Introduction to Abstract Data Types and Advanced Class Concepts in C++

In this class, we'll dive into the second set of topics in Object-Oriented Programming (OOP) in C++. These concepts build upon the basics of classes and objects, focusing on more advanced techniques like Abstract Data Types (ADTs), class interfaces, function and operator overloading, constructors, destructors, and more.

To be very honest, ADTs are just **fancy term for classes with encapsulation** which means they have some private data members and publicly available methods to get in touch with those data members.

So in general:

- **Normal Classes**: May expose data members directly, allowing both data and methods to be public.
- **ADT Classes**: Use encapsulation to hide data members, providing controlled access through public methods.

ADTs are normal classes that emphasize encapsulation, making sure that the internal state of the object is protected and manipulated only through a controlled interface. This approach not only secures the data but also makes the class easier to use and maintain.

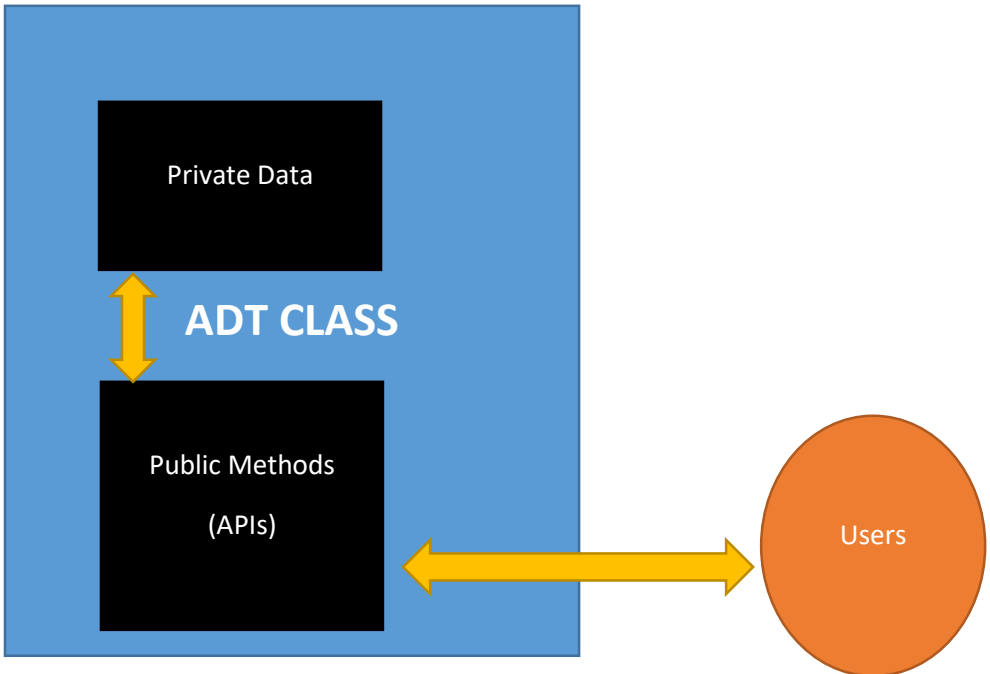

Figure 1 ADT is essentially encapsulated class. The private data is hidden and can only be accessed by public member functions known as Interface (APIs). This enhances the security.

Normal Class	ADT Class

#include <iostream>
#include <string>
using namespace std;
 	
class Person {
public:
  string name;
  int age;
 	
  void display() {
  cout << "Name: " << name << ", Age: " << age << endl;
  }
};
 	
int main() {
  Person p1;
  p1.name = "Alice";
  p1.age = 25;
  p1.display(); // Output: Name: Alice, Age: 25
 	
  return 0;
}
	#include <iostream>
	#include <string>
	using namespace std;
	
	class PersonADT {
	private:
	  string name;
	  int age;
	
	public:
	  void setName(string n) {
	  name = n;
	  }
	
	  string getName() {
	  return name;
	  }
	
	  void setAge(int a) {
	  if (a >= 0) {
	  age = a;
	  } else {
	  cout << "Age cannot be negative." << endl;
	  }
	  }
	
	  int getAge() {
	  return age;
	  }
	
	  void display() {
	  cout << "Name: " << name << ", Age: " << age << endl;
	  }
	};
	
	int main() {
	  PersonADT p1;
	  p1.setName("Alice");
	  p1.setAge(25);
	  p1.display(); // Output: Name: Alice, Age: 25
	
	  p1.setAge(-5); // Output: Age cannot be negative.

	return 0; }

So again, what are ADTs?

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation-independent view.

The term "abstract" in the context of data structures refers to the fact that the focus is on the functionality and behavior of the data structure, rather than the internal details or implementation.

In other words, when we work with an abstract data structure, we are interested in understanding how it operates and what it can do, without needing to know the specific inner workings or implementation details. The internal intricacies of the data structure are hidden or "obscured" from the user's perspective.

Let’s summarize

In simpler terms, an ADT specifies *what* operations can be performed on the data, not *how* these operations are implemented.

Think of a television remote control. The remote provides buttons for operations like turning the TV on or off, changing the channel, or adjusting the volume. You don’t need to know how the remote sends signals to the TV; you just use the buttons. The remote's interface (buttons) defines what it can do, regardless of the complex inner workings.

Designing ADTs, Class Interface (API)

Concept:

Designing an ADT involves defining the operations (methods) that will be available to users and hiding the implementation details.

The **Class Interface** (or API - Application Programming Interface) is the set of public methods that interact with the ADT.

Functions Overloading

Function Overloading allows you to define multiple functions with the same name but different parameter lists. It enables you to perform similar operations with different types or numbers of inputs.

Think of a Swiss Army knife. It has multiple tools (knife, scissors, bottle opener) that perform different functions but are all part of the same device. Similarly, overloaded functions share the same name but perform different tasks based on their parameters.

```
int add(int a, int b) {

    return a + b;

}

double add(double a, double b) {

    return a + b;

}

int add(int a, int b, int c) {

    return a + b + c;

}
```

Operator Overloading

Operator Overloading allows you to redefine the behavior of standard operators (like +, -, *, etc.) for user-defined types. It provides a way to use operators with objects in a natural, intuitive manner.

Example:

- Let's say "+" operator can be used to add a number and also concatenate two strings.
    - $(3 + 5i) + (-4 + 2i) = (-1 + 7i)$
- "+" can be used to add simple integers in different ways but complex numbers in a different way

We will see implementations of Operator Overloading after understanding some more concepts like Constructors.

Consider this …

We have primitive and non – primitive data types. Then we create data types of our own (ADTs).

Since we want **my tool my rule** kind of stuff in ADTs, we define a custom data structure with specific operations, that's where function and operator overloading comes in.

Function and operator overloading enhance the usability of ADTs by allowing you to define intuitive, natural ways to interact with the data structure, just like you would with built-in types.

ADTs

ADTs with Overloading

Let’s learn how to overload an operator! But wait, before diving into the concept. Let’s learn constructors first.

Constructors

A **Constructor** is a special function in a class that is automatically called when an object of that class is created. It initializes the object's attributes.

Imaging creating a Car Class and assigning it some default values. This can be done by constructors.

Or may be:

Think of a constructor as the process of building a new house. When you build a house, you automatically set up its structure, rooms, and basic utilities.

Similarly, a constructor sets up an object’s **initial state**.

How to create a constructor?

To create a constructor, use the same name as the class, followed by parentheses () and {}:

Syntax:

```
class ClassName {
  public:
      ClassName() {
```

```
        // constructor code
    }
};
```

NOTE:

- The constructor has the same name as the class.
- It is always public.
- It does not have any return value.

Basic Example:

```
class Car {
private:
  string brand;
  int year;
public:
  Car(string b, int y) : brand(b), year(y) {}
  void display() {
    cout << "Brand: " << brand << ", Year: " << year << endl;
  }
};
int main() {
  Car myCar("Toyota", 2020);
  myCar.display();  // Output: Brand: Toyota, Year: 2020
  return 0;
}
```

In this example, the constructor `Car(string b, int y)` initializes the brand and year of the car when the object is created.

NOTE: Constructors can also take parameters (just like regular functions), which can be useful for setting initial values for attributes.

Also, look at the following example and let's understand it.

```
class Car {         // The class
  public:           // Access specifier
    string brand;  // Attribute
    string model;  // Attribute
    int year;      // Attribute
    Car(string x, string y, int z) { // Constructor with parameters
      brand = x;
      model = y;
      year = z;
    }
};

int main() {
  // Create Car objects and call the constructor with different values
  Car carObj1("BMW", "X5", 1999);
  Car carObj2("Ford", "Mustang", 1969);

  // Print values
  cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
  return 0;
}
```

The above class have brand, model and year attributes, and a constructor with different parameters. Inside the constructor we set the attributes equal to the constructor parameters (brand=x, etc). When we call the constructor (by creating an object of the class), we pass parameters to the constructor, which will set the value of the corresponding attributes to the same.

If you notice, you can see that the constructor has saved us some extra lines of coding we used to do with Getters and Setters.

Understanding constructors in C++ is essential for effective object initialization and ensuring proper class functioning. Whether you're creating basic objects or complex data structures, constructors play a major role in setting up objects for use in your program.

NOTE: Constructors can also be defined outside the class. Yes!

In this case we have a different syntax then the regular constructor.

```
class Name {
        // class members
        public:
                Name() // Constructor declared
};
 Name::Name() {
// constructor initializations /  define
};
```

Consider this example:

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

class Car {
public:
    string brand;
    string model;
    int year;

    Car(); // Constructor declaration
};

// Constructor definition outside the class
Car::Car() {
    brand = "Ford";
    model = "Mustang";
    year = 2020;
}

int main() {
    Car myCar;
    cout << "Brand: " << myCar.brand << endl;
    cout << "Model: " << myCar.model << endl;
    cout << "Year: " << myCar.year << endl;
    return 0;
}
```

Note that the constructor is declared within the class but is defined outside the class.

Car::Car(string x, string y, int z) {brand = x; model, y; year = z;}

Summary:

- **Declaration Location**– Constructors are commonly declared in the public section of a class, though they can also be declared in the private section.
- **Return Type**– Constructors do not return values, thus they lack a return type.
- **Automatic Invocation**– Constructors are automatically invoked upon creating an object of the class.
- **Overloading**– Constructors can be overloaded, allowing multiple constructors with different parameter lists.
- **Virtual Declaration**– Constructors cannot be declared virtual, which distinguishes them from other member functions.
- **Address Reference**– The addresses of constructors cannot be directly referred to.
- **Memory Management**– Constructors implicitly call new and delete operators during memory allocation.
- **Naming Convention–** Constructors are member functions of a class and share the same name as the class itself.
- **Initialization Function**– Constructors serve as a particular type of member function responsible for initializing data members when objects are created.
- **Invocation Timing**– Constructors are invoked at the time of object creation, providing necessary data for the object.

Types of Constructors

Default Constructors

A default constructor is a constructor that can be called with no arguments or one that doesn't have any parameters. It initializes the member variables of a class to their default values.

```
class Car {
private:
   string brand;
   int year;

public:
   Car() : brand("Unknown"), year(0) {}

   void display() {
      cout << "Brand: " << brand << ", Year: " << year << endl;
   }
};

int main() {
   Car myCar;
   myCar.display();  //

   return 0;
}
```

Please NOTE that: Constructors are either provided by the programmer or automatically generated by the compiler if no constructors are defined.

Consider the following example, since no constructor is defined explicitly, the compiler provides an implicit default constructor. When an object bookObj of the Book class is created in the main() function, the default constructor is implicitly called but doesn't initialize the member variables.

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

class Book {
public:
    string title;
    string author;
};

int main() {
    Book bookObj;
    cout << "Title: " << bookObj.title << endl;
    cout << "Author: " << bookObj.author << endl;
    return 0;
}
```

Parameterized Constructor

On the other hand, we have Parameterized Constructors. A parameterized constructor has parameters that allow the initialization of member variables with specific values passed during object creation.

Syntax:

```
class ClassName {
public:
    ClassName(Type1 parameter1, Type2 parameter2, ...); // Parameterized
constructor declaration
};
```

Example: The car example we did earlier is an example of Parameterized Constructor.

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

class Rectangle {
private:
    int length;
    int width;

public:
    // Parameterized constructor defined inside the class
    Rectangle(int l, int w) {
        length = l;
        width = w;
    }

    int area() {
        return length * width;
    }
};

int main() {
    Rectangle rect(5, 3); // Creating object with parameters
    cout << "Area of Rectangle: " << rect.area() << endl;
    return 0;
}
```

You may also consider the above example.

TASK: Convert the above example of defining a parameterized constructor but this time the constructor should be defined outside the class.

NOTE: Remember to explicitly include a default constructor without parameters when you add one or more constructors with parameters to a class. If you don't, the compiler won't generate one automatically. Although it's optional, it's widely advised as a good practice always to supply a default constructor in such scenarios.

TASK: Also convert the above constructor example of Parameterized Constructor but this time initialize them with Default Values. Create three objects with the class.

1. One with no arguments passed to the object. **myObjRec();**
2. One with only one out of two arguments passed to the object. **myObjRec(10)**
3. One with both arguments passed to the object.**myObjRec(10,5)**

Copy Constructor

Think of making a photocopy of a document. The copy constructor is like the photocopier—it creates a new, identical document based on the original.

A Copy Constructor is a constructor that creates a new object as a copy of an existing object. It is called when an object is passed by value, returned from a function, or explicitly copied.

Note that the compiler provides an implicit copy constructor since no explicit copy constructor is defined.

For example, if we have a normal Class Person with a Parameterized constructor and with no class constructor. The compiler will automatically provide the implicit copy constructor.

```
int main() {
    Person person1("Alice");
    Person person2 = person1; // Copying person1 to person2
    cout << "Name of person2: " << person2.name << endl;
    return 0;
}
```

Syntax:

```
class ClassName {
public:
    ClassName(const ClassName& obj); // Copy constructor declaration
};
```

Here:

- **ClassName**: The name of the class.
- **const ClassName& obj**:
    - **const:** Ensures that the original object (obj) is not modified.
    - **ClassName&:** A reference to an object of the same class. Using a reference avoids unnecessary copying and is more efficient.
    - **obj:** The existing object that you want to copy.

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

class Person {
public:
    string name;

    // Explicit copy constructor defined
    Person(const Person& obj) {
        name = obj.name;
    }

    Person(string n) {
        name = n;
    }
};

int main() {
    Person person1("Bob");
    Person person2 = person1; // Copying person1 to person2
    cout << "Name of person2: " << person2.name << endl;
    return 0;
}
```

An object is initialized with another object of the same class (e.g., `Person person2 = person1;`).

Thus copy constructor allows you to define how an object of `Class` is copied, ensuring that any deep copies or specific copying behavior are handled appropriately.

Person person1("Bob")

Person person2 = person1

Person(const Person& obj) {

Where = What?

Name (Person2) = obj.name (Person1)

Example:

```cpp
class Car {
private:
  string brand;
  int year;

public:
  Car(string b, int y) : brand(b), year(y) {}

  Car(const Car& other) {
    brand = other.brand;
    year = other.year;
  }

  void display() {
    cout << "Brand: " << brand << ", Year: " << year << endl;
  }
```

```
};

int main() {
   Car car1("Toyota", 2020);
   Car car2 = car1;
   car2.display();
   return 0;
}
```

Destructor

A **Destructor** is a special function that is automatically called when an object is destroyed. It is used to clean up resources that the object may have acquired during its lifetime.

Think of cleaning up after a meal. The destructor is like the act of washing dishes and tidying up after the food has been eaten.

```
class Car {
public:
   Car() {
      cout << "Car is created." << endl;
   }

   ~Car() {
      cout << "Car is destroyed." << endl;
   }
};

int main() {
   Car myCar;  // Constructor is called
   // Destructor will be called automatically when myCar goes out of scope

   return 0;
}
```

NOTE: When the main() function finishes executing and the object myCar goes out of scope, this is where when the destructor is automatically invoked, releasing memory space no longer needed by the program.

General Properties of Destructors

- Destructors are special member functions in C++ that destroy class objects created by constructors.
- Destructors have the same name as their class, preceded by a tilde (~).
- Only one destructor can be defined per class and cannot be overloaded.
- Destructors are automatically called when objects go out of scope.

- They release memory occupied by objects created by constructors.
- Objects are destroyed in the reverse order of their creation within the destructor.

"This" Keyword

The **`this` keyword** is a pointer that refers to the current object instance. It is used to access members of the current object, especially when parameter names conflict with member names.

Think of referring to yourself in a conversation. When you say "I am going to the store," "I" is like the `this` pointer referring to yourself.

```
class Car {
private:
   string brand;

public:
   Car(string brand) {
      this->brand = brand;  // 'this' is used to differentiate between the parameter and the member
variable
   }

   void display() {
      cout << "Brand: " << this->brand << endl;
   }
};

int main() {
   Car myCar("Toyota");
   myCar.display();  // Output: Brand: Toyota

   return 0;
}
```