What is OOP?

In this lesson, we'll learn about the historical background of OOP and also key features of object-oriented programming.

WE'LL COVER THE FOLLOWING

Historical Background

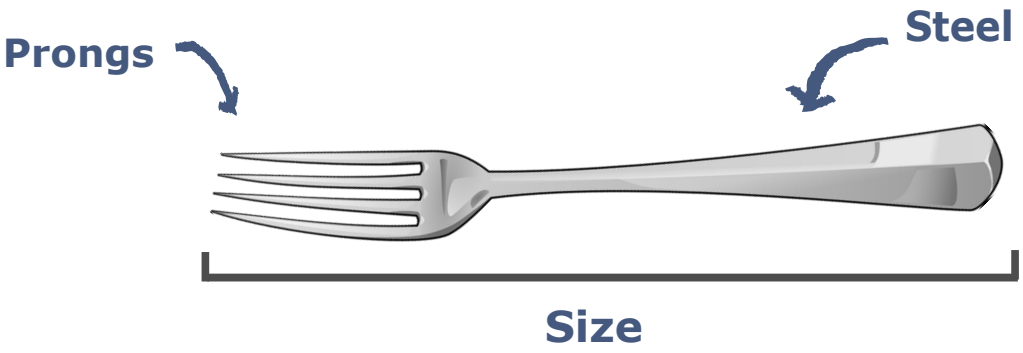
It was in the `60's` and early `70's` when the idea of **object-oriented** programming started occupying the minds of programmers. Keeping in view the benefits acquired through object orientation in SIMULA, the scientists encouraged languages using the same approach for programming. **Object-oriented** programming was a complete paradigm shift from the popular structural programming.

Object: A Fundamental Entity

Object-oriented programming is based on the idea of an `object`. An `object` is an entity with some *data* and *operations*. Data is also referred to as *properties* of the object whereas operations include accessing and modifying those properties along with other functions that depict the behavior of the object.

Example

A **fork** is an object with properties including a number of *prongs*, its *size*, and *material* (made of plastic or metal), etc. Behavior and functions of a fork include shredding, squashing, making design or may be simply eating.



Fork

Explanation

Programmers realized that when we represent entities in the program as objects, having their behaviors and properties, it becomes easy to deal with the increasing code complexity as well as the code becomes more reusable. So, a *fork* object can be part of a dinner set, and a similar object may also be sold separately. Once, we know its properties and behavior, we only need to reuse the same piece of code whenever a *fork* is needed.

In the next lesson, we'll learn about how C++ is an object-oriented programing.

Arguments and Function Scope

This lesson deals with the nature of the arguments being passed to a function.

WE'LL COVER THE FOLLOWING

In the previous lesson, we learned how to pass arguments into a function. It is a very simple process, but there are a few things we need to know about how the C++ compiler treats these arguments.

Pass-by-Value

The compiler never actually sends the variables into the function. Instead, **a copy of each argument is made and used in the scope of the function**. This concept is known as **passing-by-value**. Only the values of the arguments are passed into the functions, not the arguments themselves.

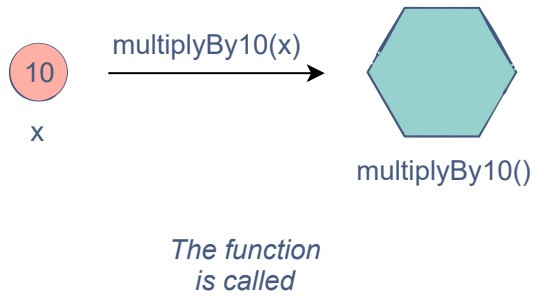
When the function ends, all the argument copies and the variables created inside the function are destroyed forever. Basically, changing the value of an argument inside a function won't change it outside the function.

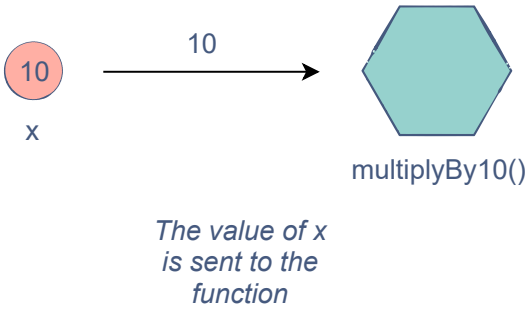
Have a look at the function below, which multiplies its argument by 10:

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

void multiplyBy10(int num){
  num = num * 10;
}

int main(){
  int x = 10;

  cout << "Before function call" << endl;
  cout << "x: " << x << endl;
```

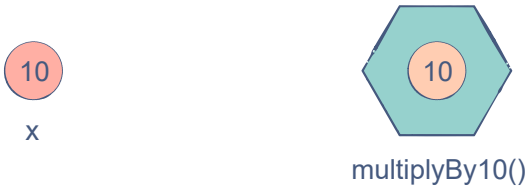
```
  // Multiplying by 10
  multiplyBy10(x);

  cout << "After function call" << endl;
  cout << "x: " << x << endl;
}
```

Just as we discussed, the value of `x` did not change outside the function scope. The illustration below will help up understanding this better.

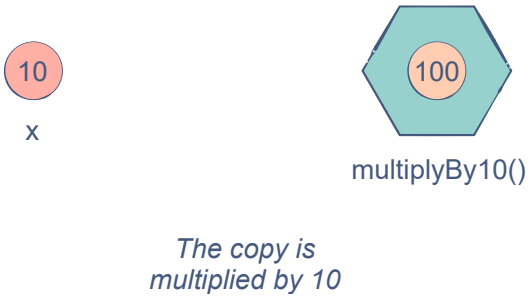


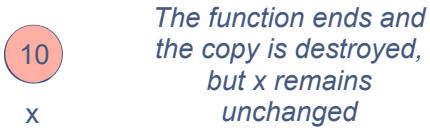


The value of x is sent to the function



A copy of x is used in the function scope





This is normal behavior in C++ but in some cases, we might want a function to manipulate variables outside its scope. Well, there is a solution for that.

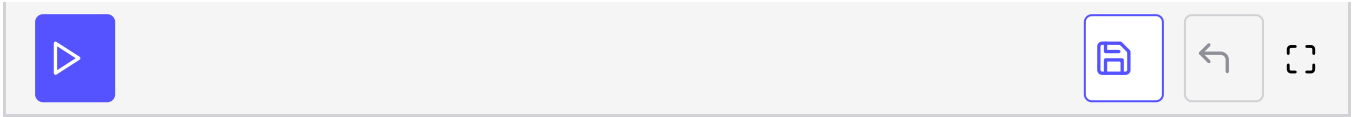
Pass-by-Reference

This is the second approach of passing elements to functions. As we know, every variable is stored at an address in the memory.

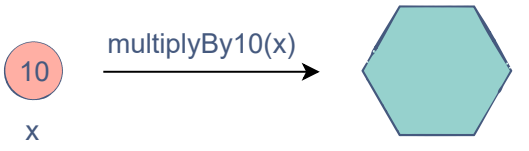
Passing-by-reference means passing the address of a variable as an argument instead of the variable itself. This will ensure that the function manipulates the actual variable.

To access the address of a variable, we need to add the `&` operator before it. Let's refactor the `multiplyBy10` so that we pass `x` by reference.

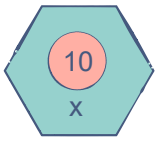
```
#include <iostream>
using namespace std;

void multiplyBy10(int &num){ // num will contain the reference of the
                             // input variable
  num = num * 10;
}

int main(){
  int x = 10;

  cout << "Before function call" << endl;
  cout << "x: " << x << endl;

  // Multiplying by 10
  multiplyBy10(x);

  cout << "After function call" << endl;
  cout << "x: " << x << endl; // The actual value of x is changed!
}
```



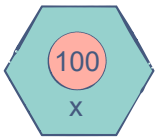
Voila! With a simple `&`, we can allow a function to edit things outside its scope. Let's see what happens when we pass arguments by reference:



The function is called



The value of x is sent to the function



multiplyBy10()

x is multiplied by 10

3 of 4

The function ends, and the value of x has changed

4 of 4

Now, we can decide when we need to pass an argument by value or by

reference. The next lesson will explain how different arguments can affect the same function.

Overloading Functions

In this lesson, we'll see the same function perform different operations based on its arguments.

WE'LL COVER THE FOLLOWING

We previously learned that each function has a specific number of arguments which have to be passed when calling it. If we increase or decrease the number of arguments in the call, we'll get a compilation error:

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

double product(double x, double y){
  return x * y;
}

int main() {
  cout << product(10, 20) << endl; // Works fine
  cout << product(10) << endl; // Error!
}
```

Line 10 blows up the code.

The compiler doesn't know how to handle arguments it wasn't expecting. However, one of the coolest things we can do with functions is that we can **overload** them.

What is Overloading?

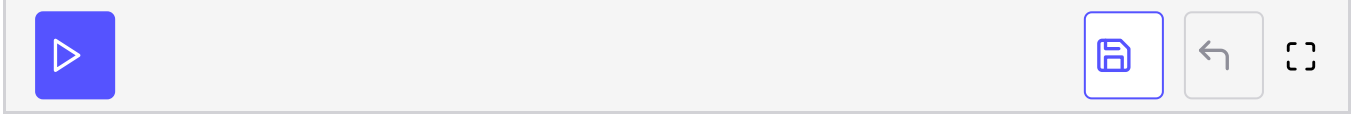
> Overloading refers to making a function perform different operations

based on the nature of its arguments.

We could redefine a function several times and give it different arguments and function types. When the function is called, the appropriate definition will be selected by the compiler!

Let's see this in action by overloading the `product` function which we just wrote:

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

double product(double x, double y){
  return x * y;
}

// Overloading the function to handle three arguments
double product(double x, double y, double z){
  return x * y * z;
}

// Overloading the function to handle floats
float product(float x, float y){
  return x * y;
}

int main() {
  double x = 10;
  double y = 20;
  double z = 5;
  float a = 12.5;
  float b = 4.654;
  cout << product(x, y) << endl;
  cout << product(x, y, z) << endl;
  cout << product(a, b) << endl;
  // cout << product(x) << endl;
}
```



The product function now works in three different ways!

In the code above, we see the same function behaving differently when encountering different types of inputs. We still have to define which cases it can handle. **Line 27** would crash since `product` doesn't know how to handle a single argument.

> **Note**: Functions which have no arguments and differ only in the return

types cannot be overloaded since the compiler won't be able to differentiate between their calls.

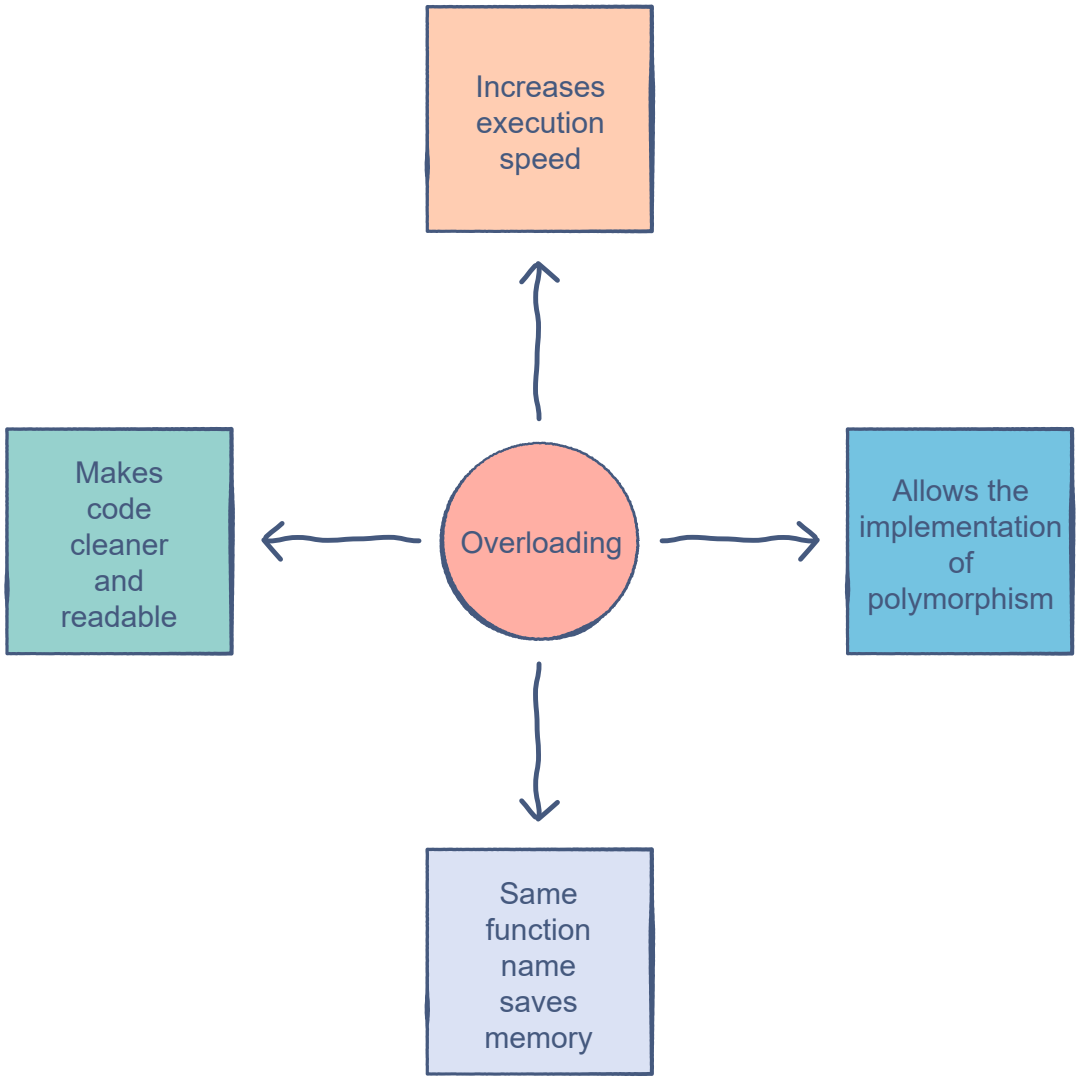
Advantages of Function Overloading

One might wonder that we could simply create new functions to perform different jobs rather than overloading the same function. However, under the hood, overloading saves us memory in the system. Creating new functions is more costly compared to overloading a single one.

Since they are memory-efficient, overloaded functions are compiled faster compared to different functions, especially if the list of functions is long.

An obvious benefit is that the code becomes simple and clean. We don't have to keep track of different functions.

Polymorphism is a very important concept in object-oriented programming. It will come up later on in the course, but function overloading plays a vital role in its implementation.



We know enough about function in order to delve deeper into the principles of OOP. In the next chapter, we will explore **pointers**.

Before that, be sure to check out the quiz and exercises ahead to test your concepts of functions. Good luck!

Pointers and Dynamic Memory

So far, we've only discussed the behavior of pointers with the stack. Let's see how pointers can be used in dynamic memory.

WE'LL COVER THE FOLLOWING

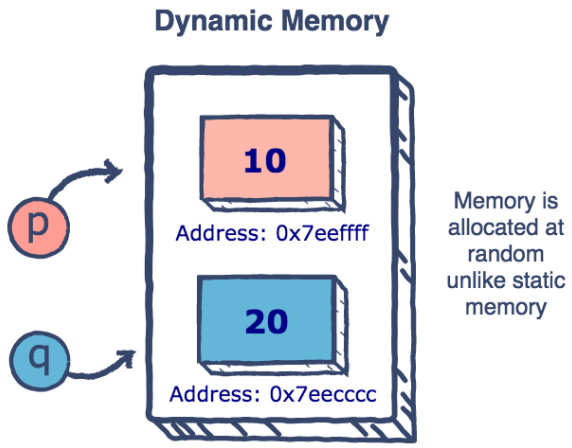
Heap: The Dynamic Memory Store

In the first lesson, we learned about the static portion of memory in C++ known as the **stack**. The counterpart for this is the dynamic section of memory known as the **heap**. This is a vast space of memory which is not being managed by the CPU.

We can specify the amount of space we want, and a random portion of the heap will be reserved for us. While the heap does allow us to use as much space as we want, the look-up operation is slower compared to a stack. However, a variable in dynamic memory (heap) can have a “global” scope instead of just being a local variable for a function.

Objects in the Heap

So, how do we access the heap? ***Pointers***. This is the true purpose of a pointer. It allows us to create dynamic objects using the `new` command.



The `delete` command releases the memory reserved by a certain pointer, making it available for future use again.

Do keep in mind that the pointer itself is stored in the stack, but can point to objects in both the stack and heap.

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

int main() {
  int *p = new int;   // dynamic memory reserved for an integer
  *p = 10;   // the object is assigned the value of 10
  cout << "The value of the object p points to: " << *p << endl;

  int *q = p;   // both pointers point to the same object
  cout << "The value of the object q points to: " << *q << endl;

  double *arr = new double[500]; // an array of size 500 has been created in the heap
  arr[0] = 50;
  cout << "arr[0]: " << arr[0] << endl;

  // delete pointers and free up space
  delete p, q;
  delete[] arr;
  cout << "p now points to a random value and should not be accesed: " << *p << endl;
  p = new int(5); // The pointer can now be re-used to point to something else
  cout << "The value of the object p points to: " << *p << endl;
}
```

In the code above, the `new` objects are created during runtime (instead of compile time). This is an advantage since we don't need to specify the amount of memory we need at compile time.

Pointer Cheat Sheet

Syntax	Purpose
`int *p`	Declares a pointer `p`
`p = new int`	Creates an integer variable in dynamic memory and places its address in p
`p = new int(5)`	Creates an integer object in dynamic memory with the value of 5
`p = new int[5]`	Creates a dynamic array of size 5 and places the address of its first index in p
`p = &var`	Points `p` to the `var` variable
`*p`	Accesses the value of the object `p` points to
`*p = 8`	Updates the value of the object `p` points to
`p`	Accesses the memory address of the object `p` points to

We've learned the basics about the nature of pointers. In the next lesson, we'll take a look at the interaction between pointers and functions.

What is a Class?

This section will familiarize us with the basic building blocks of object-oriented programming: Classes.

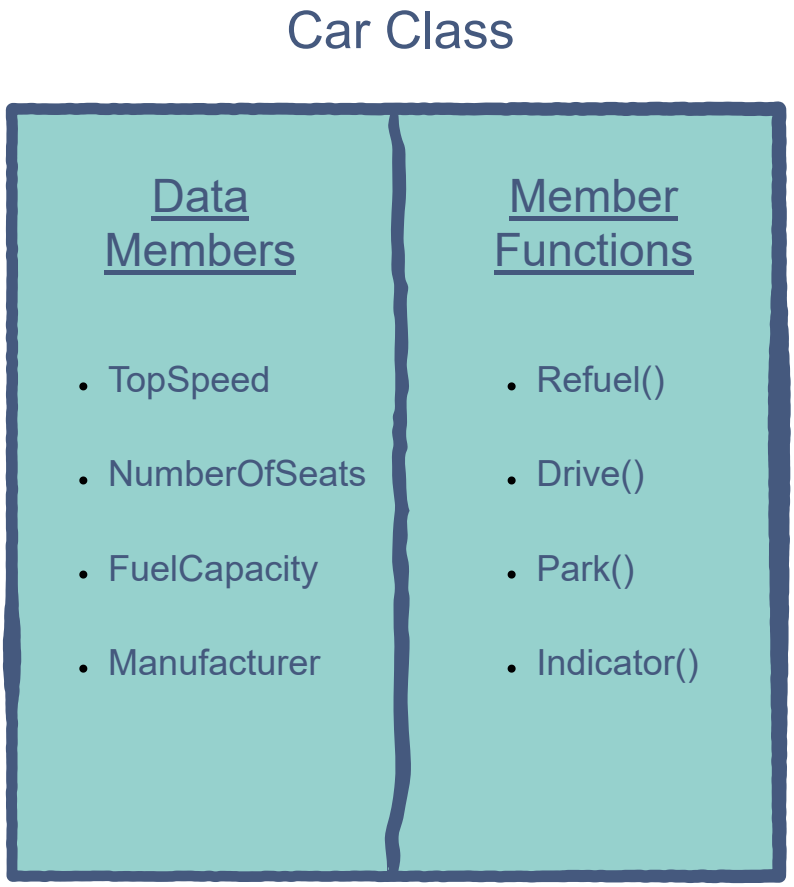
WE'LL COVER THE FOLLOWING

Custom Objects

In C++, we have several different data types like `int`, `string`, `bool` etc. An object can be created out of any of those types. An *object* is an instance of a class. Well, object-oriented programming wouldn't make sense if we couldn't make our own custom objects. This is where **classes** come into play.

Classes are used to create *user-defined data types.* The predefined data types in C++ are classes themselves. We can use these basic data types to create our own class. The cool part is that our class can contain multiple variables, pointers, and functions which would be available to us whenever a class object is created.

Let's start off with an example of a **car** class. Below, we can see the **attributes** that a car object would contain:



Car Class

We can see two types of attributes in the `Car` class above. In general, these two categories are present in all classes.

Data Members

These are also known as the member variables of a class. This is because they contain the information relevant to the object of the class. A car object would have a top speed, the number of seats it has, and so many other pieces of data that we could store in variables.

Member Functions

This category of attributes enables the class object to perform operations using the member variables. In the case of the car class, the `Refuel()` function would fill up the `FuelTank` property of the object.

Benefits of Using Classes

The concept of classes allows us to create complex objects and applications in

C++. This is why classes are the basic building blocks behind all of the OOP's principles.

Classes are also very useful in compartmentalizing the code of an application. Different components could become separate classes which would interact through interfaces. These ready-made components will also be available for use in future applications.

The use of classes makes it easier to maintain the different parts of an application since it is easier to make changes in classes (more on this later).

In the next lesson, we will start our journey into creating a class.

Access Modifiers

In this lesson, you will learn about the private, public and protected members.

WE'LL COVER THE FOLLOWING

In C++, we can impose access restrictions on different data members and member functions. The restrictions are specified through **access modifiers**. Access modifiers are tags we can associate with each member to define which parts of the program can access it directly.

There are three types of access modifiers. Let's take a look at them one by one.

Private

A private member cannot be accessed directly from outside the class. The aim is to keep it hidden from the users and other classes. It is a popular practice to **keep the data members private** since we do not want anyone manipulating our data directly. By default, all declared members are private in C++. However, we can also make members private using the `private:` heading.

```
class Class1 {
  int num; // This is, by default, a private data member
  ...
};

class Class2 {
  private: // We have explicitly defined that the variable is private
  int num;
  ...
};
```

Public

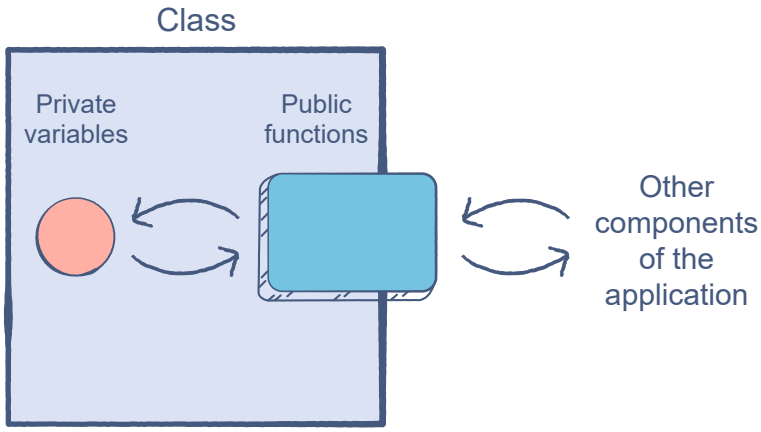
Public

This tag indicates that the members can be directly accessed by anything which is in the same scope as the class object.

Member functions are usually public as they provide the interface through which the application can communicate with our private members.

Public members can be declared using the `public:` heading.

```
class myClass {
  int num; // Private variable

  public: // Attributes in this list are public
  void setNum(){
    // The private variable is directly accessible over here!
  }
};
```



Public members of a class can be accessed by a class object using the `.` operator. For example, if we have an object `c` of type `myClass`, we could access `setNum()` like this:

```
myClass c; // Object created
c.setNum(); // Can manipulate the value of num
c.num = 20; // This would cause an error since num is private
```

Protected

The protected category is unique. The access level to the protected members lies somewhere between private and public. The primary use of the protected tag is to implement **inheritance**, which is the process of creating classes out of classes. Since this is a whole other topic for the future, we'll refrain from

going into details right now.

We've seen a hint of how data members can be created in a class. In the next lesson, we will go into further details on the topic.

Constructors

In this lesson, we explore the world of constructors and learn why they are necessary for a class.

WE'LL COVER THE FOLLOWING

What is a Constructor?

As the name suggests, the constructor is used to *construct* the object of a class. It is a special member function that outlines the steps that are performed when an instance of a class is created in the program.

> A constructor's **name** must be exactly the **same** as the name of its class.

The constructor is a special function because it **does not have a return type**. We do not even need to write `void` as the return type. It is a good practice to declare/define it as the first member function.

So, let's study the different types of constructors and use them to create class objects.

Default Constructor

The default constructor is the most basic form of a constructor. Think of it this way:

> In a default constructor, we define the default values for the data

> members of the class. Hence, the constructor creates an object in which the data members are initialized to their default values.

This will make sense when we look at the code below. Here, we have a **Date** class, with its default constructor, and we'll create an object out of it in `main()`:

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

class Date {
  int day;
  int month;
  int year;

  public:
  // Default constructor
  Date(){
    // We must define the default values for day, month, and year
    day = 0;
    month = 0;
    year = 0;
  }

  // A simple print function
  void printDate(){
    cout << "Date: " << day << "/" << month << "/" << year << endl;
  }
};

int main(){
  // Call the Date constructor to create its object;

  Date d; // Object created with default values!
  d.printDate();
}
```

Notice that when we created a `Date` object in **line 28**, we don't treat the constructor as a function and write this:

```
d.Date()
```

We create the object just like we create an `integer` or `string` object. It's that easy!

The default constructor does not need to be explicitly defined. Even if we

The default constructor does not need to be explicitly defined. Even if we don't create it, the C++ compiler will call a default constructor and set data members to `null` or `0`.

Parameterized Constructor

The default constructor isn't all that impressive. Sure, we could use `set` functions to set the values for `day`, `month` and `year` ourselves, but this step can be avoided using a **parameterized constructor**.

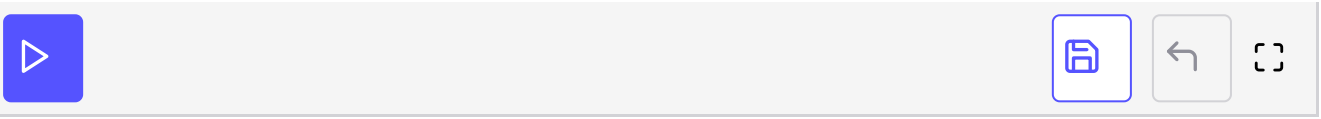
> In a parameterized constructor, we pass arguments to the constructor and set them as the values of our data members.

We are basically overriding the default constructor to accommodate our preferred values for the data members.

Let's try it out:

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

class Date {
  int day;
  int month;
  int year;

  public:
  // Default constructor
  Date(){
    // We must define the default values for day, month, and year
    day = 0;
    month = 0;
    year = 0;
  }

  // Parameterized constructor
  Date(int d, int m, int y){
    // The arguments are used as values
    day = d;
    month = m;
    year = y;
  }

  // A simple print function
  void printDate(){
    cout << "Date: " << day << "/" << month << "/" << year << endl;
  }
};
```

```
int main(){
  // Call the Date constructor to create its object;


  Date d(1, 8, 2018); // Object created with specified values!
  d.printDate();
}
```



This is much more convenient than the default constructor!

`this` Pointer

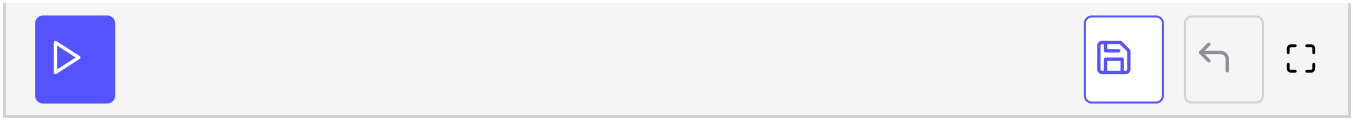
The `this` pointer exists for every class. It points to the class object itself. We use the pointer when we have an argument which has the same name as a data member. `this->memberName` specifies that we are accessing the `memberName` variable of the particular class.

> **Note**: Since `this` is a pointer, we use the `->` operator to access members instead of `.`.

Let's see it in action:

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

class Date {
  int day;
  int month;
  int year;

  public:
  // Default constructor
  Date(){
    // We must define the default values for day, month, and year
    day = 0;
    month = 0;
    year = 0;
  }

  // Parameterized constructor
  Date(int day, int month, int year){
    // Using this pointer
    this->day = day;
    this->month = month;
    this->year = year;
  }
```

```
  // A simple print function
  void printDate(){

    cout << "Date: " << day << "/" << month << "/" << year << endl;
  }
};

int main(){
  // Call the Date constructor to create its object;

  Date d(1, 8, 2018); // Object created with specified values!
  d.printDate();
}
```



This marks the end of our discussion on *constructors*. In the next lesson, we will discuss the opposite of constructors, **destructors**.

Destructors

In this lesson, we will study the purpose of destructors.

WE'LL COVER THE FOLLOWING

What is a Destructor?

A **destructor** is the opposite of a constructor. It is called when the object of a class is **no longer in use**. The object is destroyed and the memory it occupied is now free for future use.

`C++` does not have transparent garbage collection like `Java`. Hence, in order to efficiently free memory, we must specify our own destructor for a class.

In this destructor, we can specify the additional operations which need to be performed when a class object is deleted.

A class destructor can be created in a similar way to the constructor, except that the destructor is preceded by the `~` keyword.

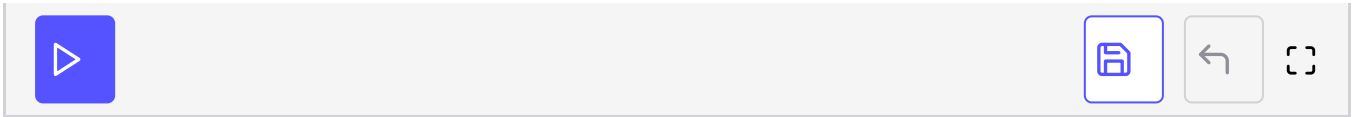
Explicit Garbage Collection

A small degree of garbage collection can be easily achieved through smart pointers. A smart pointer, the `shared_ptr` in particular, keeps a reference count for the object it points. When the counter comes down to `0`, the object is deleted.

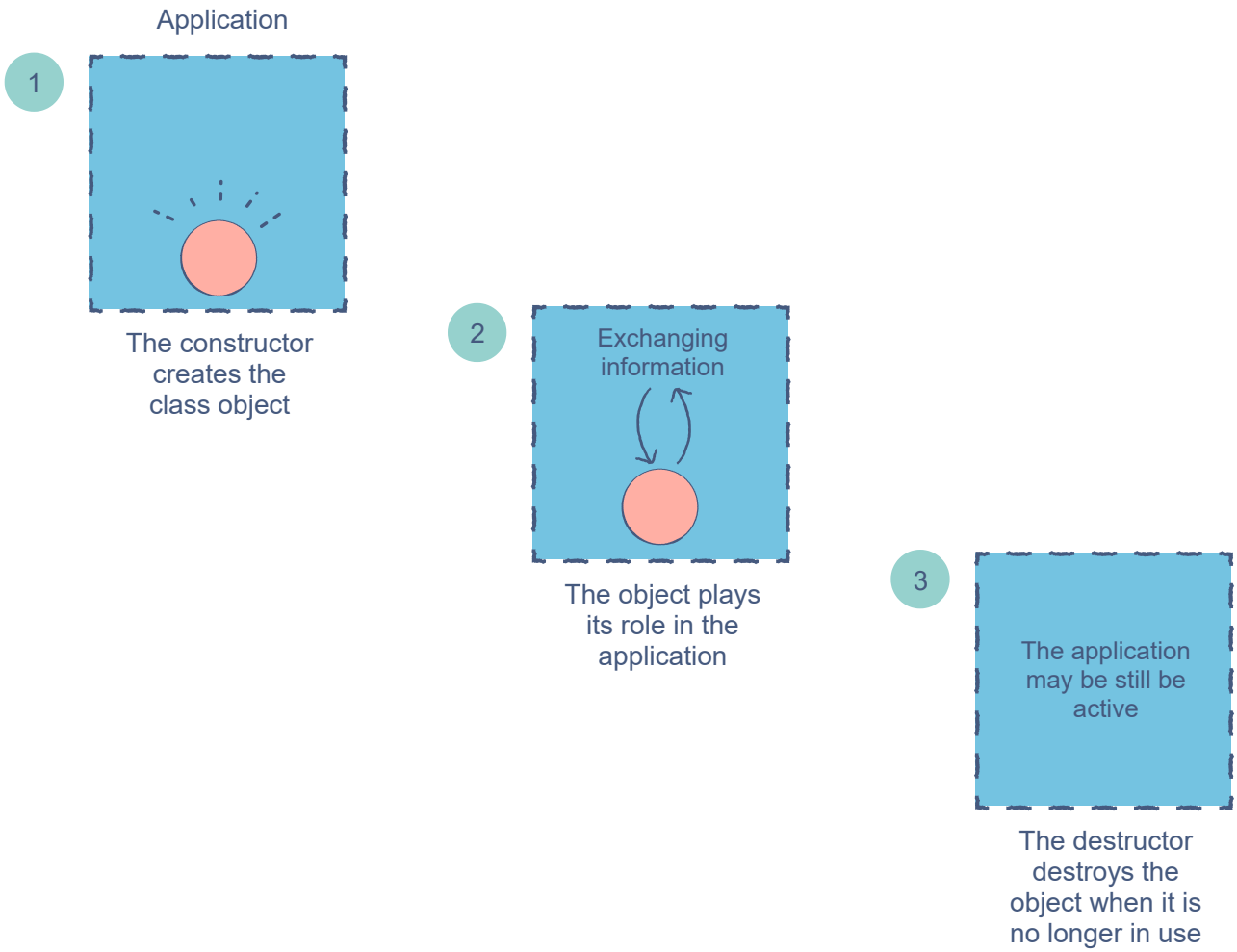
It's time to make a destructor and see how it behaves.

Example

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

class Date {
  int day;
  int month;
  int year;

  public:
  // Default constructor
  Date(){
    // We must define the default values for day, month, and year
    day = 0;
    month = 0;
    year = 0;
  }

  // Parameterized constructor
  Date(int d, int m, int y){
    // The arguments are used as values
    day = d;
    month = m;
    year = y;
  }

  // A simple print function
  void printDate(){
    cout << "Date: " << day << "/" << month << "/" << year << endl;
  }

  ~Date(){
    cout << "Deleting date object" << endl;
  }
};

int main(){
  Date d(1, 8, 2018);
  d.printDate();
}
```



As we can see, the destructor is automatically called and the memory is freed up. What's interesting is that the `cout` statement we specified is also executed on the destructor call.



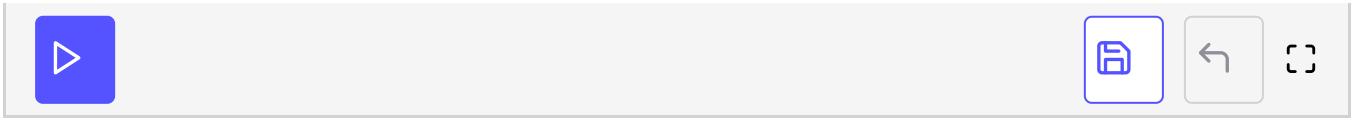
The lifecycle of an object.

Destructors and Pointers

In the case of pointers, destructors are called when we issue the `delete` command:

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

class Date {
  int day;
  int month;
  int year;

  public:
  // Default constructor
  Date(){
    // We must define the default values for day, month, and year
    day = 0;
    month = 0;
    year = 0;
  }

  // Parameterized constructor
```

```
  Date(int d, int m, int y){
    // The arguments are used as values
    day = d;

    month = m;
    year = y;
  }

  // A simple print function
  void printDate(){
    cout << "Date: " << day << "/" << month << "/" << year << endl;
  }

  ~Date(){
    cout << "Deleting date object" << endl;
  }
};

int main(){
  Date* d = new Date(1, 8, 2018); // Object created in dynamic memory
  d->printDate();
  delete d;
  cout << "End of program" << endl;
}
```



Destroying Objects is Important

If we don't deallocate the memory for the objects which are not in use, we will end up with **memory leaks** and no space for our application to work long term.

In the next lesson, we will learn about friend functions and their uses.

Friend Functions

Now, we'll take a look at a special category of functions called friends.

So far we have observed that the private data members of a class are only accessible through the functions present in that class. Nothing from outside can manipulate the class object without using its functions.

What if we need to access class variables in a function which is not a part of the class? That function would have to become a **friend** of the class.

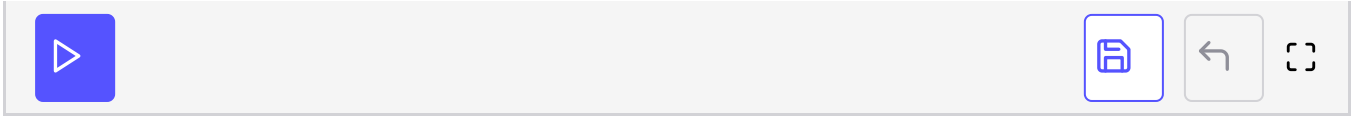
> A friend function is an independent function which has access to the variables and methods of its befriended class.

To create a friend function for a class, it must be declared in the class along with the `friend` keyword.

Let's create a `Ball` class to explain this better:

```
#include <iostream>
#include <string>

using namespace std;

class Ball{
  double radius;
  string color;

  public:
  Ball(){
    radius = 0;
    color = "";
  }

  Ball(double r, string c){
    radius = r;
    color = c;
  }

  void printVolume();
  void printRadius();
```

```
  // The friend keyword specifies that this is a friend function
  friend void setRadius(Ball &b, double r);

};

// This is a member function that calculates the volume.
void Ball::printVolume(){
  cout << (4/3) * 3.142 * radius * radius * radius << endl;
}

void Ball::printRadius(){
  cout << radius << endl;
}

// The implementation of our friend function
void setRadius(Ball &b, double r){
  b.radius = r;
}

 int main(){
   Ball b(30, "green");
   cout << "Radius: ";
   b.printRadius();
   setRadius(b, 60);
   cout << "New radius: ";
   b.printRadius();
   cout << "Volume: ";
   b.printVolume();
 }
```



In line 25, we can see that the `Ball` object is being passed by reference to the friend function. This is a crucial step in the functionality of the friend. If the object is not passed by reference, the changes made in the friend function will not work outside its scope. Basically, our object will not be altered.

The `setRadius()` function is completely independent of the `Ball` class, yet it has access to all the private variables. This is the beauty of the `friend` keyword.

This concludes our discussion on the basics of the classes in C++. The next section deals with the concept of data hiding, which plays a pivotal role in implementing efficient classes.

Creating a Function

In this lesson, we will learn how to create functions in C++ and use them in our program.

WE'LL COVER THE FOLLOWING

Similar to variables, functions need to be defined before compilation. Every function has a name and a set of operations it needs to perform. The first part of creating a function **declaration**.

Declaration

The declaration of a function means defining its **name**, **type**, and **argument(s)**. This may sound confusing right now but we'll get the hang of it really soon. Here's the template for function declaration:

```
type functionName(argument(s));
```

Let's take a look at the three components one by one:

- `type` refers to the type of value the function produces. In formal terms, we say that a function **returns** something when we use it. The type of object it returns must be specified in the declaration. So, if a function returns a number, its `type` would be `int` or `double` etc. Any data type available in C++ can be used as the function's type. If the function does not return anything, its type will be `void`.

- `functionName` is simply the label we'll use for the function, just like we do for variables.

- `arguments` are the inputs of a function. We can give a function different

objects as arguments, and the function will then perform operations using them. For each argument, a data type and name must be defined. A function could also have no arguments at all. In that case, the `arguments` section remains empty.

Now that we've been through all the components of function declaration, let's see a few examples:

```
int sum(int num1, int num2); // Returns an integer which is a sum of
                             //two integers, num1 and num2

string blah();  // Returns a string and does not contain any arguments

double[] createSortedArray(int size); // Returns an array of doubles

void printName(string name); // Prints a string
```

> **Note**: It is a good practice to give meaningful names to the function and its arguments. That makes the code much more readable.

It's time to move on to the second part of creating a function: **definition**.

Definition

The definition (also known as implementation) of a function refers to the set of instructions which the function performs. Without the definition, a function will not know what to do. Hence, we need to make sure our implementation is flawless and doesn't produce any bugs. Here's the template for the definition of a function:

```
 type functionName(arguments) {
  // Definition
}
```

The curly braces, `{}`, contain the definition of a function. This is known as the **scope** of the function (more on this later). Here, we can write a normal C++ code which will execute once every time the function is called.

You may have noticed by now that the `int main(){}` in a C++ program is also a function! It is the function that the compiler runs to execute our code.

To return something from a function, we use the `return` command. This ends

To return something from a function, we use the `return` command. This ends the function.

Below, we've defined a few functions and called them in the main program. The first approach only uses the definition to create a function. The second approach first declares the functions and then defines them below the `main()`.

Definition | Declaration+Definition

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

int sum(int num1, int num2){
  return num1 + num2;
}

string blah(){
  string s = "Hello World";
  return s;
}

void printName(string name){
  cout << name << endl;
  // No return statement needed as this is a void function
}

int main()
{
  int x = 10;
  int y = 20;
  int total = sum(x, y); // We store the returned value into a variable
  // of the corresponding type
  cout << total <<endl;

  string str = blah();
  cout << str <<endl;

  printName("Educative");  // This is a void function, so it doesn't need to be stored
}
```

At this point, we're equipped to write basic functions. The basic structure of the functions is clear. Remember, a function can call other functions as well. After all, the `main` function does that as well.

In the next lesson, we will learn some interesting details about the arguments of a function.

Encapsulation

This lesson shows us how to implement the first component of data hiding: encapsulation.

WE'LL COVER THE FOLLOWING

A Real Life Example

For the sake of explanation, we'll start off by creating a simple `movie` class which contains three members:

```
class Movie{
  string title;
  int year;
  string genre;

  public:
  Movie(){
    title = "";
    year = -1;
    genre = "";
  }

  Movie(string t, int y, string g){
    title = t;
    year = y;
    genre = g;
  }
};
```

There must be a way to interact with the `title`, `year` and `genre` variables. They hold all the information about a movie, but how do we access or modify them?

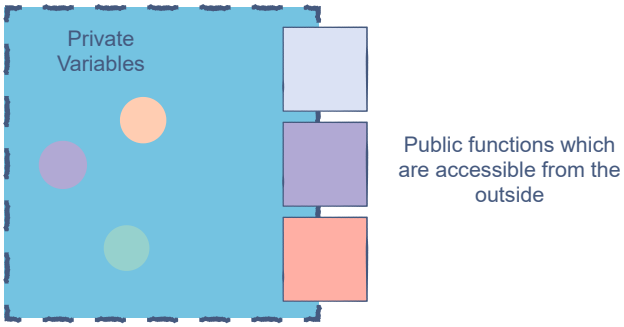
We could create a `getTitle()` method which would return the title to us. Similarly, the other two members could also have corresponding `get` functions.

By observing the emerging pattern, we can make a definitive conclusion. These functions should be part of the class of itself! Let's try it out.

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

class Movie{
  string title;
  int year;
  string genre;

  public:
  Movie(){
    title = "";
    year = -1;
    genre = "";
  }

  Movie(string t, int y, string g){
    title = t;
    year = y;
    genre = g;
  }

  string getTitle(){
    return title;
  }
  void setTitle(string t){
    title = t;
  }

  int getYear(){
    return year;
  }
  void setYear(int y){
    year = y;
  }

  string getGenre(){
    return genre;
  }
  void setGenre(string g){
    genre = g;
  }

  void printDetails(){
    cout << "Title: " << title << endl;
    cout << "Year: " << year << endl;
    cout << "Genre: " << genre << endl;
  }
};

int main() {
  Movie m("The Lion King", 1994, "Adventure");
  m.printDetails();
```

```
  cout << "---" << endl;
  m.setTitle("Forrest Gump");

  cout << "New title: " << m.getTitle() << endl;
}
```

We have now provided an interface of public methods to interact with the `Movie` class. Our private variables cannot be accessed directly from the outside, but we have provided read and write functions which allow access those variables.



This, in essence, is **data encapsulation**.

Advantages of Encapsulation

- Classes are easier to change and maintain.
- We can specify which data member we want to keep hidden or accessible.
- We decide which variables have read/write privileges (increases flexibility).

In the next lesson, we'll discuss the second component of data hiding: **abstraction**.

Abstraction in Classes

This lesson will define what abstraction is and how it relates to data hiding.

WE'LL COVER THE FOLLOWING

Abstraction is the second component of data hiding in OOP. It is an extension of encapsulation and further simplifies the structure of programs.

What is Abstraction?

> Abstraction focuses on revealing only the relevant parts of the application while keeping the inner implementation hidden.

Users will perform actions and expect the application to respond accordingly. They will not be concerned with how the response in generated. Only the final outcome is relevant.

There are countless real life examples which follow the rules of abstraction. Take the *Volume* button on a television remote. With a click of a button, we request the T.V. to increase its volume. Let's say the button calls the `volumeUp()` function. The T.V. responds by producing a sound larger than before. How the inner circuitry of the T.V. implements this is

oblivious to us, yet we know the

exposed function needed to interact
with the T.V.'s volume.

Another instance of abstraction is our daily use of vehicles. To our general knowledge, the race peddle tells the car to consume fuel and increase its speed. We do not need to understand the mechanical process happening inside.

Class Abstraction

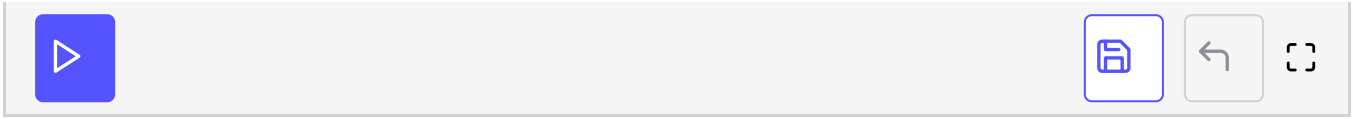
So, let's put all this theory into practice. In the code below, we have a basic class for a circle:

```
class Circle{
  double radius;
  double pi;
};
```

It has two variables, `radius` and `pi`. Now let's add the constructor and functions for the area and perimeter:

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

class Circle{
  double radius;
  double pi;

  public:
  Circle (){
    radius = 0;
    pi = 3.142;
  }
  Circle(double r){
    radius = r;
    pi = 3.142;
  }

  double area(){
    return pi * radius * radius;
  }

  double perimeter(){
    return 2 * pi * radius;
  }
};
```

```
int main() {
  Circle c(5);
  cout << "Area: " << c.area() << endl;
  cout << "Perimeter: " << c.perimeter() << endl;
}
```



As you can see, we only need to define the radius of the circle in the constructor. After that, the `area()` and `perimeter()` functions are available to us. This interface is part of encapsulation. However, the interesting part is what happens next.

All we have to do is call the functions and voila, we get the area and perimeter as we desire. Users would not know what computations were performed inside the function. Even the `pi` constant will be hidden to them. Only the input and the output matter. This is the process of **abstraction using classes**.

In the next lesson, we will explain the second technique of abstraction: **abstraction using header files**.

Abstraction in Header Files

The second strategy for implementing abstraction is creating header files. Find out more below!

WE'LL COVER THE FOLLOWING

In the last lesson, we created the `Circle` class which had two functions, `area()` and `perimeter()`. At that point, all the code was in a single file. Since our goal is to hide the unnecessary details from the users, we can divide the code into different files. This is where **header files** come into play.

Creating Header Files

Let's take a look at the `Circle` class we created in the previous lesson:

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

class Circle{
  double radius;
  double pi;

  public:
  Circle (){
    radius = 0;
    pi = 3.142;
  }
  Circle(double r){
    radius = r;
    pi = 3.142;
  }

  double area(){
    return pi * radius * radius;
  }

  double perimeter(){
    return 2 * pi * radius;
  }
};

int main() {
```

```
  Circle c(5);
  cout << "Area: " << c.area() << endl;

  cout << "Perimeter: " << c.perimeter() << endl;
}
```

To hide our class, we will follow a few steps. The first step is to create a header file. This file will only contain the declaration of the class and its members. A header file always has the `.h` extension:

main.cpp

Circle.h

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

class Circle{
  double radius;
  double pi;

  public:
  Circle (){
    radius = 0;
    pi = 3.142;
  }
  Circle(double r){
    radius = r;
    pi = 3.142;
  }

  double area(){
    return pi * radius * radius;
  }

  double perimeter(){
    return 2 * pi * radius;
  }
};

int main() {
  Circle c(5);
  cout << "Area: " << c.area() << endl;
  cout << "Perimeter: " << c.perimeter() << endl;
}
```

As you can see, the header file isn't very useful if the complete implementation is still visible in our `main` file. Therefore, the second step is to move all the implementation to a separate file. Let's call this `Circle.cpp`.

In this file, we must **include the header file**. The `include` command should already be familiar to you. We use it all the time to include libraries like

`iostream` or `vector`. We can include header files in the same way!

Since we're implementing all the methods of our `Circle` class in `Circle.cpp`, we must mention the name of the class along with the *scope resolution operator* (`::`). Let's do this now:

main.cpp

Circle.cpp

Circle.h

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

class Circle{
  double radius;
  double pi;

  public:
  Circle (){
    radius = 0;
    pi = 3.142;
  }
  Circle(double r){
    radius = r;
    pi = 3.142;
  }

  double area(){
    return pi * radius * radius;
  }

  double perimeter(){
    return 2 * pi * radius;
  }
};

int main() {
  Circle c(5);
  cout << "Area: " << c.area() << endl;
  cout << "Perimeter: " << c.perimeter() << endl;
}
```

At this point, everything is in place. We can remove the `Circle` class from our `main.cpp`. All we have to do is include the header file and the compiler will handle the rest:

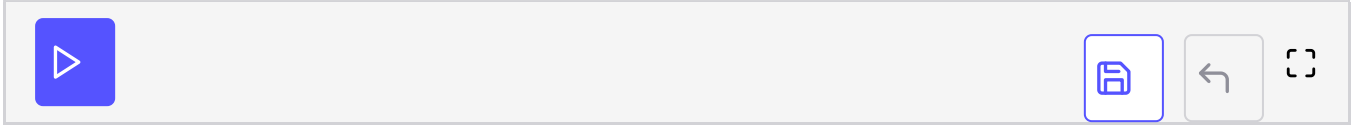
main.cpp

Circle.h

Circle.cpp

```
#include <iostream>
#include "./Circle.h"

using namespace std;


int main() {
  Circle c(5);
  cout << "Area: " << c.area() << endl;
  cout << "Perimeter: " << c.perimeter() << endl;
```

```
}
```



In the header file, we have two commands:

```
#ifndef CIRCLE_H
#define CIRCLE_H
```

These commands tell the compiler that this header file can be used in multiple places. The `#ifndef` command ends with `#endif`.

What we're seeing now is complete abstraction. None of the implementation is visible to users. If they need to know what methods are available in the `Circle` class, they can simply refer to the header file.

This ends our discussion on data hiding. Combining encapsulation and abstraction gives us a simple and reusable interface for our program. In the next section, we'll explore the concept of inheritance.

What is Inheritance?

In this lesson, we'll be learning about the core concept of the object-oriented paradigm, i.e., Inheritance and why there is a need for it?

WE'LL COVER THE FOLLOWING

Why do We Need Inheritance?

In the classes chapter, we've covered the `HAS-A` relationship. We know a class `HAS-A` data members and member functions. Now, we want the data members, and member functions of the class are accessible from other classes. So, the capability of a class to derive properties and characteristics from another class is called `Inheritance`. In inheritance, we have `IS-A` relationship between classes e.g a *car* is a *vehicle* and a *ship* is a *vehicle*.

Let's take the example of `Vehicle` here.

Vehicle Class

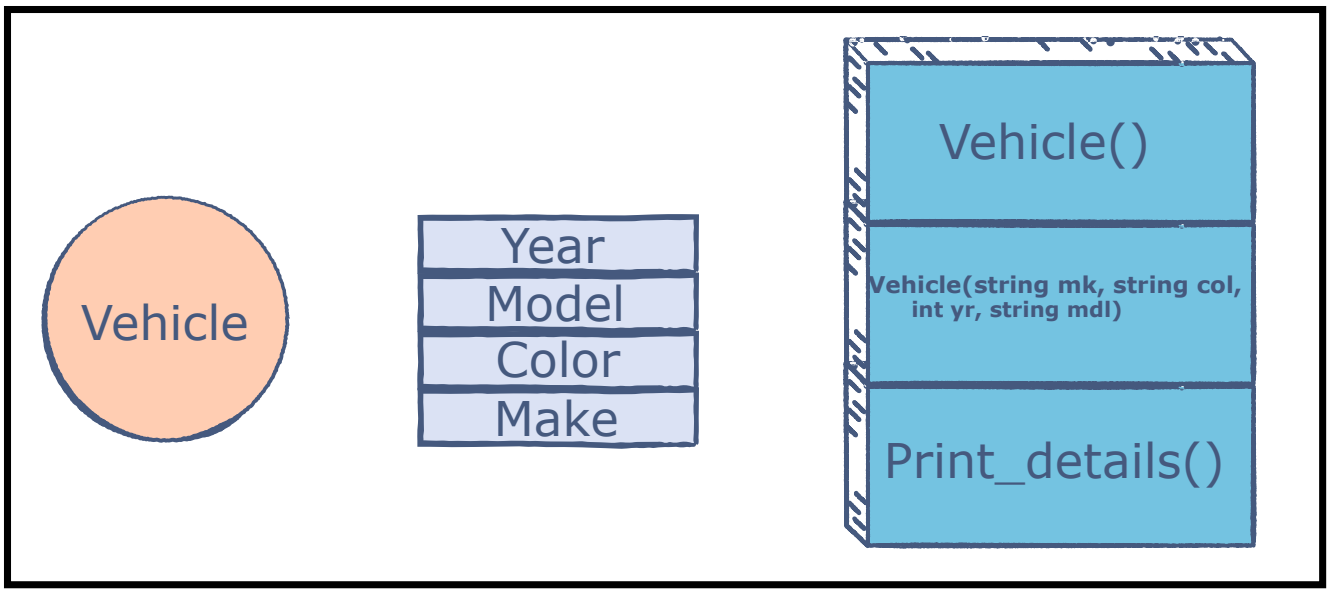
In a Vehicle class, we have many data members like *Make*, *Color*, *Year* and *Model*. A `Vehicle` *HAS-A* Model, Year, Color and Make.

Implementation of `Vehicle` Class

Let's look at the implementation of `Vehicle` class:

```
class Vehicle{
  protected:
  string Make;
  string Color;
  int Year;
  string Model;

  public:
  Vehicle(){
    Make = "";
    Color = "";
    Year = 0;
    Model = "";
  }

  Vehicle(string mk, string col, int yr, string mdl){
    Make = mk;
    Color = col;
    Year = yr;
    Model = mdl;
  }

  void print_details(){
    cout << "Manufacturer: " << Make << endl;
    cout << "Color: " << Color << endl;
    cout << "Year: " << Year << endl;
    cout << "Model: " << Model << endl;
  }
};

int main(){
  Vehicle v("Ford Australia", "Yellow", 2008, "Falcon");
  v.print_details();
}
```

The following illustration depicts the structure of the `Vehicle` class:



These attributes are also attributes of all *Cars*, *Ships* and *Airplanes* but every type of `vehicle` has some attributes that are different from other types of vehicles, as we will see in detail.

Cars Class

The implementation of a `Cars` class needs the same data members and member functions of `Vehicle` class but we have to include them in the `Cars` class. Cars do have a trunk and every trunk has a capacity to store things upto some limit.

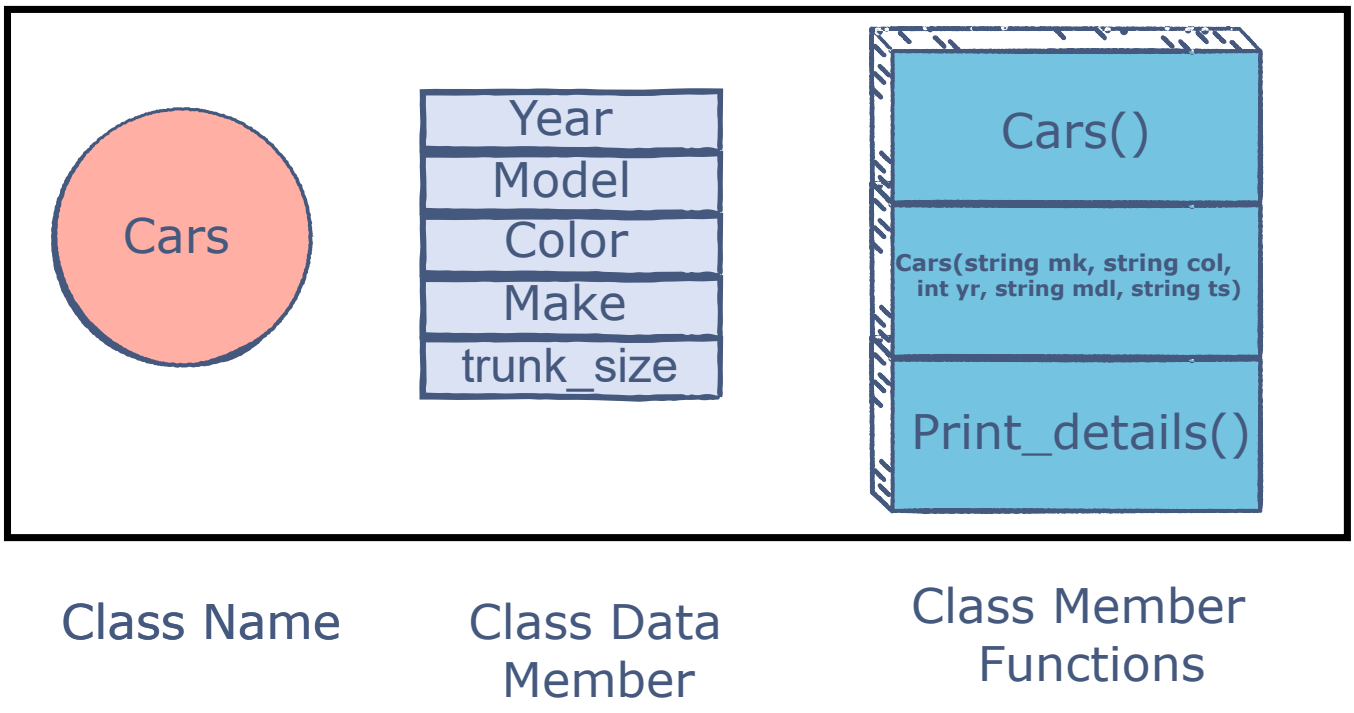
Implementation of `Cars` Class

Let's look at the implementation of the `Cars` class:

```
class Cars{
  string Make;
  string Color;
  int Year;
  string Model;
  string trunk_size;


  public:
  Cars(){
    Make = "";
    Color = "";
    Year = 0;
```

```
    Model = "";
    trunk_size = "";
  }

  Cars(string mk, string col, int yr, string mdl, string ts){
    Make = mk;
    Color = col;
    Year = yr;
    Model = mdl;
    trunk_size = ts;
  }

  void print_details(){
    cout << "Manufacturer: " << Make << endl;
    cout << "Color: " << Color << endl;
    cout << "Year: " << Year << endl;
    cout << "Model: " << Model << endl;
    cout << "Trunk size: " << trunk_size << endl;
  }
};

int main(){
  Cars car("Chevrolet", "Black", 2010, "Camaro", "9.1 cubic feet");
  car.print_details();
}
```

The following illustration depicts the structure of the `Cars` class:



Ships Class

The implementation of a `Ships` class needs the same data members and

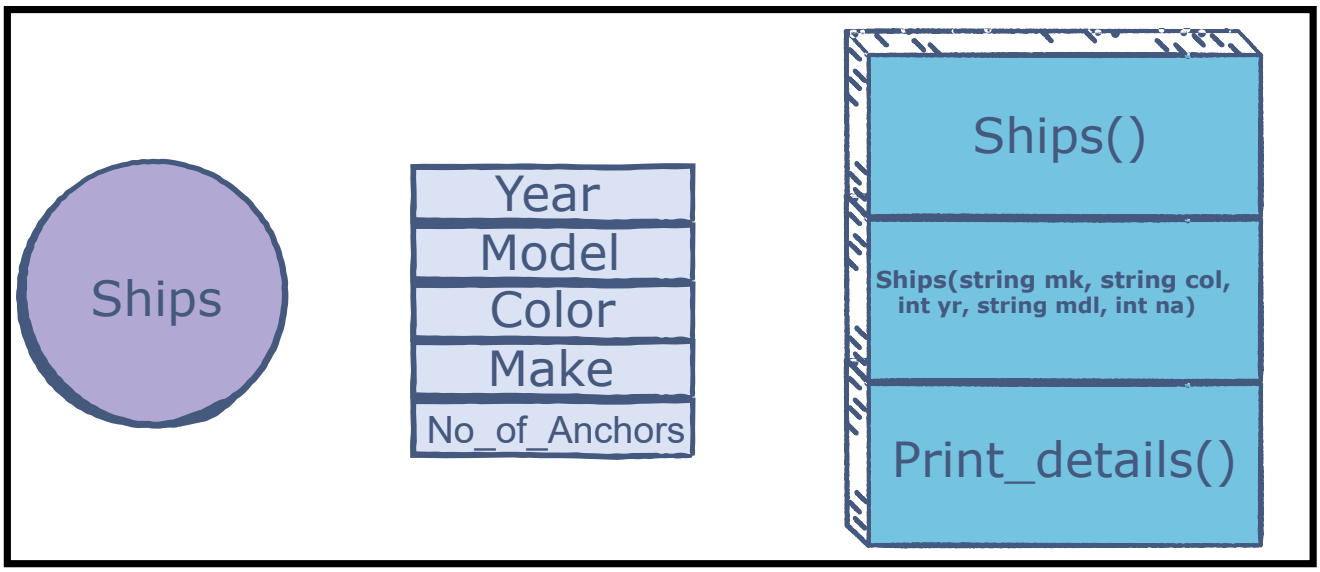
member functions of `Vehicle` class but we have to include them in the `Ships` class. Ships do have anchors and they vary in numbers.

Implementation of `Ships` Class

Let's look at the implementation of the `Ships` class:

```
class Ships{
  string Make;
  string Color;
  int Year;
  string Model;
  int Number_of_Anchors;


  public:
  Ships(){
    Make = "";
    Color = "";
    Year = 0;
    Model = "";
    Number_of_Anchors = 0;
  }

  Ships(string mk, string col, int yr, string mdl, int na){
    Make = mk;
    Color = col;
    Year = yr;
    Model = mdl;
    Number_of_Anchors = na;
  }

  void print_details(){
    cout << "Manufacturer: " << Make << endl;
    cout << "Color: " << Color << endl;
    cout << "Year: " << Year << endl;
    cout << "Model: " << Model << endl;
    cout << "Number of Anchors: " << Number_of_Anchors << endl;
  }
};

int main(){
  Ships ship("Harland and Wolff, Belfast", "Black and whilte",
             1912, "RMS Titanic", 3);
  ship.print_details();
}
```

The following illustration depicts the structure of the `Ships` class:



Class Name | Class Data Member | Class Member Functions

In the declared classes for different types of vehicles (`Cars` and `Ships`), we have many repetitive attributes which should be in one base class and should be inherited in the derived classes.

In the next lesson, we'll be learning about base class and derived class.

Base Class and Derived Class

In this lesson, we'll be learning about how a base class attributes are available to the derived classes and how to define base and a derived class.

WE'LL COVER THE FOLLOWING

In the last lesson, we have seen that `Vehicle` class attributes are shared by the other two classes(`Cars` and `Ships`).

Vehicle as a Base Class

We can consider the `Vehicle` class as a `base` class as it has common attributes.

Derived Classes

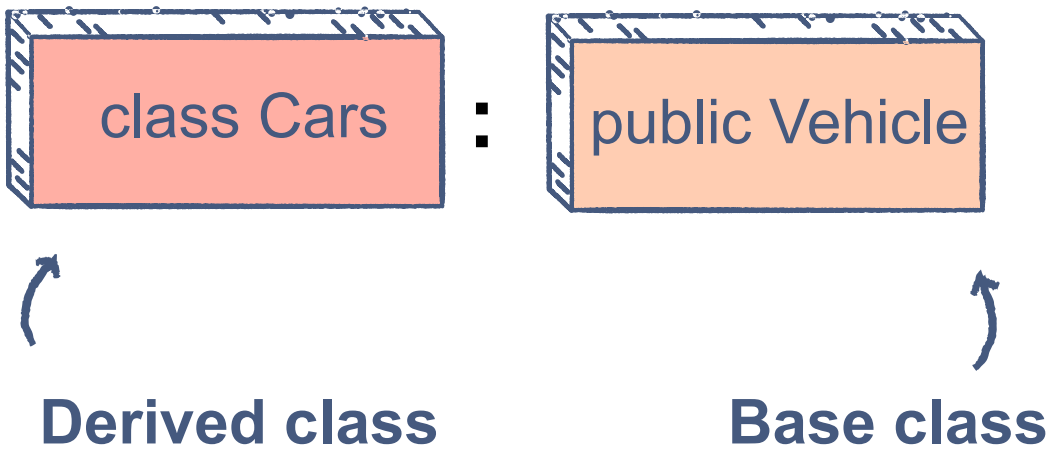
Cars and *Ships* are considered as `derived` classes as they're inheriting properties from `vehicle` class.

Modes of Inheritance

There are three modes of class inheritance: `public`, `private` and `protected`. The basic syntax for inheritance is given below:

```
class derivedClassName : modeOfInheritance baseClassName
```

We use the keyword **public** to implement **public** inheritance.



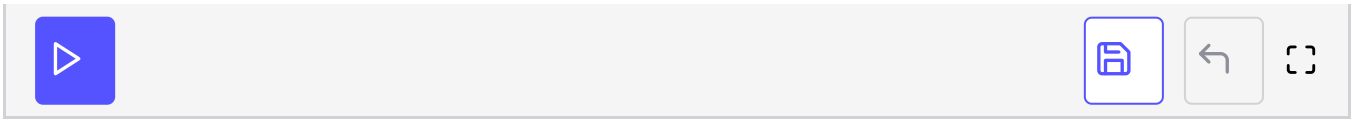
Now, the class `Cars` have access to the public members of a base class `vehicle` and the protected data is inherited as protected data, and the private data is not inherited, but it can be accessed directly by the public member functions of the class.

Public Inheritance

We are updating our `Cars` and `Ships` class so that *Make*, *Color*, *Year*, *Model* and the function `void print_details()` can be inherited from base class `Vehicle`. So, we have removed these variables and function from the derived classes. The following example shows the classes `Cars` and `Ships` that inherits **publicly** from the base class `Vehicle`.

```
class Vehicle{
  protected:
  string Make;
  string Color;
  int Year;
  string Model;

  public:
  Vehicle(){
    Make = "";
    Color = "";
    Year = 0;
    Model = "";
  }

  Vehicle(string mk, string col, int yr, string mdl){
    Make = mk;
    Color = col;
    Year = yr;
    Model = mdl;
  }

  void print_details(){
```

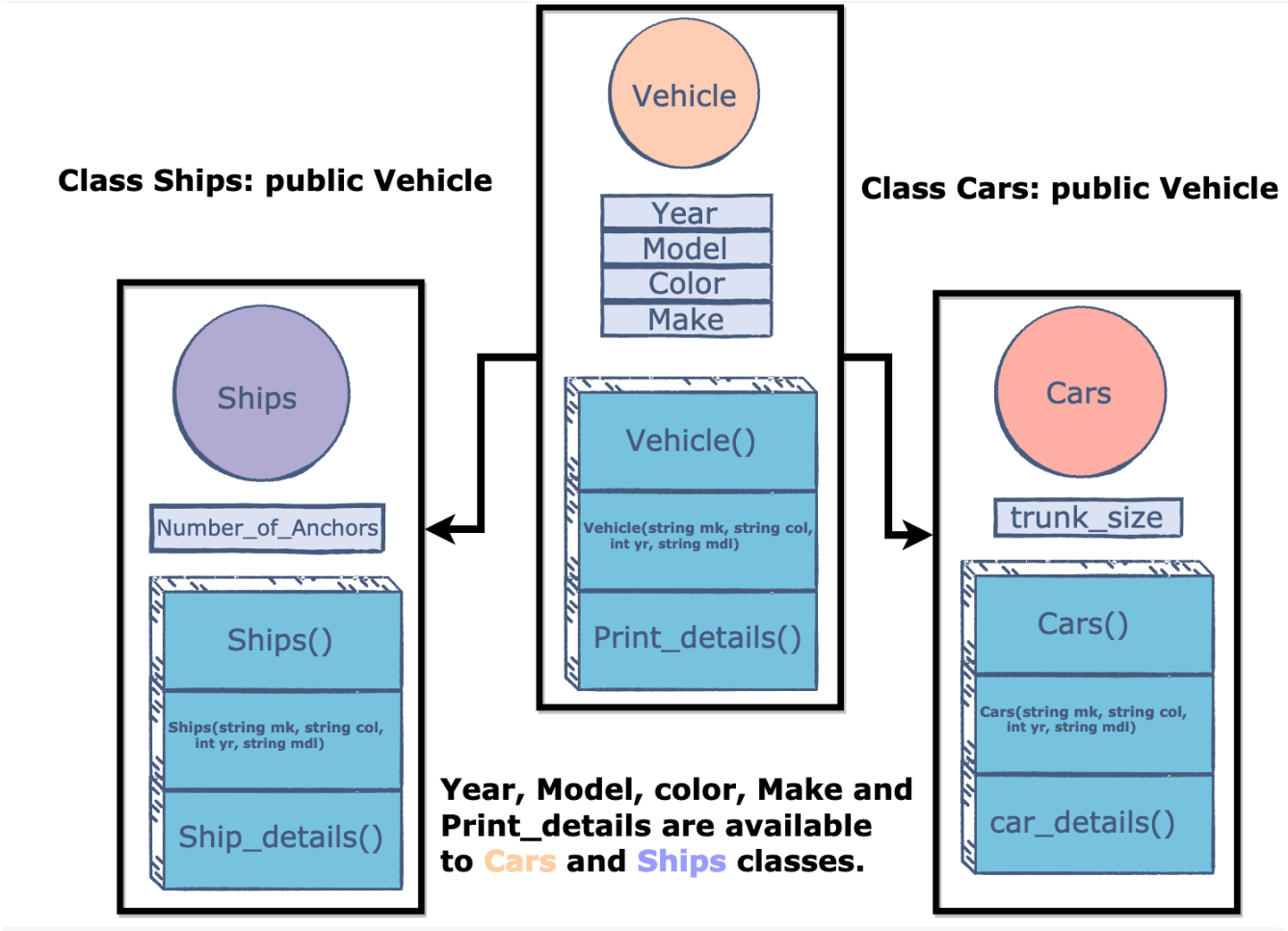
```
    cout << "Manufacturer: " << Make << endl;
    cout << "Color: " << Color << endl;
    cout << "Year: " << Year << endl;
    cout << "Model: " << Model << endl;
  }
};

class Cars: public Vehicle{
  string trunk_size;

  public:
  Cars(){
    trunk_size = "";
  }

  Cars(string mk, string col, int yr, string mdl, string ts)
    :Vehicle(mk, col, yr, mdl){
    trunk_size = ts;
  }

  void car_details(){
    print_details();
    cout << "Trunk size: " << trunk_size << endl;
  }
};

class Ships: public Vehicle{
  int Number_of_Anchors;

  public:
  Ships(){
    Number_of_Anchors = 0;
  }

  Ships(string mk, string col, int yr, string mdl, int na)
  :Vehicle(mk, col, yr, mdl){
    Number_of_Anchors = na;
  }

  void Ship_details(){
    print_details();
    cout << "Number of Anchors: " << Number_of_Anchors << endl;
  }
};

int main(){
  Cars car("Chevrolet", "Black", 2010, "Camaro", "9.1 cubic feet");
  car.car_details();

  cout << endl;

  Ships ship("Harland and Wolff, Belfast", "Black and whilte",
             1912, "RMS Titanic", 3);
  ship.Ship_details();
}
```



The highlighted lines in the above code indicate how to achieve inheritance in

C++ by using `:` and mentioning the mode of inheritance.

The following illustration explains the concept of inheritance in the above code:



Explanation

Now the `Ships` and `Cars` classes have access to public member functions of the base class `Vehicle` as shown in the above illustration. `Protected` and `public` data members are accessible to derived classes.

Now that we have learned about the base and derived classes. So, let's move to the next lesson in which we'll learn about the base class constructors and destructors.

Base Class Constructor and Destructor

In this lesson, we'll learn how constructors and destructors are called in derived and base classes during inheritance.

WE'LL COVER THE FOLLOWING

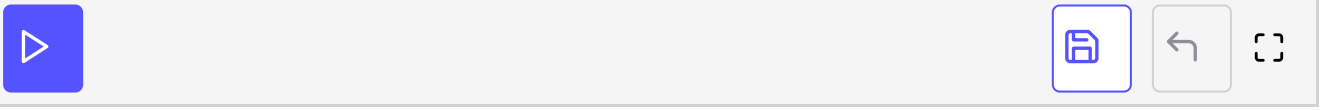
Base Class Constructor

When we make an instance of the `Derived` class without parameters it will first call the default constructor of the `Base` class and then the `Derived` class. In the same way, when we call the parameterized constructor of the derived class, it will first call the parameterized constructor of the `Base` class and then `Derived` class.

The following code explains how this is done:

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

// Base class
class Base {

    public:
    Base(){
      cout << "Base class default constructor!" << endl;
    }
    // Base class's parameterised constructor
    Base(float i) {
        cout << "Base class parameterized constructor" << endl;
    }
};


// Derived class
class Derived : public Base {
    public:
    Derived(){
      cout << "Derived class default constructor!" << endl;
```

```
    }

    // Derived class's parameterised constructor
    Derived(float num): Base(num){
        cout << "Derived class parameterized constructor" << endl;
    }
};

// main function
int main() {
    // creating object of Derived Class
    Derived obj;
    cout << endl;
    Derived obj1(10.2);
}
```

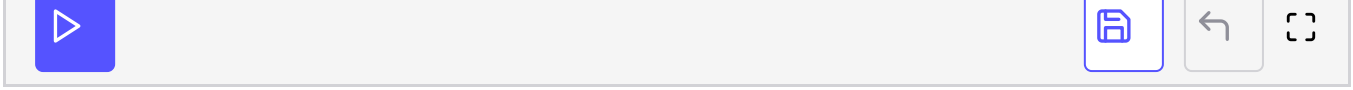


Base Class Destructor

When we make an instance of the `Derived` class it will first call the destructor of the `Derived` class and then the `Base` class.

The following code explains how this is done:

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

// Base class
class Base {

    public:
    ~Base(){
      cout << endl << "Base class Destructor!" ;
    }
};


// Derived class
class Derived : public Base {
    public:

    ~Derived(){
      cout << endl << "Derived class Destructor!" ;
    }
};

// main function
int main() {
    // creating object of Derived Class
    Derived obj;

}
```



In the next lesson, we'll be learning about the `public`, `protected` and `private` inheritance.

Modes of Inheritance

In this lesson, we'll learn about how Public, Private and Protected inheritance is done in C++.

WE'LL COVER THE FOLLOWING

You are already familiar with Access Modifiers from the Classes chapter. By using these specifiers, we limit the access of the data members and member functions to the other *classes* and *main*.

`private` Mode of Inheritance

By using `private` inheritance, the *private* data members and member functions of the base class are inaccessible in the derived class and in `main`. *Protected* and *Public* members of the base class are accessible to the derived class but not in `main`.

Let's look at the implementation using `private` inheritance:

```
class Vehicle{

  string Make;
  string Color;
  int Year;

  protected:
  string Model;

  public:
  Vehicle(){
    Make = "";
    Color = "";
    Year = 0;
    Model = "";
```

```
  }

  Vehicle(string mk, string col, int yr, string mdl){
    Make = mk;
    Color = col;
    Year = yr;
    Model = mdl;
  }

  void print_details(){
    cout << "Manufacturer: " << Make << endl;
    cout << "Color: " << Color << endl;
    cout << "Year: " << Year << endl;
  }
};

class Cars: private Vehicle{
  string trunk_size;

  public:
  Cars(){
    trunk_size = "";
  }

  Cars(string mk, string col, int yr, string mdl, string ts)
    :Vehicle(mk, col, yr, mdl){
    trunk_size = ts;
  }

  void car_details(){
    print_details();
    cout << "Trunk size: " << trunk_size << endl;
    cout << "Model: " << Model << endl;  // Model is protected and
    // is accessible in derived class
  }
};

int main(){
  Cars car("Chevrolet", "Black", 2010, "Camaro", "9.1 cubic feet");
  // car.Year = 2000;     // this will give error as Year is private
  // car.Model = "Accord";   // this will give error as Model is protected

  car.car_details();
  //car.print_details();   // public functions of base class are inaccessible in main
}
```

`protected` Mode of Inheritance

By using `protected` inheritance, the *private* members of the base class are inaccessible in the derived class and in `main`. *Protected* and *Public* members of the base class are accessible to the derived class but not in `main`.

Let's take an example of `protected` inheritance:

```
class Vehicle{

  string Make;
  string Color;
  int Year;

  protected:
  string Model;

  public:
  Vehicle(){
    Make = "";
    Color = "";
    Year = 0;
    Model = "";
  }

  Vehicle(string mk, string col, int yr, string mdl){
    Make = mk;
    Color = col;
    Year = yr;
    Model = mdl;
  }

  void print_details(){
    cout << "Manufacturer: " << Make << endl;
    cout << "Color: " << Color << endl;
    cout << "Year: " << Year << endl;
  }
};

class Cars: protected Vehicle{
  string trunk_size;

  public:
  Cars(){
    trunk_size = "";
  }

  Cars(string mk, string col, int yr, string mdl, string ts)
    :Vehicle(mk, col, yr, mdl){
    trunk_size = ts;
  }

  void car_details(){
    print_details();
    cout << "Trunk size: " << trunk_size << endl;
    cout << "Model: " << Model << endl;  // Model is protected and
    // is accessible in derived class
  }
};

int main(){
  Cars car("Chevrolet", "Black", 2010, "Camaro", "9.1 cubic feet");
  // car.Year = 2000;     // this will give error as Year is private
  // car.Model = "Accord";   // this will give error as Model is protected

  car.car_details();
```

```
  //car.print_details();   // public functions of base class are inaccessible in main
}
```

`public` Mode of Inheritance

By using `public` inheritance, the *private* members of the base class are inaccessible in the derived class and in `main`. *Protected* members of the base class are accessible to the derived class but not in `main`. *Public* members of the base class are accessible to the derived class and in `main`.

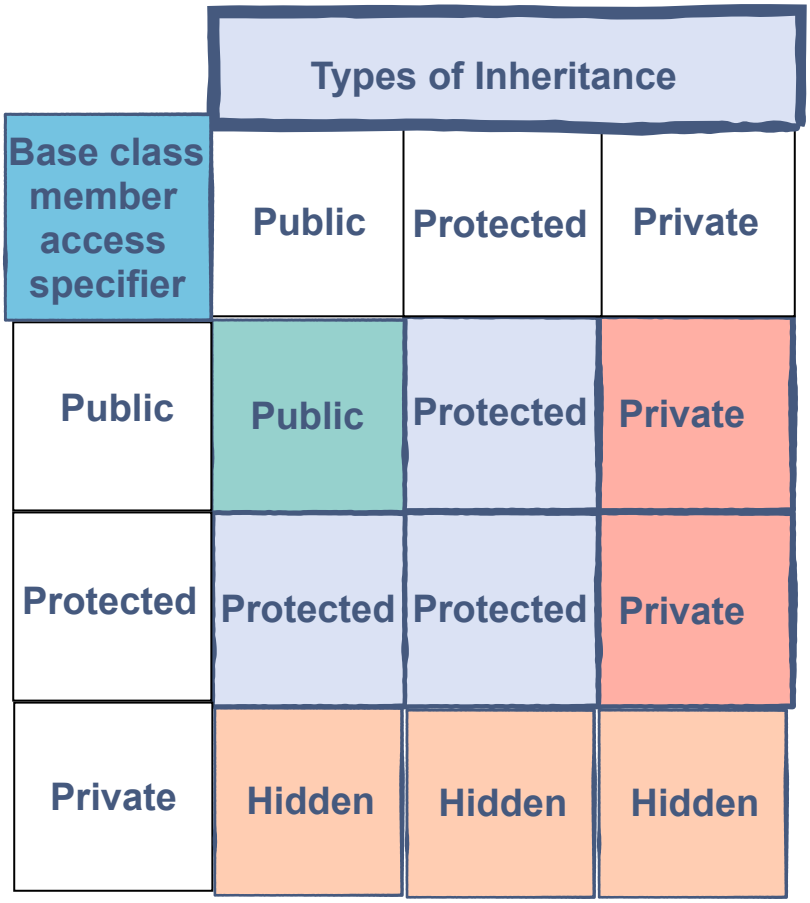
Let's look at the implementation using `public` inheritance:

```
class Vehicle{

  string Make;
  string Color;
  int Year;

  protected:
  string Model;

  public:
  Vehicle(){
    Make = "";
    Color = "";
    Year = 0;
    Model = "";
  }

  Vehicle(string mk, string col, int yr, string mdl){
    Make = mk;
    Color = col;
    Year = yr;
    Model = mdl;
  }

  void print_details(){
    cout << "Manufacturer: " << Make << endl;
    cout << "Color: " << Color << endl;
    cout << "Year: " << Year << endl;
  }
};

class Cars: public Vehicle{
  string trunk_size;

  public:
  Cars(){
    trunk_size = "";
  }

  Cars(string mk, string col, int yr, string mdl, string ts)
```

```
    :Vehicle(mk, col, yr, mdl){
    trunk_size = ts;
  }


  void car_details(){
    cout << "Trunk size: " << trunk_size << endl;
    cout << "Model: " << Model << endl;  // Model is protected and
    // is accessible in derived class
  }
};

int main(){
  Cars car("Chevrolet", "Black", 2010, "Camaro", "9.1 cubic feet");
  // car.Year = 2000;     // this will give error as Year is private
   //car.Model = "Accord";   // this will give error as Model is protected

  car.car_details();
  car.print_details();   // public functions of base class are accessible in main
}
```

Modes of Inheritance in Base Class

The given table depicts the access of members of our base class when we use specific modifiers and its behavior.



In the next lesson, we'll be learning about multiple inheritance which is a core concept of inheritance.

Types of Inheritance

In this lesson, we'll learn about the types of inheritance which includes multiple inheritance and multilevel inheritance.

WE'LL COVER THE FOLLOWING

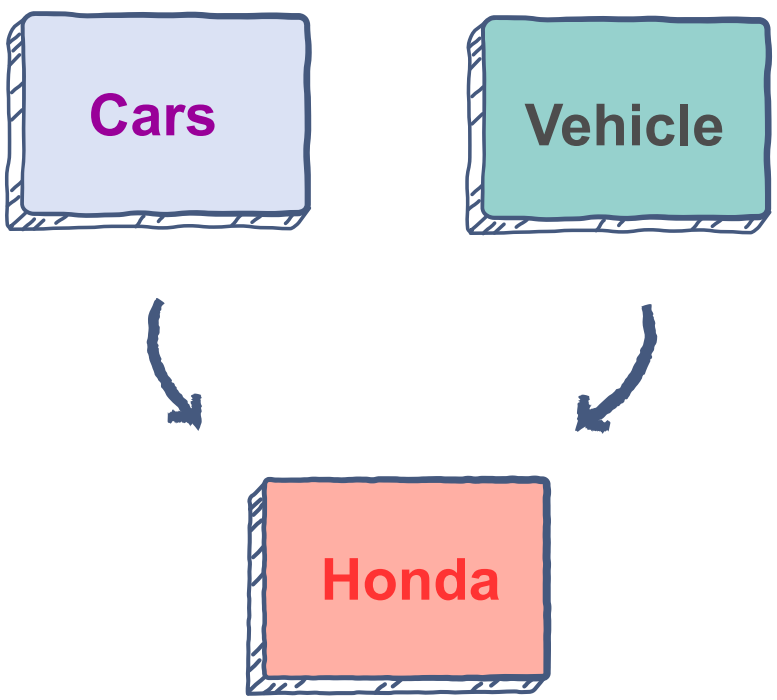
Multiple Inheritance

We can inherit the base class attributes to the derived class if we want derived class to have access data members and member functions of the base class. But to inherit multiple classes data members and member functions to the derived, the concept of *multiple inheritance* comes in. We can inherit multiple classes as base classes separated by `,`

```
class Derived : public Base1 , public Base2 , ...
```

Example

Let's take the example of `Vehicle` and `Cars` classes which acts as the base classes of the `Honda` class:

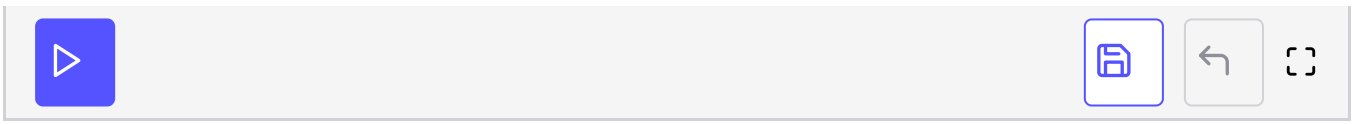


Implementation

Implementation of the `Honda` class is given below:

```
class Vehicle{
  protected:
  string Make;
  string Color;
  int Year;
  string Model;

  public:
  Vehicle(){
    Make = "";
    Color = "";
    Year = 0;
    Model = "";
  }

  Vehicle(string mk, string col, int yr, string mdl){
    Make = mk;
    Color = col;
    Year = yr;
    Model = mdl;
  }

  void print_details(){
```

```
    cout << "Manufacturer:   " << Make << endl;
    cout << "Color: " << Color << endl;
    cout << "Year: " << Year << endl;
    cout << "Model: " << Model << endl;
  }
};

class Cars{
  string trunk_size;

  public:
  Cars(){
    trunk_size = "";
  }

  Cars(string ts){
    trunk_size = ts;
  }

  void car_details(){
    cout << "Trunk size: " << trunk_size << endl;
  }
};

class Honda: public Vehicle, public Cars{
  int top_speed;

  public:
  Honda(){
    top_speed = 0;
  }

  Honda(string mk, string col, int yr, string mdl, string na, int ts)
  :Vehicle(mk, col, yr, mdl), Cars(na){
    top_speed = ts;
  }

  void Honda_details(){
    print_details();
    car_details();
    cout << "Top speed of the car: " << top_speed << endl;
  }
};

int main(){
  Honda car("Honda", "Black", 2006, "Accord", "14.7 cubic feet", 260);
  car.Honda_details();
}
```



Now, the `Honda` class object has access to all member functions of `Cars` and `Vehicle` classes as they're now base classes of `Honda` class. The highlighted lines in the code indicate how multiple inheritance is achieved.

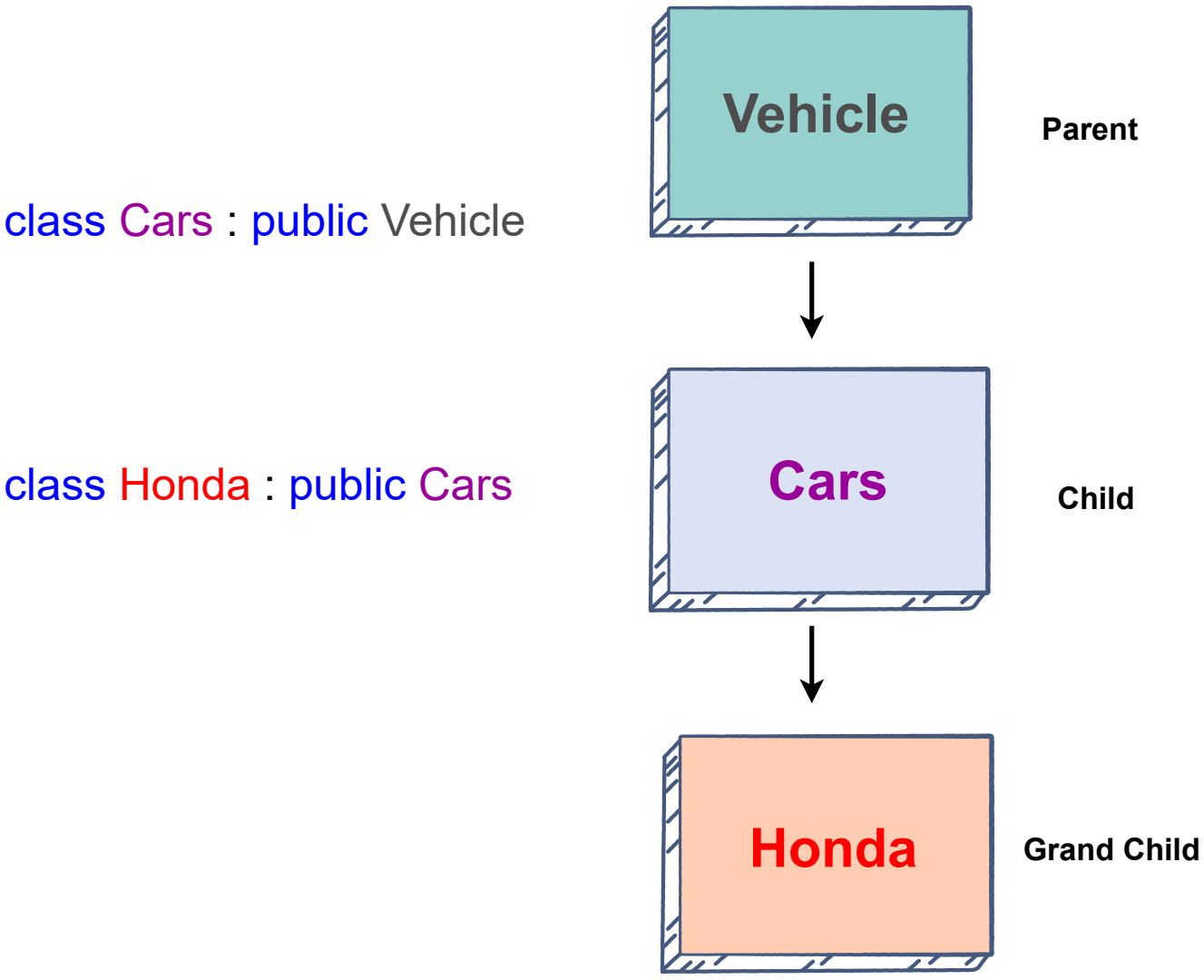
Multilevel Inheritance

If we want to inherit data members and member functions of the base class which is already inherited from another class, the concept of `multilevel inheritance` comes in. This contains a more hierarchical approach.

```
class parent

class child : public parent

class grandChild : public child
```

Example

Let's take the example of `Vehicle` class which acts as a parent to `Cars` class. Now `Cars` class acts as a parent to `Honda` class.



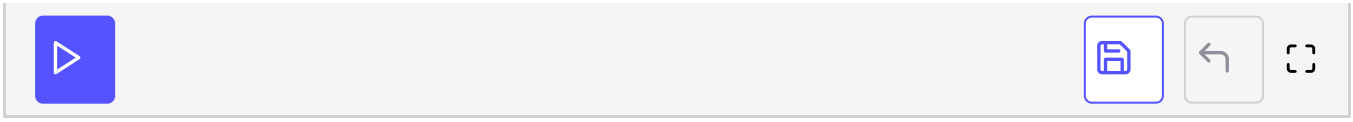
Multilevel Inheritance

Implementation

Implementation of the `Honda` class is given below:

```
class Vehicle {
  protected:
  string Make;
  string Color;
  int Year;
  string Model;

  public:
  Vehicle(){
    Make = "";
    Color = "";
    Year = 0;
    Model = "";
  }

  Vehicle(string mk, string col, int yr, string mdl){
    Make = mk;
    Color = col;
    Year = yr;
    Model = mdl;
  }

  void print_details(){
    cout << "Manufacturer: " << Make << endl;
    cout << "Color: " << Color << endl;
    cout << "Year: " << Year << endl;
    cout << "Model: " << Model << endl;
  }
};

class Cars: public Vehicle{
  string trunk_size;

  public:
  Cars(){
    trunk_size = "";
  }

  Cars(string mk, string col, int yr, string mdl, string ts)
    :Vehicle(mk, col, yr, mdl){
      trunk_size = ts;
    }

  void car_details(){
    cout << "Trunk size: " << trunk_size << endl;
  }
};

class Honda: public Cars{
  int top_speed;

  public:
```

```
  Honda(){
    top_speed = 0;
  }

  Honda(string mk, string col, int yr, string mdl, string na, int ts)
    :Cars(mk, col, yr, mdl, na){
      top_speed = ts;
    }

  void Honda_details(){
    print_details();
    car_details();
    cout << "Top speed of the car: " << top_speed << endl;
  }
};

int main(){
  Honda car("Honda", "Black", 2006, "Accord", "14.7 cubic feet", 260);
  car.Honda_details();
}
```



Now, `Honda` class object has access to all member functions of `Cars` class and the `Cars` class has access to all members functions of the `Vehicle` class as they're now base classes of `Honda` class. The highlighted lines in the code indicate how multilevel inheritance is achieved.

In the next lesson, we'll learn about the advantages of inheritance.

Advantages of Inheritance

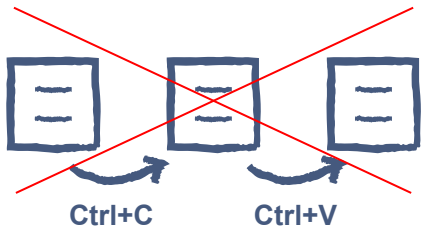
In this lesson, you'll get to know about the advantages of Inheritance.

WE'LL COVER THE FOLLOWING

We have learned that we can implement *inheritance* which will result in avoiding **redundant coding** and will also save the programmer's *time and effort*.

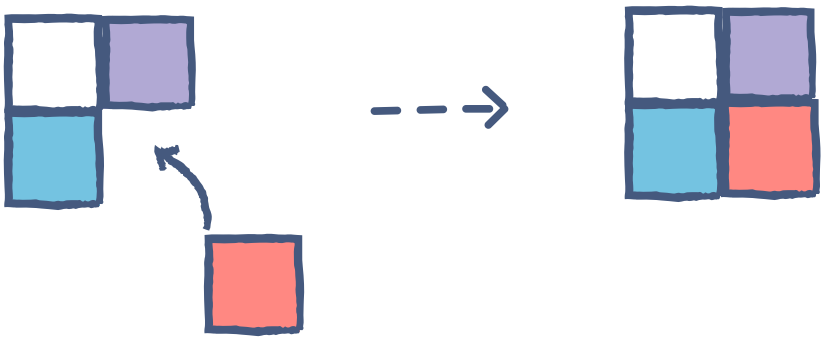
Avoiding Duplication of Code

Considering the previous example, if we have to implement another class for `MoneyMarketAccount` we **don't need to duplicate** the code for the `deposit()` and `withdraw()` methods inside this new Class because we can inherit and use the parent class's methods. In this way, we can avoid the *duplication of code*.



Extensibility

Using `inheritance`, one can extend the base class logic as per the business logic of the derived class. This is an easy way to upgrade or enhance specific parts of a product without changing the core attributes. An existing class can act as a base class to derive a new class having upgraded features.



Data Hiding

The base class can decide to keep some data private so that it cannot be altered by the derived class. This concept i.e. Encapsulation has already been discussed in the previous chapter.

Let's move on to quiz for checking your understanding of inheritance.

What is Polymorphism?

In this lesson, we will be learning about the basics of polymorphism with the implementation details.

WE'LL COVER THE FOLLOWING

> The word **Polymorphism** is a combination of two Greek words, **Poly** means *many* and **Morph** means *forms*.

Definition

When we say *polymorphism* in programming that means something which exhibits many forms or behaviors. So far, we have learned that we can add new data and functions to a class through inheritance. But what about if we want our derived class to inherit a method from the base class and have a different implementation for it? That is when polymorphism comes in, a fundamental concept in OOP paradigm.

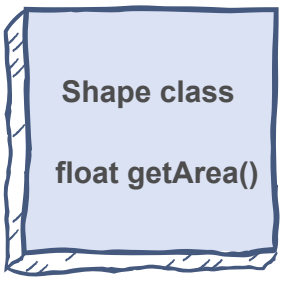
Shape Class

We are considering here the example of **Shape** class, which is base class for many shapes like *Rectangle and Circle*. This class contains a function

getArea() which calculates the area for the *derived* classes.

Implementation

Let's look at the implementation of **Shape** class:

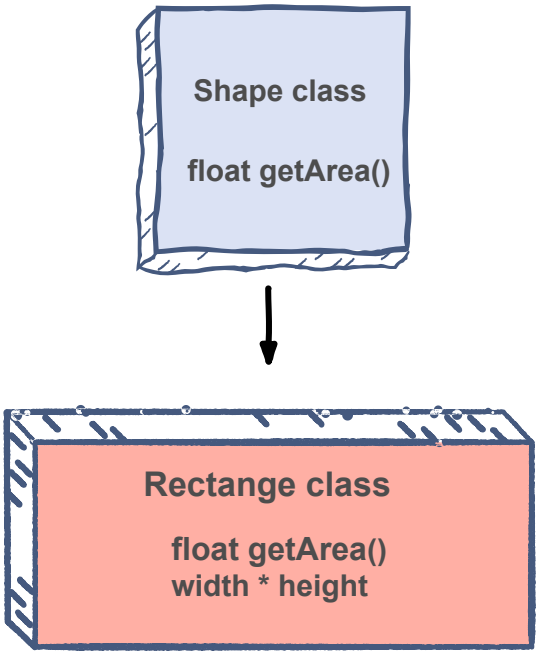


Shape class

```
// A simple Shape interface which provides a method to get the Shape's area
class Shape {
  public:
  float getArea(){}
};
```

Rectangle Class

Consider the **Rectangle** class which is derived from *Shape* class. It has two data members, i.e., *width* and *height* and it returns the *Area* of the rectangle by using **getArea()** function.



Rectangle class

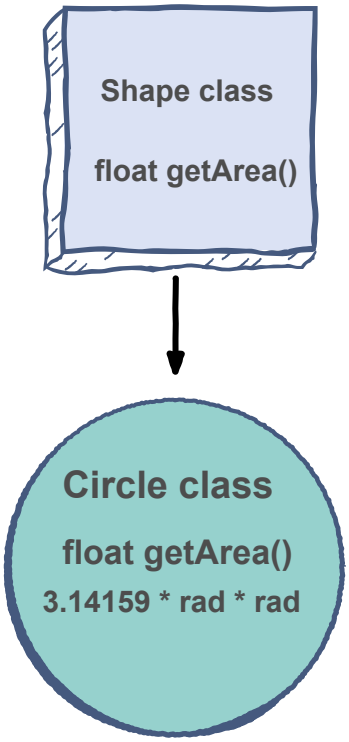
Implementation

Let's look at the implementation of the **Rectangle** class:

```
// A Rectangle is a Shape with a specific width and height
class Rectangle : public Shape {   // derived form Shape class
  private:
  float width;
  float height;

  public:
  Rectangle(float wid, float heigh) {
    width = wid;
    height = heigh;
  }
  float getArea(){
    return width * height;
  }
};
```

Circle Class

Consider the **Circle** class which is derived from *Shape* class. It has one data member, i.e., *radius* and it returns the *Area* of the circle by using **getArea()** function.



Circle class

Implementation

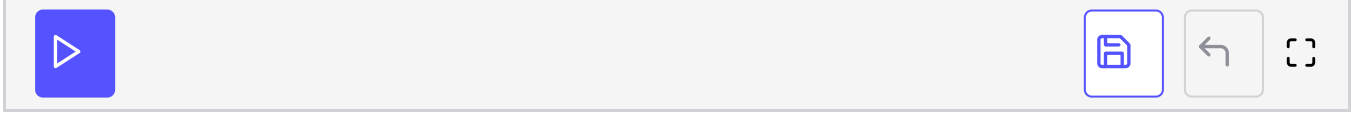
Let's look at the implementation of the **Circle** class:

```
// A Circle is a Shape with a specific radius
class Circle : public Shape {
  private:
  float radius;

  public:
  Circle(float rad){
    radius = rad;
  }
  float getArea(){
    return 3.14159f * radius * radius;
  }
};
```

Now, if we merge all the classes then by calling the **getArea()** function, let's see what happened:

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

// A simple Shape interface which provides a method to get the Shape's area
```

```
class Shape {
  public:
  float getArea(){}

};

// A Rectangle is a Shape with a specific width and height
class Rectangle : public Shape {   // derived form Shape class
  private:
  float width;
  float height;

  public:
  Rectangle(float wid, float heigh) {
    width = wid;
    height = heigh;
  }
  float getArea(){
    return width * height;
  }
};

// A Circle is a Shape with a specific radius
class Circle : public Shape {
  private:
  float radius;

  public:
  Circle(float rad){
    radius = rad;
  }
  float getArea(){
    return 3.14159f * radius * radius;
  }
};

int main() {
  Rectangle r(2, 6);    // Creating Rectangle object

  Shape* shape = &r;   // Referencing Shape class to Rectangle object

  cout << "Calling Rectangle getArea function: " << r.getArea() << endl;      // Calls Rectar
  cout << "Calling Rectangle from shape pointer: " <<  shape->getArea() << endl <<endl; // Ca

  Circle c(5);    // Creating Circle object

  shape = &c;   // Referencing Shape class to Circle object

  cout << "Calling Circle getArea function: " << c.getArea() << endl;
  cout << "Calling Circle from shape pointer: " <<shape->getArea() << endl << endl;
}
```



Explanation of Code

Polymorphism only works with a pointer and reference types, so we have created a **Shape** pointer, and pointed to the *derived* class objects. But due to

multiple existences of the same functions in classes, it will get confused

between which class **getArea()** function it's calling. The derived classes function has a different implementation but the same name and that's why we are not getting the expected output.

In the next lesson, we'll be learning about the fundamental concept of **overriding**.

Overriding

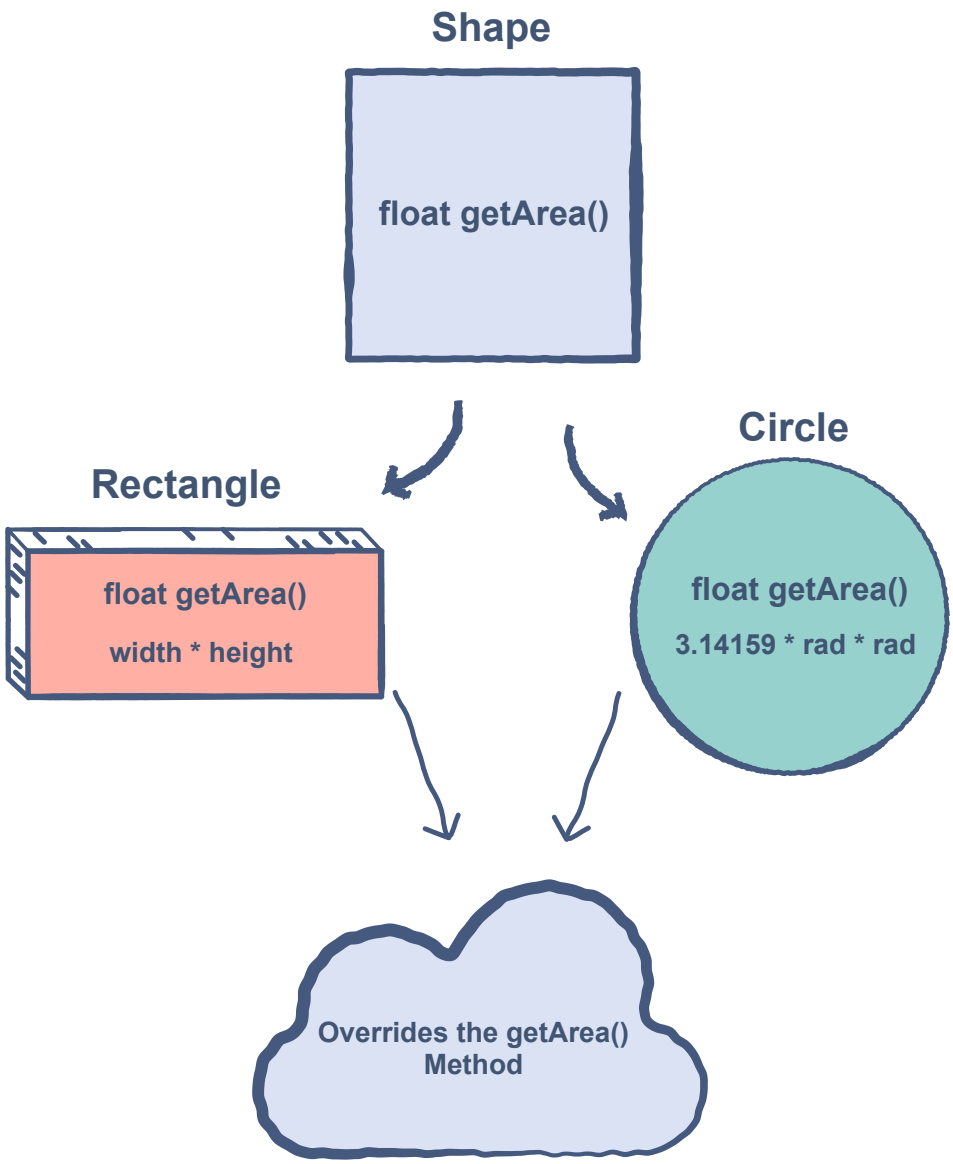
In this lesson, we'll be learning about how overriding is done in C++.

WE'LL COVER THE FOLLOWING

In object-oriented programming when we allow a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes is known as **Function Overriding**.

`getArea()` Overridden Function

As you have already seen the implementation of the function **getArea()** in the previous lesson, which depicts the concept of overriding.

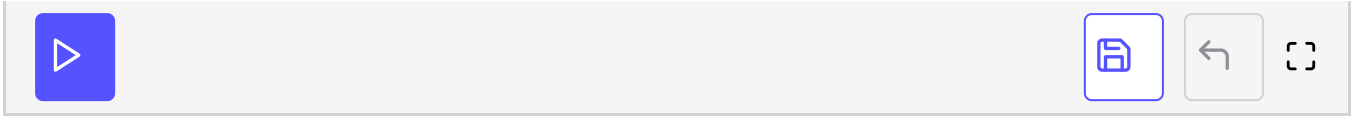


Implementation

Highlighted portion shows where overriding is done. Let's Have a look!

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

// A simple Shape interface which provides a method to get the Shape's area
class Shape {
  public:
  float getArea(){}
};

// A Rectangle is a Shape with a specific width and height
class Rectangle : public Shape {   // derived form Shape class
  private:
  float width;
  float height;

  public:
  Rectangle(float wid, float heigh) {
    width = wid;
    height = heigh;
```

```
  }
  float getArea(){
    return width * height;
  }
};

// A Circle is a Shape with a specific radius
class Circle : public Shape {
  private:
  float radius;

  public:
  Circle(float rad){
    radius = rad;
  }
  float getArea(){
    return 3.14159f * radius * radius;
  }
};

int main() {
  Rectangle r(2, 6);    // Creating Rectangle object

  Shape* shape = &r;   // Referencing Shape class to Rectangle object

  cout << "Calling Rectangle getArea function: " << r.getArea() << endl;      // Calls Rectar
  cout << "Calling Rectangle from shape pointer: " <<  shape->getArea() << endl <<endl; // Ca

  Circle c(5);    // Creating Circle object

  shape = &c;   // Referencing Shape class to Circle object

  cout << "Calling Circle getArea function: " << c.getArea() << endl;
  cout << "Calling Circle from shape pointer: " <<shape->getArea() << endl << endl;
}
```



Advantages of the Method Overriding

Method overriding is very useful in OOP and have many advantages. Some of them are stated below:

- The derived classes can give its own specific implementation to inherited methods without modifying the parent class methods.
- If a child class needs to use the parent class method, it can use it, and the other classes that want to have different implementation can use the overriding feature to make changes.

Key Features of Overriding

Here are some key features of the *Method Overriding:*

- Overriding needs inheritance and there should be at least one derived class.
- Derived class/es must have the same declaration, i.e., name, same parameters and same return type of the function as of the base class.
- The function in derived class/es must have different implementation from each other.
- The method in the base class must need to be overridden in the derived class.

In the next lesson, we'll learn about how to make a function virtual.

Virtual Member Functions

In this lesson, we'll be learning about a very important concept of polymorphism, i.e., Virtual member.

WE'LL COVER THE FOLLOWING

> **Virtual** means existing in appearance but not in reality.

Definition

A `virtual` function is a member function which is declared within the base class and is *overridden* by the derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for the function call. They are mainly used to achieve Runtime polymorphism. Functions are declared with a `virtual` keyword in a base class. The function resolution call is done at run-time.

Why Do We Need a Virtual Function?

Suppose you have a number of objects of different classes like in this case, we have multiple shapes classes. You want to put them all in an array and perform a particular operation on them using the same function call. Given an example, we want to access the same function `getArea()` from multiple child classes.

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


// A simple Shape interface which provides a method to get the Shape's area
class Shape {
  public:
  virtual float getArea(){}
};

// A Rectangle is a Shape with a specific width and height
class Rectangle : public Shape {   // derived form Shape class
  private:
  float width;
  float height;

  public:
  Rectangle(float wid, float heigh) {
    width = wid;
    height = heigh;
  }
  float getArea(){
    return width * height;
  }
};

// A Circle is a Shape with a specific radius
class Circle : public Shape {
  private:
  float radius;

  public:
  Circle(float rad){
    radius = rad;
  }
  float getArea(){
    return 3.14159f * radius * radius;
  }
};

int main() {
  Rectangle r(2, 6);    // Creating Rectangle object
  Shape* shape = &r;   // Referencing Shape class to Rectangle object

  cout << "Calling Rectangle from shape pointer: " <<  shape->getArea() << endl; // Calls sha

  Circle c(5);    // Creating Circle object
  shape = &c;   // Referencing Shape class to Circle object

   cout << "Calling Circle from shape pointer: " <<shape->getArea() << endl;

}
```

Explanation

we have seen earlier when we're trying to print the child class function

`getArea()` by referencing a parent class pointer, it gave us an error. Just by writing the keyword `virtual` we can reference a parent class pointer to child class object.

In the next lesson, we'll be learning about the `Pure Virtual` functions.

Pure Virtual Member Functions

In this lesson, we'll be learning about a very important concept of polymorphism, i.e., Pure Virtual Member Functions.

WE'LL COVER THE FOLLOWING

Abstract Class

As we've seen in the Inheritance chapter, we can only make derived class's objects to access their functions, and we will never want to instantiate objects of a base class, we call it an `abstract` class. Such a class exists only to act as a parent of derived classes that will be used to instantiate objects.

How to Write a Pure Virtual Function?

It may also provide an interface for the class hierarchy by placing at least one pure virtual function in the base class. A pure virtual function is one with the expression `=0` added to the declaration.

`=0` Sign

The equal sign `=` here has nothing to do with the assignment, the value 0 is not assigned to anything. The `=0` syntax is simply how we tell the compiler that a virtual function will be pure.

Overriding Virtual Function

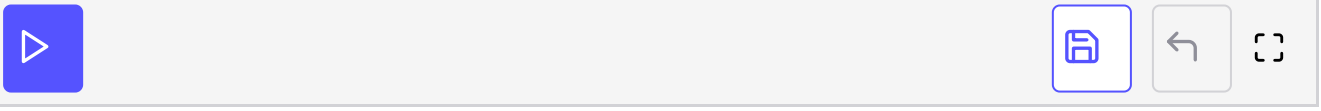
Once you've placed a pure virtual function in the base class, you must override it in all the derived classes from which you want to instantiate

objects. If a class doesn't override the pure virtual function, it becomes an

abstract class itself, and you can't instantiate objects from it (although you might from classes derived from it). For consistency, you may want to make all the virtual functions in the base class pure.

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

// A simple Shape interface which provides a method to get the Shape's area
class Shape {
  public:
  virtual float getArea() = 0;
};
// A Rectangle is a Shape with a specific width and height
class Rectangle : public Shape {   // derived form Shape class
  private:
  float width;
  float height;

  public:
  Rectangle(float wid, float heigh) {
    width = wid;
    height = heigh;
  }
  float getArea(){
    return width * height;
  }
};

// A Circle is a Shape with a specific radius
class Circle : public Shape {
  private:
  float radius;

  public:
  Circle(float rad){
    radius = rad;
  }
  float getArea(){
    return 3.14159f * radius * radius;
  }
};

// A Square is a Shape with a specific length
class Square : public Shape {
  private:
  float length;

  public:
  Square(float len){
    length = len;
  }
  float getArea(){
    return length * length;
  }
};
```

```
int main() {
  Shape * shape[3];   // Referencing Shape class to Rectangle object

  Rectangle r(2, 6);    // Creating Rectangle object
  shape[0] = &r;   // Referencing Shape class to Rectangle object

  Circle c(5);    // Creating Circle object
  shape[1] = &c;   // Referencing Shape class to Circle object

  Square s(10);   // Creating Square object
  shape[2] = &s;     // Referencing Shape class to Circle object

  for(int i=0; i<3; i++)
    cout << shape[i]->getArea() << endl;
}
```



Explanation

Now in `main()` you attempt to create objects of a `Shape` class, the compiler will complain that you're trying to instantiate an object of an abstract class. It will also tell you the name of the pure virtual function that makes it an abstract class. Notice that, although this is only a declaration, you never need to write a definition of the `Shape` class **getArea()**. Initialize the `Shape` class pointer and point it to objects of derived classes to access the **getArea()** function of respective classes.

Let's move on to a quick quiz to test your understanding of polymorphism.