1. What is a lambda function in C++11?

A lambda function in C++11 is an anonymous function that can be defined inline within a code block. It provides a way to create and use small, one-time, and throwaway functions without the need for explicitly defining a separate named function. Lambdas are often used as arguments to higher-order functions or for concise code snippets. They capture variables from their surrounding scope and can be invoked and executed like regular functions.

2. How do you define a lambda function in C++11?

In C++11, a lambda function can be defined using the following syntax:

```
[capture-list](parameters) mutable(optional) exception-specifier(optional) -> return-type(optional)
{
        // Lambda function body
}
```

Let's break down the components:

- **Capture-list**: It specifies which variables from the surrounding scope should be captured by the lambda function. It can be empty (`[]`), capture specific variables by value (`[var1, var2]`), capture all local variables by value (`[=]`), capture all local variables by reference (`[&]`), or capture specific variables by reference (`[&var1, &var2]`).

- **Parameters**: It represents the list of parameters that the lambda function accepts. These parameters are similar to regular function parameters and can have types, names, and default values.

- **Mutable (optional)**: If specified, it allows the lambda function to modify the captured variables by value.

- **Exception-specifier (optional)**: It specifies the exception handling behavior of the lambda function. This is similar to the exception specification on regular functions.

- **Return-type (optional)**: It defines the return type of the lambda function. If not specified, the return type is deduced automatically based on the return statements within the lambda body.

3. What is the syntax for capturing variables in a lambda function?
In C++11, variables can be captured in a lambda function using the capture list, which is specified at the beginning of the lambda function definition. The capture list has the following syntax:

```
[capture-list]
```

There are different ways to capture variables:

- **Capture by value**: To capture variables by value, you can specify them in the capture list using their names or using the general capture `[=]` syntax to capture all variables by value. For example, `[x]` captures variable `x` by value, and `[=]` captures all variables by value.

- **Capture by reference**: To capture variables by reference, you can specify them in the capture list using the ampersand `&` followed by their names or using the general capture `[&]` syntax to capture all variables by reference. For example, `[&y]` captures variable `y` by reference, and `[&]` captures all variables by reference.

- **Mixed capture**: You can also mix capture by value and capture by reference in the same capture list. For example, `[x, &y]` captures variable `x` by value and variable `y` by reference.

- **Omitted capture**: If you don't need to capture any variables from the surrounding scope, you can use an empty capture list `[]`.

Note that variables captured by value are copied into the lambda function, while variables captured by reference are accessed directly.

4. Explain the concept of capturing variables by value and by reference in a lambda function.

When capturing variables in a lambda function, you have the option to capture them either by value or by reference. Here's an explanation of these two concepts:

1. **Capture by value**: When you capture variables by value, a copy of the variable's value is made and stored within the lambda function. Any changes made to the captured variable inside the lambda function will not affect the original variable in the surrounding scope. Capturing by value is denoted by `[var]` or `[=]` in the capture list, where `var` represents the variable to be captured.

Example:
```cpp
int x = 5;
auto lambda = [x]() {
        // x is captured by value
        // Any changes to x will not affect the original variable in the surrounding scope
        // ...
};
```

2. **Capture by reference**: When you capture variables by reference, the lambda function holds a reference to the original variable in the surrounding scope. Any changes made to the captured variable inside the lambda function will directly affect the original variable. Capturing by reference is denoted by `[&var]` or `[&]` in the capture list, where `var` represents the variable to be captured.

Example:
```cpp
int y = 10;
auto lambda = [&y]() {
        // y is captured by reference
        // Any changes to y will affect the original variable in the surrounding scope
        // ...
};
```

It's important to be cautious when capturing variables by reference because if the lambda function outlives the lifetime of the captured variable, accessing the captured reference will lead to undefined behavior.

5. What is the purpose of the mutable keyword in a lambda function?

In C++11, the `mutable` keyword is used in a lambda function to indicate that the lambda function can modify variables captured by value. By default, variables captured by value are considered `const` within the lambda function, meaning that any attempt to modify them would result in a compilation error.

However, when the `mutable` keyword is specified in the lambda function definition, it allows the lambda function to modify the captured variables. This is particularly useful when you want to modify variables within the lambda function without affecting the original variables in the surrounding scope.

Here's an example that demonstrates the use of `mutable`:

```cpp
int x = 5;
auto lambda = [x]() mutable {
        // x is captured by value and is considered const by default
        // Adding the mutable keyword allows modifying x within the lambda function
        x += 10;
        // ...
};
```

In this example, `x` is captured by value and considered `const` by default. However, with the `mutable` keyword, the lambda function is allowed to modify the captured `x` by incrementing its value by 10. The changes made to `x` inside the lambda function do not affect the original variable `x` in the surrounding scope.

6. How do you invoke a lambda function?

To invoke a lambda function in C++11, you can treat it like a regular function and use the function call operator `()` along with the necessary arguments. Here's an example:

```cpp
// Lambda function that takes two integers and returns their sum
auto lambda = [](int a, int b) {
        return a + b;
};

// Invoking the lambda function
int result = lambda(3, 4);
```

In this example, the lambda function takes two integer parameters and returns their sum. To invoke the lambda function, you simply use the function call operator `()` and provide the required arguments (`3` and `4` in this case). The result of the lambda function invocation is stored in the variable `result`.

Note that you can also invoke a lambda function without storing it in a variable, by directly calling it at the point of its definition:

```cpp
int result = [](int a, int b) {
        return a + b;
}(3, 4);
```

In this case, the lambda function is defined and immediately invoked with the provided arguments (`3` and `4`), and the result is assigned to the variable `result`.

7. Can a lambda function have a return type? If yes, how do you specify it?

Yes, a lambda function can have a return type in C++11. You can specify the return type using the trailing return type syntax. Here's how you can specify the return type of a lambda function:

```cpp
// Lambda function with explicit return type
auto lambda = [](int a, int b) -> int {
        return a + b;
};
```

In this example, the lambda function takes two integer parameters and returns their sum as an integer. The return type is specified after the parameter list using the `->` arrow syntax, followed by the desired return type (`int` in this case).

It's important to note that in most cases, the return type of a lambda function can be deduced automatically by the compiler based on the return statements within the lambda body. Explicitly specifying the return type is necessary only when the return type cannot be deduced unambiguously by the compiler, or when you want to enforce a specific return type.

8. How does type inference work with lambda functions in C++11?

In C++11, the type inference for lambda functions is based on the context in which the lambda function is used and the form of the lambda function itself. Here's how type inference works with lambda functions in C++11:

1. **Parameter types**: The compiler can infer the types of the lambda parameters based on the types of the arguments passed to the lambda function when it is invoked or assigned to a variable. The compiler examines the context in which the lambda function is used to deduce the parameter types.

Example:
```cpp
auto lambda = [](int a, float b) {
    // Lambda function body
};

// Parameter types are deduced based on the argument types when invoking the lambda
lambda(10, 3.14f);
```
In this example, the lambda function is invoked with integer and float arguments. The compiler deduces that the lambda function has parameters of type `int` and `float` based on the argument types provided.

2. **Return type**: In C++11, the return type of a lambda function cannot be automatically deduced by the compiler. The return type needs to be explicitly specified using the trailing return type syntax.

Example:
```cpp
auto lambda = [](int a, int b) -> int {
    // Lambda function body
    return a + b;
};
```
In this example, the return type of the lambda function is explicitly specified as `int` using the trailing return type syntax (`-> int`). The compiler does not infer the return type automatically, and it needs to be specified explicitly.

It's important to note that while C++11 provides limited type inference capabilities for lambda functions, later versions of C++ (C++14 and beyond) introduced more powerful type inference, allowing the use of `auto` to automatically deduce both parameter types and the return type of a lambda function in many cases.

9. What are the advantages of using lambda functions in C++11?

Lambda functions in C++11 offer several advantages that make them a valuable feature in the language. Here are some benefits of using lambda functions:

1. **Concise and readable code**: Lambda functions allow you to write compact and self-contained code snippets directly inline, reducing the need for defining separate named functions. This can enhance code readability, especially when the functionality is simple and doesn't warrant a separate function.

2. ****Closure and access to surrounding scope****: Lambda functions can capture variables from their surrounding scope, providing access to external variables within the function body. This makes it convenient to work with variables in the context where the lambda function is defined.

3. ****Flexible and adaptable****: Lambda functions can be used as arguments to higher-order functions, such as algorithms or functions that accept function pointers or function objects. This enables more flexibility and adaptability in designing algorithms and behavior.

4. ****Avoiding code duplication****: Lambda functions help avoid code duplication by allowing you to define small, specialized functions inline where they are needed. This eliminates the need to write separate, repetitive function definitions for specific use cases.

5. ****Improved code organization****: By defining functions inline using lambda functions, you can better organize and structure your code by keeping related logic closer together. This can improve code maintainability and make it easier to understand and modify.

6. ****Local scoping****: Lambda functions have their own local scope, allowing you to declare and use variables that are specific to the lambda function without polluting the surrounding scope. This can help reduce naming conflicts and make the code more modular.

7. ****Code reusability****: While lambda functions are often used as one-time, throwaway functions, they can also be stored in variables or passed around, making them reusable in different parts of the codebase. This promotes code modularity and reusability.

Overall, lambda functions in C++11 provide a powerful tool for concise and flexible coding, enabling the creation of compact and specialized functions within the scope where they are needed.

10. Can you provide an example of using a lambda function in C++11 to sort a vector of integers?

Certainly! Here's an example that demonstrates the usage of a lambda function in C++11 to sort a vector of integers:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> numbers = {5, 2, 8, 1, 9, 3};

    // Sorting the vector using a lambda function
    std::sort(numbers.begin(), numbers.end(), [](int a, int b) {
    return a < b;
    });

    // Printing the sorted vector
    for (const auto& num : numbers) {
    std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

In this example, the `std::sort` algorithm is used to sort a vector of integers (`numbers`). The lambda function is passed as the third argument to `std::sort`, which determines the sorting order. The lambda function compares two integers (`a` and `b`) and returns `true` if `a` is less than `b`, indicating that `a` should appear before `b` in the sorted sequence.

The output of this code will be: `1 2 3 5 8 9`, showing the sorted vector in ascending order.

Please note that these questions are focused specifically on C++11 and lambda functions.