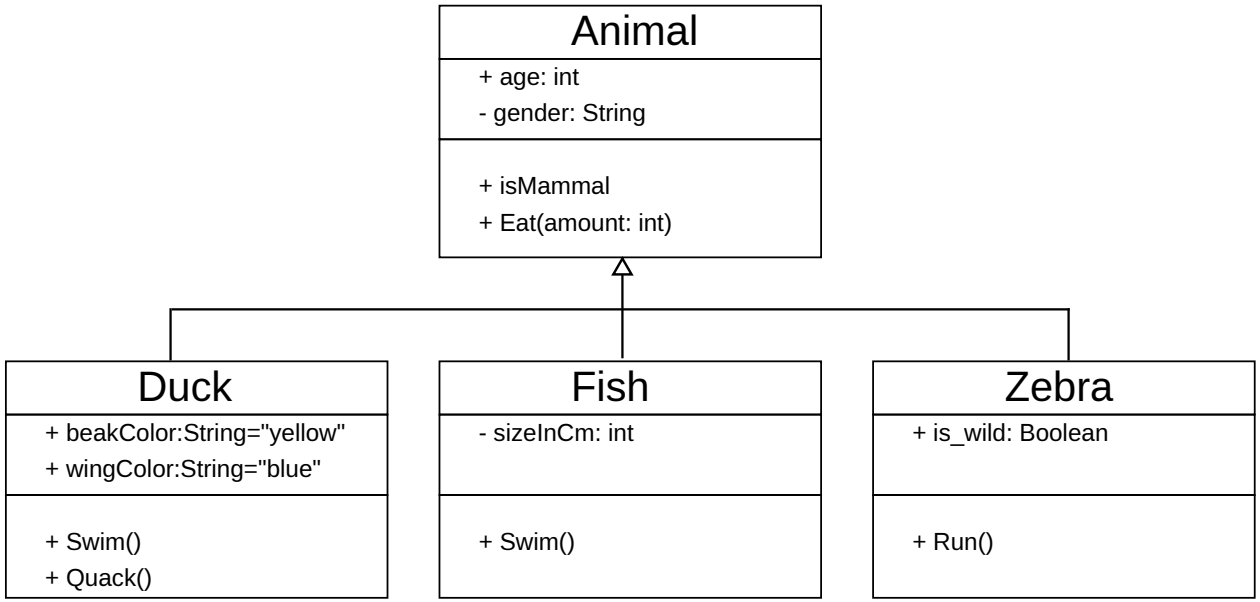
See the following **example**:

```
1  using namespace std;
2
3  class GFG {
4  private:
5      int private_variable;
6
7  protected:
8      int protected_variable;
9
10 public:
11     GFG()
12     {
13         private_variable = 10;
14         protected_variable = 99;
15     }
16
17     // friend class declaration
18     friend class F;
19 };
```

Inheritance in C++

One of the most important concepts in object-oriented programming is that of **inheritance**. Inheritance allows us to define a class in terms of another class, allowing to reuse the code functionality.

In the following **example**, the class `Animal` serves as the base class generalizing animal species (e.g. in a computer game), which are themselvse represented as classes that inherit and extend the functionality of the base class.



Inheritance in C++

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes.

To define a derived class, we use a class derivation list to specify the base class(es). See the following **example** in which we have a base class `Shape` and its derived class `Rectangle` as follows:

Inheritance in C++


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

class Shape {  // Base class
   public:
      void setWidth(int w) {
         width = w;
      }
      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

class Rectangle: public Shape { // Derived class
   public:
      int getArea() {
         return (width * height);
      }
};

int main(void) {
   Rectangle Rect;
   Rect.setWidth(5);
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```

Inheritance in C++

When the above code is compiled and executed, it produces the following result, showing us that the `getArea()` function from class `Shape` was reused through inheritance:

```
Total area: 35
```

Inheritance in C++

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

A derived class inherits all base class methods with the following exceptions:

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

Multiple Inheritance in C++

A C++ class can inherit members from more than one class. See the following **example**:

```
1  #include <iostream>
2  using namespace std;
3
4  // Base class Shape
5  class Shape {
6     public:
7        void setWidth(int w) {
8           width = w;
9        }
10       void setHeight(int h) {
11          height = h;
12       }
13
14    protected:
15       int width;
16       int height;
17 };
18
19 // Base class PaintCost
20 class PaintCost {
21    public:
22       int getCost(int area) {
23          return area * 70;
24       }
25 };
```

Class Member Functions

continuation of source code:

```
26 // Derived class
27 class Rectangle: Shape, PaintCost {
28    public:
29       int getArea() {
30          return (width * height);
31       }
32 };
33
34 int main(void) {
35    Rectangle Rect;
36    int area;
37
38    Rect.setWidth(5);
39    Rect.setHeight(7);
40
41    area = Rect.getArea();
42
43    // Print the area of the object.
44    cout << "Total area: " << Rect.getArea() << endl;
45
46    // Print the total cost of painting
47    cout << "Total paint cost: $" << Rect.getCost(area) << endl;
48
49    return 0;
50 }
```

Class Member Functions

When the above code is compiled and executed, it produces the following result, showing us that the inheritance both from class `Shape` and class `PaintCost` works:

```
Total area: 35
Total paint cost: $2450
```

Polymorphism in C++

The word **polymorphism** means *having many forms*. Typically, polymorphism occurs when there is a hierarchy of classes which are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function. This is done by **function overriding**.

Consider the following **example** where a base class has been derived by other two classes:

Polymorphism in C++


```
1  class Shape {
2     protected:
3        int width, height;
4
5     public:
6        Shape(int a = 0, int b = 0){
7           width = a;
8           height = b;
9        }
10       int area() {
11          cout << "Parent class area :" << width * height << endl;
12          return width * height;
13       }
14 };
15
16 class Rectangle: public Shape {
17    public:
18       Rectangle(int a = 0, int b = 0):Shape(a, b) { }
19
20       int area() {
21          cout << "Rectangle class area :" << width * height << endl;
22          return (width * height);
23       }
24 };
25
26 class Triangle: public Shape {
27    public:
28       Triangle(int a = 0, int b = 0):Shape(a, b) { }
29
30       int area() {
31          cout << "Triangle class area :" << (width * height)/2 << endl;
32          return (width * height / 2);
33       }
34 };
```

Polymorphism in C++

The reason for the incorrect output is that the call of the function `area()` is set once by the compiler as the version defined in the base class. This is called **static resolution** (also called **early binding**) of the function call, or static linkage: the function call is fixed before the program is executed.

Virtual Function in C++

Sometimes we want that the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage** (also called **late binding**).

A **virtual** function is a function in a base class that is declared using the keyword `virtual`. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function. See the following **example**:

```
1  virtual int area() {
2      cout << "Parent class area :" << width * height << endl;
3      return width * height;
4      }
```

Virtual Function in C++

We can also define **pure virtual functions**:

This is useful when a function will only be implemented in a derived class but there is no meaningful definition you could give for the function in the base class.

We can change the virtual function `area()` in the base class to the following:

```
1  class Shape {
2     protected:
3        int width, height;
4
5     public:
6        Shape(int a = 0, int b = 0) {
7           width = a;
8           height = b;
9        }
10
11       // pure virtual function
12       virtual int area() = 0;
13 };
```

Summary: Object Oriented Programming

With this introduction to Object Oriented Programming (OOP), we have seen the major concept that makes C++ different from C.

What we have learned also translates well to OOP found in other programming languages, like Java and Python.

As this introduction cannot cover everything, I can recommend looking up more online if you want to go deeper into OOP.

An Outlook: Integrating C++ and Python

So far, we have used both Python and C++. We have seen they have somewhat different usage, and different benefits come with them.

To make best use of both, in the next session, we will extend this to **integrating C++ and Python** into the same application or generally, software project.