## Lambda

* C++11 introduced lambda expressions, which serve as concise alternatives to function objects. They create anonymous function objects without the need for explicitly defining a class or structure and overloading the function call operator.
* Lambda expressions can be passed as arguments to functions accepting function objects, behaving similarly to functions with parameters and return values.
* Internally, lambda expressions are implemented as nameless function objects, with the compiler generating a class that overloads the function call operator.
* To provide a name to a lambda expression, you can use `auto` to create a variable and assign the lambda expression to it.

**A lambda expression consists of:**

* The lambda introducer `[ ]`, marks the start of the expression and contains an optional capture clause.
* Optional arguments and specifications like `mutable`.
* Optional exception specifications.
* Trailing return type syntax for specifying the return type.
* The implementation in the lambda body is enclosed by curly braces `{ }`.

**More ditals of A lambda expression**

1. Lambda Introducer: The lambda introducer consists of a pair of square brackets []. Inside these brackets, you can specify capture clauses to capture variables from the enclosing scope. For example:

* [ ] captures nothing.
* [&] captures all variables by reference.
* [=] captures all variables by value.
* [x, &y] captures x by value and y by reference.
* [=, &y] captures all by value and y by reference.
* [&, y] captures all by reference and y by value.
* Optional Arguments: After the capture clause, you can specify the arguments to the lambda expression, followed by an optional **mutable** keyword if you want to modify captured variables inside the lambda.
* The capture list can capture only those variables that have been declared before the lambda expression.

1. Optional Exception Specification: You can specify an exception specification if needed.
2. Return Type: The return type of a lambda expression is deduced by the compiler in most cases. However, you can specify the return type explicitly using trailing return type syntax with the arrow ->. For example: -> int specifies that the lambda expression returns an integer.
3. Lambda Body: The implementation of the lambda expression occurs within the lambda body, enclosed by curly braces { }. This is where you write the code that defines the behavior of the lambda expression.

A white background with black text

Description automatically generated

A white background with black text

Description automatically generated

A computer screen shot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

A computer screen shot of a program

Description automatically generated

1. **Lambda Expression Overview:**

* Lambda expressions offer a concise syntax for defining anonymous function objects, eliminating the need to explicitly declare classes or structures.
* They provide a convenient mechanism for creating small, inline functions directly within code blocks.

1. **Return Type Inference:**

* Lambda expressions can deduce their return type if all return statements within the body yield the same type.
* In cases where multiple return types are present, the return type must be explicitly specified using trailing function syntax (->).

1. **Internal Implementation:**

* Internally, a lambda expression functions as a function object, essentially creating a temporary, unnamed class with an overloaded function call operator.
* The compiler synthesizes this structure and may also generate a type conversion operator for compatibility with function pointers.

1. **Generality via Templates:**

* Lambda expressions can be made generic by using `auto` for argument types, enabling them to work with various data types.
* This feature, akin to templates for function objects, was introduced in C++14, expanding the versatility of lambda expressions.

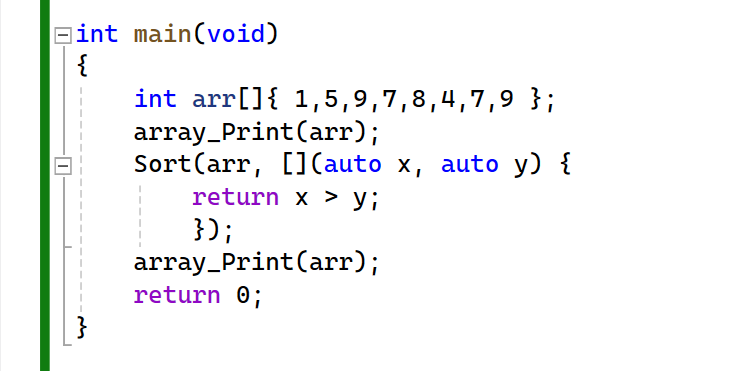
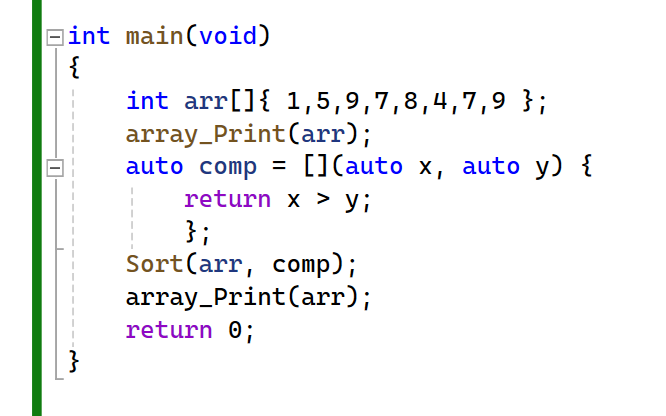
1. **Exception Specification:**

* Lambda expressions can be annotated with exception specifications to indicate whether they throw exceptions.
* By default, lambda expressions do not throw exceptions unless explicitly specified using `noexcept(false)`.

1. **Usage and Benefits:**

* Lambda expressions enhance code readability and maintainability by encapsulating functionality directly where it's needed, promoting a more functional programming style.
* They facilitate concise and expressive code, particularly in scenarios requiring short, one-off functions or custom predicates for algorithms like `std::sort`.

# Lambda Expressions Capture List



**Using a lambda expression offers several advantages over traditional function pointers or function objects:**

1. **Readability and Context:**

* Lambda expressions provide a clear and immediate understanding of the callback's purpose right at the point of use. This eliminates the need for readers to search through the code to understand the functionality encapsulated by a function pointer or function object.

1. **Conciseness and Locality:**

* With lambda expressions, you can define the callback inline with the function call, avoiding the need to create separate structures or functions for simple operations. This keeps the code concise and maintains a clear relationship between the callback and its usage.

1. **Namespace Pollution:**

* Traditional functions or function objects introduce new names into the global namespace, potentially cluttering it with entities that are only relevant to a specific context. Lambda expressions, being anonymous, do not pollute the namespace, leading to cleaner and more maintainable code.

1. **Reusability:**

* While lambda expressions are often used inline, they can also be stored in automatic variables for reuse within the same function or even in different parts of the program. This allows for flexibility in code organization without sacrificing readability or efficiency.

In summary, lambda expressions offer a powerful combination of clarity, conciseness, and flexibility, making them a preferred choice for defining simple callbacks in modern C++ programming. Their ability to encapsulate functionality within the immediate context of use enhances code readability and maintainability while minimizing namespace clutter and facilitating code reuse.

* To simplify performing operations on elements of an array without needing to write a for loop each time, we can create a generic algorithm called `foreach`. This algorithm will iterate over each element in the array and apply a user-defined operation to it, which will be specified as a callback.
* The `foreach` algorithm takes an array (`arr`) and a callback (`operation`) as arguments. It iterates over each element in the array and calls the callback function, passing the current element as an argument.
* To use `foreach`, we specify the array and provide the callback. The callback can be implemented using a function pointer, a function object, or a lambda expression.

**For example, using a lambda expression to print each element of the array:**

A computer code with many numbers and symbols

Description automatically generated with medium confidence

By providing this abstraction, `foreach` allows users to easily apply custom operations to each element of an array, enhancing code readability and maintainability.

A screenshot of a computer

Description automatically generated

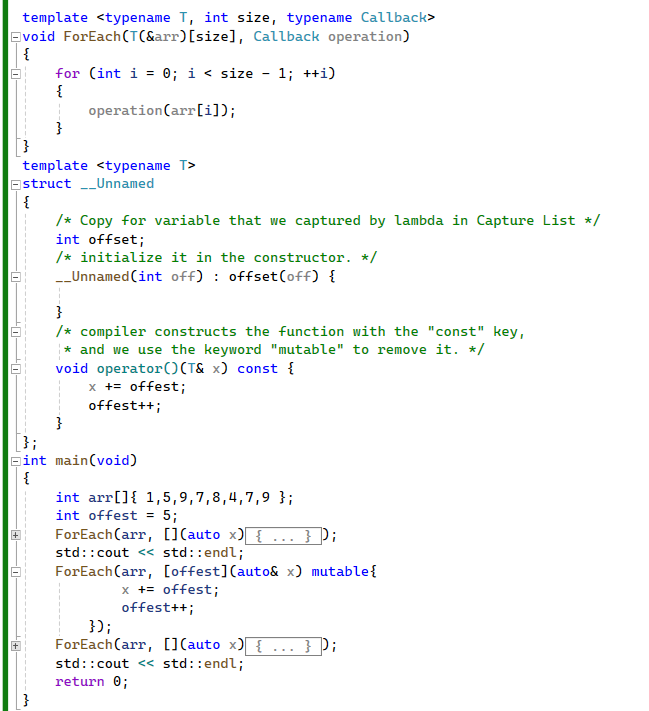
* To modify all elements of an array by adding a specified offset value to each element, we utilize the foreach algorithm along with lambda expressions. In this scenario, the lambda expression accepts the offset value by reference, enabling the application of the desired operation on each element of the array.
* Suppose we aim to add an offset of 5 to each element within the array. We can achieve this by applying the operation x += offset inside the lambda expression. However, upon attempting to compile the code, an error occurs due to the lambda's inability to access the offset variable defined in the main function.
* To address this issue, we explicitly capture the offset variable within the lambda's capture list. By capturing the variable by copy, we ensure its accessibility within the lambda expression. It's important to note that the captured offset variable inside the lambda expression is distinct from the original offset variable.

**Below is the revised implementation using templates and lambda expressions:**

A screenshot of a computer program

Description automatically generated

# Mutable



**Let's break down the process and the implementation of the lambda expression with a mutable offset:**

1. **Lambda Expression Requirements:**

* We want to add the offset in increasing order to each element inside the array.
* For each call to the lambda expression, we want to increment the offset.

1. **Compiler Implementation:**

* Lambda expressions are internally implemented as anonymous function objects.
* These function objects have a function call operator (`operator()`), which is const by default.
* By default, member variables within the function object cannot be modified.

1. **Modifying Offset Inside Lambda Expression:**

* To modify the offset inside the lambda expression, we need to remove the constness of the function call operator.
* We can achieve this by using the `mutable` keyword in the lambda expression.

1. **Implementation Details:**

* The lambda expression is translated by the compiler into an unnamed struct (`\_\_Unnamed`) with a mutable function call operator.
* Inside the struct, the offset variable is initialized in the constructor and is modified within the mutable function call operator.

1. **Example Usage:**

* We demonstrate the usage of the lambda expression with a mutable offset by applying it to each element of an array.
* The offset is incremented for each call to the lambda expression, effectively adding increasing values to the array elements.

This approach allows us to modify variables within the lambda expression, providing flexibility and control over the behavior of the operation applied to each element of the array.

# Capture List with this pointer

A screen shot of a computer code

Description automatically generated

we're utilizing a lambda expression within a class, specifically the `Product` class. The purpose of this class is to hold information about a product, including its name and base price. Additionally, the class contains a method called `AssignFinalPrice()`, which calculates the final price of the product by adding taxes to the base price.

1. **Class Definition:**

* The `Product` class contains member variables for the product name (`name`) and base price (`price`).
* It also defines a method `AssignFinalPrice()` to calculate the final price of the product.

1. **Taxes Array:**

* The taxes to be applied are stored in an array, representing different percentages of taxes.

1. **Lambda Expression Usage:**

* The `ForEach()` function is used to iterate over the taxes array.
* Within the lambda expression passed to `ForEach()`, each tax percentage is added to the base price to calculate the final price (`taxedPrice`).
* The lambda expression captures the `this` pointer to access member variables of the class, such as `price`.

1. **Capture List:**

* Local variables, such as `basePrice`, are captured by value within the lambda expression.
* Member variables are accessed via the `this` pointer, allowing modification within the lambda.

the usage of lambda expressions within a class to perform operations involving member variables. By capturing the `this` pointer, lambda expressions can access and modify member variables effectively.

**Lambda expression can be called from another Lambda expression.**

A close-up of a computer screen

Description automatically generated

# a lambda expression to a function pointer



A screenshot of a computer program

Description automatically generated

* The behavior where a lambda expression decomposes to a function pointer if it doesn't specify a capture clause allows for using lambda expressions as callbacks in C functions.
* For instance, the `atexit()` C function, which registers a callback to be invoked before program termination, can accept a lambda expression as its argument. Since the callback for `atexit()` doesn't accept any arguments, a lambda expression without a capture clause decomposes to a function pointer.
* This decomposition occurs because every lambda expression without a capture clause provides a type conversion operator, which returns a pointer to an internal static member function inside the lambda expression. This static member function then internally invokes the overloaded function call operator of the lambda expression.
* However, it's important to note that if a lambda expression includes a capture clause, the type conversion operator isn't provided. In such cases, attempting to use the lambda expression as a function pointer will result in an error.
* In summary, lambda expressions offer flexibility and convenience, especially when used as callbacks in functions like `atexit()`, provided they don't include a capture clause.

# Generalized Lambda Capture

A screenshot of a computer

Description automatically generated

A computer screen shot of a program code

Description automatically generated

Generalized lambda capture, introduced in C++14, offers the ability to create new variables directly within the capture clause of a lambda expression. These variables are initialized using an expression and their types are automatically deduced. Additionally, references can be created by prefixing the variable name with an ampersand (&).

Consider the following examples to illustrate the usage of generalized lambda capture:

1. **Basic Example**:
   * A variable **x** is initialized with a value, and a lambda expression captures it by value.
   * Instead of directly capturing **x**, a new variable **val** is created in the capture clause using generalized capture.
   * The lambda expression then returns the sum of **val** and its argument **arg**.
2. **Usage with File I/O**:
   * When writing data to a file using an **ofstream** object within a lambda expression, the object needs to be captured.
   * Since **ofstream** objects are non-copyable, they must be captured by reference.
   * Generalized lambda capture allows for the creation of a new **ofstream** object within the capture clause, ensuring its usage is limited to the lambda expression.
   * The **move** keyword moves the state of the object into the lambda's capture, and **mutable** is used to modify member variables within the lambda expression.
3. **Unique Pointer Example**:
   * Generalized lambda capture is also useful when working with **unique\_ptr** objects.
   * Similar to the file I/O example, it allows for the creation of a new **unique\_ptr** within the capture clause, ensuring its usage is restricted to the lambda expression.

Generalized lambda capture provides a convenient way to manage variables within lambda expressions, improving readability and encapsulation. However, it's essential to note that this feature is specific to C++14 and may not be available in earlier versions of the language.

Top of Form