## 4. Classes & Objects

A white background with black text

Description automatically generated

A white background with black text

Description automatically generated

A screenshot of a computer program

Description automatically generated

A white background with black text

Description automatically generated

A white background with black text

Description automatically generated

A white background with black text

Description automatically generated

A screen shot of a computer

Description automatically generated

A white paper with black text

Description automatically generated

# Introduction to Object-Oriented Programming (OOP)

1. **Purpose:**
   * The purpose of OOP is to model real-world entities and systems using objects, facilitating a more intuitive and modular approach to software development.
2. **Key Concepts:**

* **Objects and Classes:** Objects are instances of classes, which define the blueprint for creating objects with shared properties and behaviors.
* **Relationships:** Objects interact with each other through relationships, such as inheritance and containment.
* **Inheritance:** Classes can inherit properties and behaviors from other classes, forming hierarchical relationships and promoting code reuse.
* **Encapsulation:** Encapsulation hides the internal implementation details of a class, exposing only the necessary interface to interact with it.
* **Polymorphism:** Polymorphism allows objects of different types to be treated uniformly, enabling flexibility and extensibility in the code.

1. **Abstraction:**

* Abstraction focuses on essential details while omitting unnecessary complexities, allowing developers to model real-world entities in software without replicating their intricacies.

1. **Encapsulation:**

* Encapsulation ensures that the internal workings of a class remain hidden from external entities, providing a clear separation between interface and implementation. This promotes ease of use and maintenance.

1. **Inheritance:**

* Inheritance establishes a hierarchy among classes, allowing child classes to inherit properties and behaviors from parent classes. This promotes code reuse and facilitates the modeling of relationships between objects.

1. **Polymorphism:**

* Polymorphism allows objects of different types to be treated interchangeably, depending on the context. This enables dynamic behavior and simplifies the implementation of complex systems.

1. **Implementation:**

* OOP principles can be implemented using various constructs in programming languages, such as classes, interfaces, and inheritance mechanisms.

1. **Benefits:**

* OOP promotes modularity, reusability, and maintainability of code, making it easier to develop and maintain large-scale software systems.
* By modeling real-world entities as objects, OOP enables developers to represent complex systems in a more intuitive and understandable manner.Top of Form

A white background with black text

Description automatically generated

A white screen with black text

Description automatically generated

A white background with black text

Description automatically generated

A white screen with black text

Description automatically generated

A screenshot of a computer error

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer error

Description automatically generated

1. **Introduction to Classes:**

* A class serves as a blueprint or template for creating objects.
* Each object is an instance of a class, and objects are created through a process called instantiation.

1. **Class Members:**

* Classes contain characteristics (attributes or member variables) and behaviors (member functions).
* Modifiers such as private, public, and protected control the visibility and accessibility of class members.
* Private members are accessible only within the class scope.

1. **Constructors:**

* Constructors are special member functions invoked automatically when an object is created.
* They initialize the object's state and can be overloaded to accept different sets of arguments.
* Types of constructors include default constructors (no arguments) and parameterized constructors (with arguments).

1. **Destructors:**

* Destructors are invoked automatically when an object is destroyed, releasing any resources acquired during its lifetime.
* They have the same name as the class preceded by a tilde (~) and cannot be overloaded or have arguments.

1. **Implementation in C++:**

* Class definitions are typically written in header (.h) files, while member function definitions can be implemented in separate source (.cpp) files.
* Accessing member functions and attributes of a class instance is demonstrated through example code snippets.

1. **Compiler Synthesis:**

* The compiler may automatically synthesize default constructors if no other constructors are provided by the user.
* Constructor and destructor calls can be observed during program execution, demonstrating their automatic invocation by the compiler.

1. **Parameterized Constructor:**

* Parameterized constructors accept arguments to initialize object attributes with user-defined values.

1. **Initialization:**

* It's crucial to initialize object attributes properly, either through constructors or directly within the class definition.

A white background with black text

Description automatically generated

1. **Creating User-Defined Types**: Structs, defined using the **struct** keyword, are an alternative to classes for creating user-defined types in C++.
2. **Similarity to Classes**: Structs bear a striking resemblance to classes, sharing many features and functionalities.
3. **Difference in Default Access**: The primary distinction lies in the default access level of their members. In a struct, member variables **are public by default**, whereas in a class, they are **private by default** unless explicitly stated otherwise.
4. **Usage in Representing Data Types**: Structs are commonly employed to represent abstract data types or record-like structures that necessitate public access to their members.
5. **Convenience of Structs**: Utilizing structs can be more convenient, especially when all members require public access. This avoids the need for manually specifying access modifiers, enhancing readability and ease of use.
6. **Example Demonstration**: An example illustrating the usage of structs is provided, focusing on a function called **DrawLine()**. By encapsulating related data (such as point coordinates) within a **Point** struct, readability and usability of the function are significantly improved.

A screen shot of a computer code

Description automatically generated

1. **Accessing Member Variables**: The default public access of struct members allows for direct access to member variables, simplifying code implementation.
2. **Comparison with Classes**: While structs and classes share many similarities, the key difference in default access highlights the flexibility and versatility of structs.
3. **Versatility of Structs**: It's emphasized that structs can be used interchangeably with classes in most scenarios, offering developers flexibility in choosing the appropriate type definition based on their specific needs.

A white paper with black text

Description automatically generated

* The compiler will automatically generate the initialization code and this code is inserted into the constructors. If the user has not implemented any constructor at all, then the compiler will synthesize one and insert the initialization code in that constructor.

A screenshot of a computer program

Description automatically generated

A computer code with text

Description automatically generated

A screenshot of a computer program

Description automatically generated

In C++11, non-static data member initializers offer a streamlined approach to initialize member variables directly within the class declaration itself, instead of relying solely on constructors. Here's a breakdown of the key points:

1. **Introduction of Non-Static Data Member Initializers**: These allow for member variable initialization directly within the class declaration, offering an alternative to initializing within constructors.
2. **Advantages of Non-Static Data Member Initializers**:

* Simplified Syntax: Initialization during declaration reduces the need for repetitive initialization code in constructors.
* Always Initialized: Member variables are guaranteed to be initialized with valid values, even if constructors are not explicitly implemented.
* Compatibility with Various Types: Member variables of any type, including variables, pointers, arrays, or user-defined objects, can be initialized using this approach.

1. **Automatic Code Generation**: The compiler automatically generates initialization code for member variables with non-static data member initializers, which is inserted into constructors.
2. **Example Illustration**:

* Demonstrates how member variables can be initialized during declaration within the class definition.
* Shows the removal of initialization code from constructors to avoid redundancy.
* Illustrates the initialization of member variables both with default values and user-specified values.

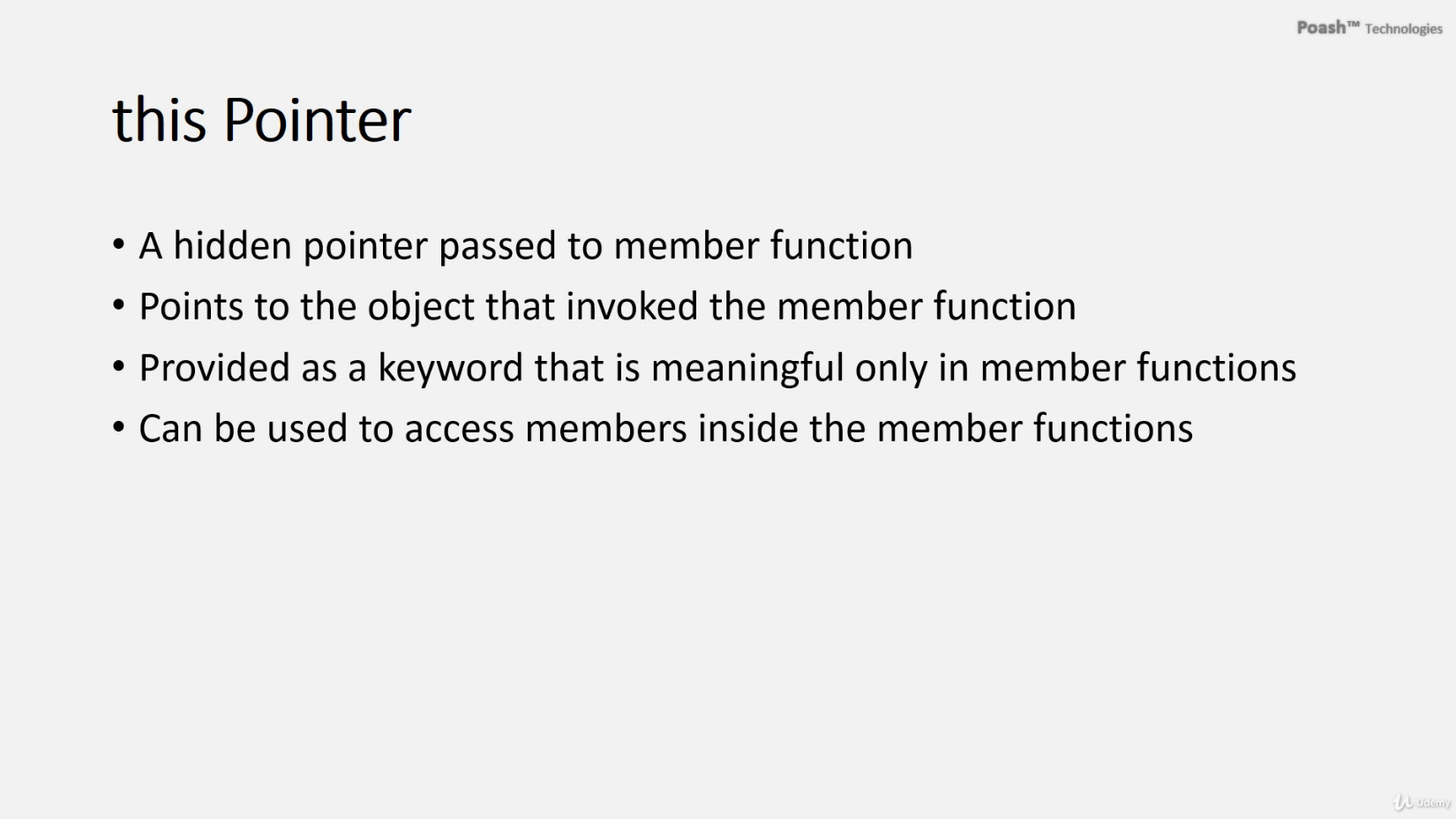
1. **Limitations**:
   * Use of non-static data member initializers restricts the use of the **auto** keyword for type deduction.
2. **Debugging and Compiler Output**:

* Debugging session showcases the assembly code generated by the compiler for initializing member variables.
* Highlights the automated nature of code generation for member variable initialization.

1. **Guidance on Usage**:

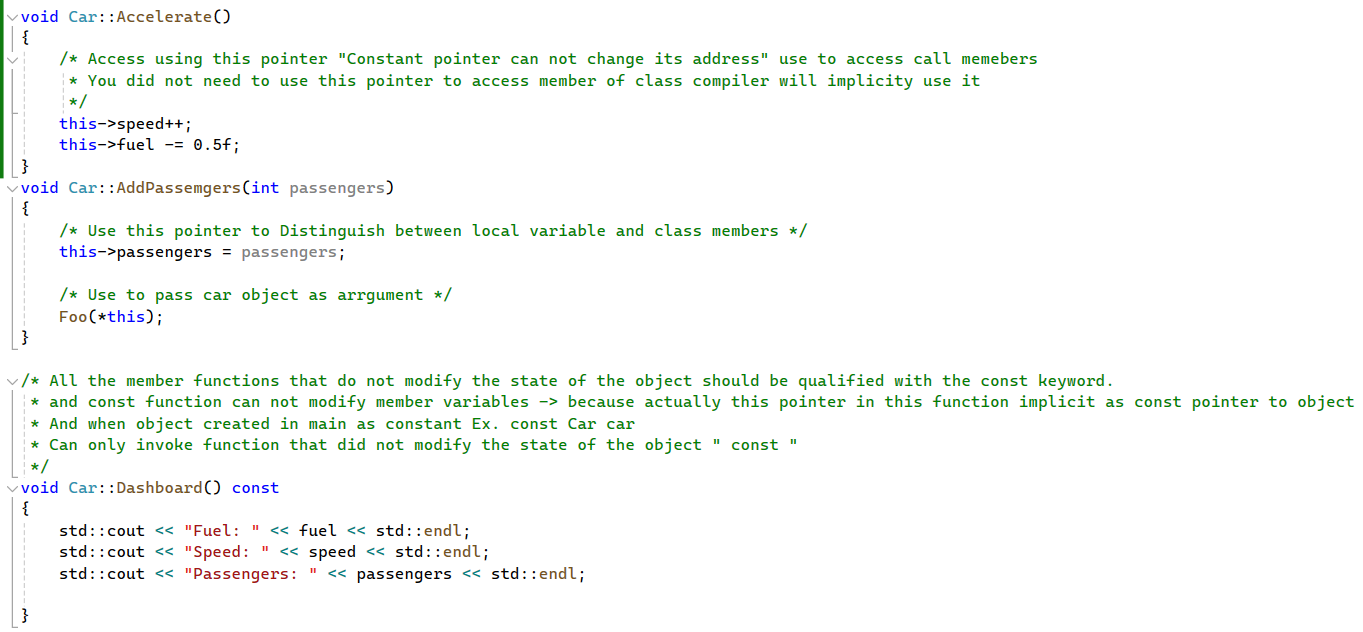
* Recommended for simple value initialization of member variables.
* For more complex initialization logic, constructors should be utilized.

In summary, non-static data member initializers offer a convenient and efficient way to ensure member variables are consistently initialized with desired values, while also reducing code redundancy and enhancing code readability.



**Used:**

* **To help the compiler distinguish between the member variable and the local variable that have the same name,**
* **if we have a function that accepts a Car object.**
* **some member functions that need to return the current object to the caller**.



# this Pointer

In C++, all member functions of a class implicitly receive a hidden parameter known as the "this" pointer. Here's a breakdown of the information provided:

1. **Purpose of "this" Pointer**:

* The "this" pointer contains the memory address of the object that invokes the member function.
* It enables member functions to differentiate between different objects invoking them.

1. **Usage**:

* When accessing members of a class inside a member function, the compiler internally utilizes the "this" pointer.
* While optional, the "this" pointer can be used as a prefix when accessing members within member functions.

1. **Address Passing**:

* Debugging session showcases how the address of the object is passed to member functions before each call.
* Assembly code analysis reveals the compiler-generated code for initializing the "this" pointer.
* Copy the address of the obeject and put it into register.

A screenshot of a computer program

Description automatically generated

* Put the address of object from the register to **“this” pointer**

A screenshot of a computer code

Description automatically generated

A close up of text

Description automatically generated

1. **Examples of "this" Pointer Usage**:

* To distinguish between member variables and local variables with the same name, the "this" pointer is used.
* When passing the object to a function from within a member function, the "this" pointer is dereferenced to obtain the object.
* In member functions that need to return the current object, the "this" pointer is utilized.

1. **Examples for Operator Overloading**:

* Member functions such as the prefix form of increment or decrement operators and assignment operators often require the "this" pointer for returning the current object.

In summary, the "this" pointer plays a crucial role in enabling member functions to access and manipulate member variables of the object they are invoked upon, facilitating object-oriented programming paradigms such as encapsulation and operator overloading.

A white background with black text

Description automatically generated

The **this** pointer in C++ is a pointer that holds the address of the current object. It is a special keyword that is implicitly available within non-static member functions of a class. The essence of using the **this** pointer lies in its ability to differentiate between local variables and member variables of the class when they share the same name.

Importance of the **this** pointer:

1. **Clarity and Disambiguation**: In member functions, the **this** pointer allows you to explicitly reference member variables of the class. This helps clarify the scope and origin of variables, especially in complex codebases or when dealing with inheritance.
2. **Accessing Member Functions and Variables**: Using **this->** allows you to access member functions and variables of the current object, even if they have the same name as local variables or parameters. This ensures you're working with the member variables of the class.
3. **Dynamic Memory Management**: The **this** pointer is particularly useful in dynamic memory management. It allows member functions to access and manipulate dynamically allocated memory of the object they are invoked on.
4. **Chaining Member Function Calls**: By returning **\*this** from member functions, you can enable method chaining, where multiple member functions can be called in succession on the same object.

# Constant Member Functions

Constant member functions in C++ are designated with the **const** keyword, ensuring they do not modify any member variables of the class:

1. **Purpose of const Member Functions**:

* **const** member functions are read-only functions that cannot alter the state of the object.
* They are useful for accessing and reading the state of an object without modification.

1. **Behavior with Constant Objects**:

* If an object is declared as constant, it can only invoke constant member functions.
* This restriction prevents modification of the object's state, maintaining data integrity.

1. **Qualifying Member Functions with const**:

* Member functions that do not modify the object's state should be qualified with **const**.
* This qualifier should be applied in both the declaration and the definition of the function.

1. **Applying const to this Pointer**:

* The **const** qualifier applied to a member function affects the **this** pointer implicitly.
* Specifically, it applies to **\*this**, indicating that the object itself is constant within the member function.

A screenshot of a computer error message

Description automatically generated

1. **Implications for Member Function Invocation**:

* A **const** member function can only invoke other **const** member functions of the object.
* This restriction ensures that no non-constant member functions are inadvertently called on a constant object.

By employing **const** member functions, C++ facilitates the creation of objects with read-only access, enhancing program reliability and maintainability by enforcing data immutability.

Top of Form

Top of Form

A white background with black text

Description automatically generated

A white background with black text

Description automatically generated

Static member variables in C++ are shared among all instances of a class and are declared with the **static** keyword:

1. **Definition and Scope**:
   * Static member variables are declared inside the class but defined outside.
   * They are not part of any individual object but belong to the class itself.
   * Only one copy of the static variable exists, shared among all objects of the class.
2. **Initialization**:
   * Static variables cannot be initialized inside the class declaration.
   * They must be defined and initialized outside the class, typically in a source file.
   * If no explicit initializer is provided, static variables are automatically initialized to zero or the default value of their type.
3. **Accessing Static Variables**:
   * Static member variables can be accessed in all member functions of the class.
   * They are commonly used for maintaining shared state or counting instances of a class.
4. **Example Implementation**:
   * An example is provided where a **totalCars** static member variable is declared in the **Car** class to count the total number of **Car** objects.
   * The **totalCars** variable is defined and initialized outside the class declaration, and it's automatically initialized to zero.
   * Member functions, such as constructors and destructors, are used to update the **totalCars** count accordingly.
5. **Static Member Functions**:
   * Static member functions are declared with the **static** keyword and belong to the class, not to individual objects.
   * They do not receive a **this** pointer and can only access static members of the class.
   * Static member functions can be invoked either through an object of the class or through the class name itself.
6. **Example of Static Member Function**:
   * An example **ShowCount()** static member function is added to the **Car** class to display the total count of **Car** objects.
   * The **ShowCount()** function is defined outside the class and is called without needing an object, demonstrating its static nature.
7. **Debugging Static Member Function**:
   * Debugging reveals that static member functions do not receive a **this** pointer, as opposed to non-static member functions.
   * Static member functions are accessed directly without passing an object address.

Overall, static member variables and functions provide a way to maintain shared state and behavior across all instances of a class in C++

# Copy constructor



A copy constructor in C++ is a special constructor that creates a new object as a copy of an existing object. It plays a crucial role in managing object copying and resource allocation. Let's delve into the details:

**1. Creation of Copy:**

* A copy of an object is created in several scenarios:
  + When it is passed by value into a function.
  + When the function returns the object by value.
  + When a manual copy of the variable is created.

**2. Shallow Copy vs. Deep Copy:**

* Copying a pointer results in a shallow copy, where only the addresses are copied. Changes made to one pointer reflect in all other pointers sharing the same address.
* To avoid this, a deep copy is performed, where the actual values at the memory addresses are copied, ensuring independent memory space for each pointer.

**3. Performing Deep Copy:**

* To perform a deep copy, new memory is allocated, and the value at the address is copied. This prevents changes made to one object from affecting others sharing the same address.

**4. Copy Constructor Usage:**

* A copy constructor is invoked when an object is passed into a function by value, when a function returns an object by value, or when one object is assigned to another object.

**5. Need for User-Defined Copy Constructor:**

* A user-defined copy constructor is essential to handle deep copying properly, especially when dealing with dynamically allocated resources like pointers.
* It should accept the object by reference to prevent an infinite loop of copy constructor invocations, and qualifying the object parameter with the const keyword ensures that the original object cannot be modified within the function.



**Compiler Synthesized Copy Constructor**:

* If the user does not provide a copy constructor, the compiler automatically synthesizes one.
* The default implementation of the synthesized copy constructor simply copies the values of member variables.
* However, this default behavior may not be desirable, especially when dealing with dynamically allocated resources like pointers.

**Example with the Integer Class**:

* An example is provided with a class called **Integer**, which is a wrapper around the primitive type **int**.
* The **Integer** class contains a pointer as a member variable and dynamically allocates memory for the integer value.
* Constructors (**default** and **parameterized**), **GetValue()**, and **SetValue()** functions are implemented.
* A user-defined destructor is implemented to release the dynamically allocated memory.

**Issues with Automatic Copy Constructor**:

* One problem with the provided **Integer** class is that the compiler-synthesized copy constructor does a shallow copy of the pointer.
* This results in multiple objects pointing to the same dynamically allocated memory, leading to issues like double deletion or memory leaks.

**Cases Requiring Copy Constructor:**

* Various scenarios are mentioned where a copy of an object is created:
* Direct invocation of the copy constructor.
* Passing an object by value into a function.
* Returning an object by value from a function.

**Rule of 3:**

* The Rule of 3 states that if a class implements any of the following functions (destructor, copy constructor, copy assignment operator), it should implement all three. This ensures proper resource management, especially when dealing with dynamically allocated memory.

By understanding these concepts and adhering to best practices such as the Rule of 3, developers can effectively manage object copying and resource allocation, mitigating common issues related to shallow copying and ensuring proper memory management in C++ classes.

A white paper with black text

Description automatically generated

A screenshot of a computer program

Description automatically generated

# A delegating constructor

**A delegating constructor in C++ allows one constructor to invoke another constructor within the same class:**

**1. Purpose of Delegating Constructors:**

* Delegating constructors are beneficial when a class has multiple constructors, and all of them need to perform some common initialization.
* By using delegating constructors, the common initialization code can be written in one constructor and invoked from all other constructors, reducing code duplication.

**2. Example with Default and Parameterized Constructors:**

* The provided example involves a class containing two constructors: a default constructor and a parameterized constructor.
* The parameterized constructor includes the common initialization code, while the default constructor delegates to the parameterized constructor by passing default values as arguments.

**3. Common Initialization Code:**

* The common initialization code occurs only once in the parameterized constructor, reducing the chances of bugs due to inconsistent or skipped initialization.

**4. Application of Delegating Constructors:**

* By applying the concept of delegating constructors introduced in C++11, the common initialization code is kept in one constructor, and other constructors delegate calls to this constructor.
* This ensures that the state of the object is always initialized correctly, regardless of which constructor is used.

**5. Order of Constructor Invocation:**

* The order in which the constructors are invoked is explained, demonstrating how the delegating constructor is called before the body of the constructor is executed.
* This sequence ensures that the common initialization code is executed before specific initialization code in other constructors.

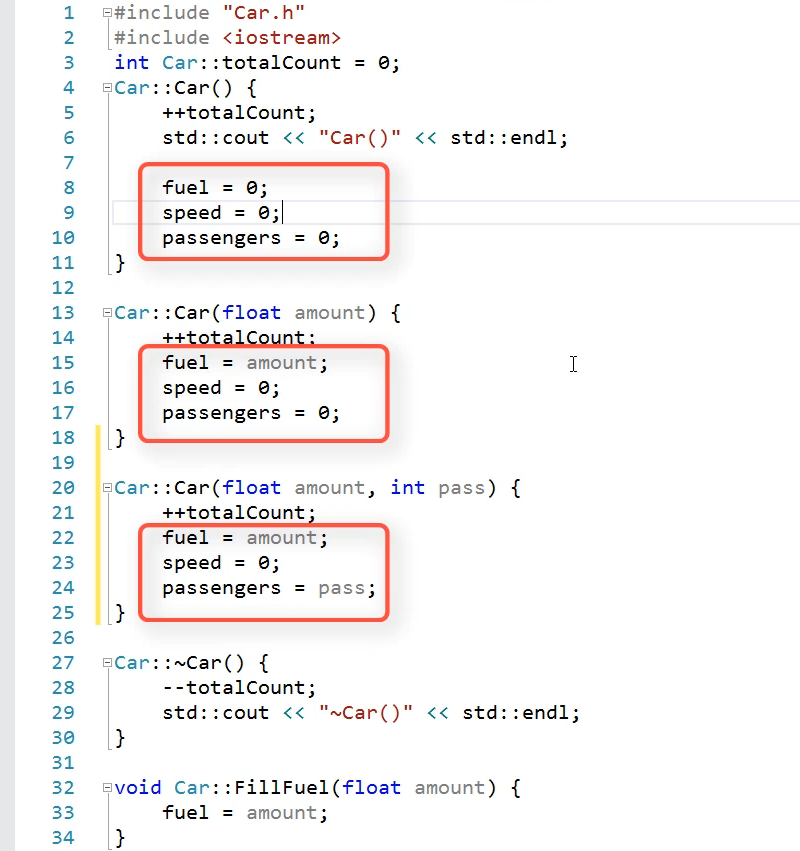
**6. Importance of Avoiding Code Duplication:**

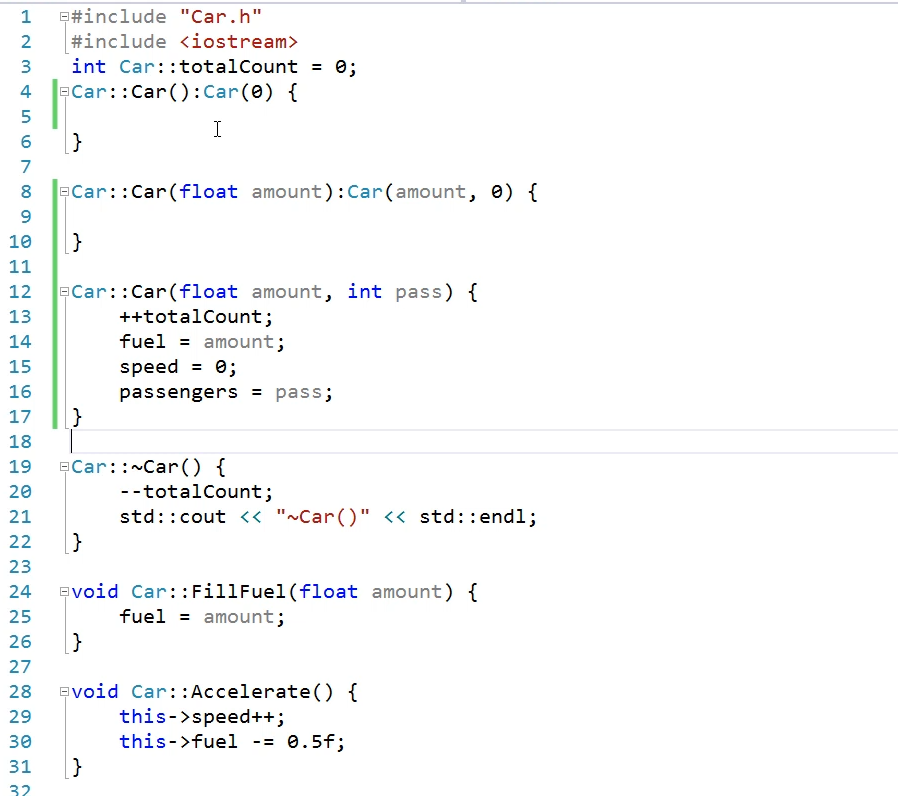
* Keeping the common initialization code in one constructor helps avoid code duplication, reducing the chances of errors and simplifying maintenance.

**7. Complete Execution of Constructors:**

* It's emphasized that all constructors are executed, ensuring that the object's state is fully initialized regardless of which constructor is used.

By understanding and implementing delegating constructors, developers can streamline the initialization process, reduce code duplication, and ensure consistent object state initialization across multiple constructorsTop of Form





# Default & Deleted Functions

In C++11, several new keywords were introduced that can be used with classes to control their behavior and automatically generate or prevent the generation of certain functions. Let's discuss two of these keywords: **default** and **delete**.

1. **default Keyword:**
   * The **default** keyword can be used to instruct the compiler to generate default implementations of certain member functions if they are not explicitly defined.
   * This keyword is particularly useful for generating default constructors, copy constructors, copy assignment operators, and destructors.
   * Example usage:

A computer code with text

Description automatically generated

* + When a member function is marked with **default**, the compiler will automatically synthesize its implementation if no explicit definition is provided.

1. **delete Keyword:**
   * The **delete** keyword can be used to explicitly delete certain member functions, preventing their use.
   * This keyword is useful for preventing the generation of default implementations by the compiler or for disabling certain operations for specific functions.
   * Example usage:

A close-up of a white background

Description automatically generated

* + When a member function is marked with **delete**, attempting to use that function will result in a compile-time error, preventing its invocation.

These keywords provide greater control over the behavior of classes and allow developers to enforce certain design decisions and constraints at compile time. They contribute to safer and more expressive C++ code.