复习材料：函数式编程与可调用类型 (第五章)

1. C++对函数式编程的支持

C++ 提供了多种可调用类型，使得函数式编程变得更加灵活。以下是支持的几种主要方式：

- **Lambda 表达式**：可以使用 Lambda 在代码中内联创建可调用对象，Lambda 表达式通常用于定义短小的、只需要使用一次的函数。
- **`std::function`**： `std::function` 是一个通用的函数包装器，允许存储任意可调用对象，如 Lambda、函数指针、仿函数等。
- **函数指针**：函数指针用于保存函数的地址，可以用来动态地调用不同的函数。
- **组合与绑定**：C++ 支持函数的组合和参数绑定。可以通过 `std::bind` 将函数与参数绑定，并生成新的可调用对象；可以通过函数组合将两个或多个函数组合成一个新的函数。

2. 函数指针的命名语法

函数指针 是指向函数的指针，它可以用来存储函数地址并在需要时调用。它的语法比较复杂，尤其是在处理返回类型和参数列表时。

函数指针的基本语法：

```
返回类型 (*指针名)(参数列表);
```

例如：

```
std::size_t (*strlen_ptr)(const char*) = &std::strlen;
```

解释：

- `std::size_t` 是返回类型。
- `(*strlen_ptr)` 是指针名，表示这个指针指向一个接受 `const char*` 类型参数的函数。
- `&std::strlen` 表示将标准库中的 `strlen` 函数的地址赋给函数指针 `strlen_ptr` 。

之后你可以通过 `strlen_ptr` 来调用该函数：

```
std::size_t length = strlen_ptr("Hello, world!");
```

示例：
定义一个函数指针指向返回 `int` 且接受两个 `int` 参数的函数：

```
int (*func_ptr)(int, int);
```

函数指针不仅可以用于 C 风格函数，还可以指向 Lambda 表达式或者其他可调用对象。

3. Lambda 表达式的语法

Lambda 表达式 是 C++11 引入的功能，用于定义匿名函数。Lambda 可以捕获外部作用域的变量并在内部使用，语法如下：

```
[捕获列表](参数列表) -> 返回类型 { 函数体 };
```

- **捕获列表**：捕获外部作用域的变量，可以按值或按引用捕获。
- **参数列表**：Lambda 的输入参数。
- **返回类型**：可以显式指定，也可以省略，让编译器推导。
- **函数体**：Lambda 的具体实现。

示例：

```
auto lambda = [](int x) { return x * 2; };
```

如果 Lambda 捕获外部变量，可以这样做：

```
int factor = 2;
auto multiply = [factor](int x) { return x * factor; };
```

调用：

```
int result = multiply(5);  // result = 10
```

4. 使用 `std::function` 定义 Lambda

`std::function` 是一个模板类，它可以用于存储任意类型的可调用对象，如函数指针、Lambda 表达式等。相比直接使用 `auto`， `std::function` 提供了更多的灵活性，尤其是在需要将 Lambda 作为参数或返回值时。

语法:

```
std::function<返回类型(参数列表)> lambda = [](参数列表) -> 返回类型 { 函数体 };
```

示例:

```
std::function<int(int)> square = [](int x) -> int {
    return x * x;
};
```

解释:

- `std::function<int(int)>` 定义了一个接受 `int` 类型参数并返回 `int` 类型的 Lambda。
- Lambda 内部返回传入参数的平方。

使用:

```
int result = square(5);  // result = 25
```

在一些复杂场景中， `std::function` 可以为 Lambda 提供明确的类型定义和更多灵活性:

```
std::function<float(float)> sin_lambda = [](float x) { return std::sin(x); };
std::function<float(float)> cos_lambda = [](float x) { return std::cos(x); };
```

在这种情况下， `std::function` 可以被传递到需要函数对象的地方，比如函数组合或者绑定。

5. 组合与绑定

C++ 提供了函数组合（composition）和函数绑定（binding）功能。可以使用 `std::bind` 函数将某些参数与函数绑定，生成一个新的函数对象，或者将多个函数组合起来。

示例：绑定参数

```
template <class F, class G>
auto compose(F f, G g)
{
    return [f, g](auto x)
    { return f(g(x)); };
}

std::function<float(float)> composed = compose(sin_lambda, cos_lambda);
auto binded = std::bind(composed, 3.14159);
```

这里，我们将两个 Lambda 表达式 `sin_lambda` 和 `cos_lambda` 组合成一个新的函数，然后通过 `std::bind` 绑定一个具体的参数。

调用：

```
float result = binded();  // 计算绑定参数后的结果
```

考题示例

考题示例 1：
"How do you declare a function pointer to a function that takes a `const char*` and returns `std::size_t` ?"

答案：

```
std::size_t (*func_ptr)(const char*) = &std::strlen;
```

考题示例 2：
"What is the syntax to define a lambda expression in C++?"

答案：

```
[捕获列表](参数列表) -> 返回类型 { 函数体 };
```

考题示例 3:

"What is the difference between using `auto` and `std::function` for defining a lambda expression?"

答案:

- 使用 `auto` 可以让编译器自动推导 Lambda 的类型，效率更高，但类型是匿名的，只适合局部使用。
- 使用 `std::function` 定义 Lambda 可以使其类型明确，适合传递和保存 Lambda 表达式，但会有轻微的性能开销。

When would you use callable objects (lambda, bind, functional) in C++?

1. **Lambda Expressions**:
   - **When you need short, inline functions**: Lambda expressions are perfect for situations where you need a small, one-time function, especially in contexts like algorithms (`std::for_each` , `std::sort` , etc.) or event handling (e.g., GUI callbacks).
   - **For passing functions to algorithms**: If you want to pass a custom operation to an STL algorithm like `std::sort` , you can define the operation inline using a lambda expression.
   - **Capturing variables**: When you need to capture variables from the surrounding scope, lambdas allow you to capture and work with those variables in the function body.

   Example: Sorting with a custom comparator:

   ```
   std::vector<int> v = {1, 4, 2, 8, 5};
   std::sort(v.begin(), v.end(), [](int a, int b) { return a > b; });
   ```

2. **`std::bind`** :
   - **When you need partial function application**: `std::bind` is useful when you want to bind certain parameters of a function to fixed values and create a callable object that you can use later with fewer parameters. It essentially "freezes" certain arguments of a function and allows you to call the function later with fewer parameters.
   - **For compatibility with legacy code**: In some cases, older APIs may expect certain function signatures, and `std::bind` allows you to adapt new functions to fit these signatures by binding parameters in advance.

   Example: Binding a function to specific arguments:

```cpp
#include <functional>
#include <iostream>

void print_sum(int a, int b) {
    std::cout << a + b << std::endl;
}

int main() {
    auto bound_func = std::bind(print_sum, 2, std::placeholders::_1);
    bound_func(5);  // Outputs: 7 (binds 2 to the first argument)
    return 0;
}
```

3. **`std::function`** :
   - **When you need type erasure for callable objects**: `std::function` allows you to store callable objects with different types (such as lambdas, function pointers, or `std::bind` objects) in a single variable. It's especially useful when you need to pass or store functions in a polymorphic way, without knowing the exact type of the callable object at compile time.
   - **For storing callbacks**: In event-driven programming (e.g., GUIs, game engines), you often need to store callback functions for later use. `std::function` is ideal for this because it can store any callable object that matches the specified function signature.

   Example: Using `std::function` to store different callable types:

```cpp
#include <functional>
#include <iostream>

int add(int a, int b) {
    return a + b;
}

int main() {
    std::function<int(int, int)> func;

    // Assign a lambda to std::function
    func = [](int a, int b) { return a * b; };
    std::cout << func(2, 3) << std::endl;  // Output: 6

    // Assign a function pointer to std::function
    func = add;
    std::cout << func(2, 3) << std::endl;  // Output: 5

    return 0;
}
```

Why is it important?

1. **Improved Code Readability and Maintainability**:
   - **Lambda expressions** make the code more concise and expressive. Instead of writing separate, named functions for simple operations, lambdas allow you to define small, throwaway functions inline. This reduces code clutter and makes the code more readable, especially when working with algorithms.
   - `std::function` provides a consistent interface for handling callable objects, whether they are lambdas, function pointers, or `std::bind` objects. This leads to more maintainable code because you don't need to worry about the exact type of callable object being used.
2. **Functional Programming Capabilities**:
   - These features bring functional programming techniques to C++. You can pass functions as arguments, return them from other functions, and store them in data structures, enabling a higher level of abstraction and flexibility in your code.
   - **Higher-order functions**: Functions like `std::sort` or custom algorithms can take other functions (lambdas) as parameters, which allows you to write more generic and reusable code.

3. **Flexibility and Reusability**:
    - `std::bind` allows for partial application, making your functions more flexible and reusable by pre-fixing certain arguments and leaving the rest to be filled in later. This is helpful in cases where you need to adapt existing functions to fit new contexts without modifying the original function.
    - **Capturing variables** in lambdas enables complex closures, where lambdas maintain access to the variables in the surrounding scope. This is particularly useful in event-driven programming and asynchronous execution (e.g., capturing local state in concurrent tasks).
4. **Type Erasure and Polymorphism**:
    - `std::function` provides a powerful form of type erasure. It allows you to handle any callable object uniformly, regardless of its actual type. This makes it invaluable for frameworks, libraries, or APIs that need to store or manipulate user-defined functions without knowing their exact types in advance.
5. **Callbacks and Event Handling**:
    - In many modern C++ applications, such as GUIs, game engines, or networked applications, handling events and callbacks is crucial. **Lambdas** and `std::function` are essential tools for this, as they enable you to store and pass around functions that can later be executed in response to specific events.
6. **Compatibility and Adaptation**:
    - `std::bind` and `std::function` can help you interface between old-style function pointers and modern C++ callable objects. This is particularly important when dealing with legacy code or when integrating C++ code with external libraries or frameworks that require specific function signatures.

In summary:

- **When**: You would use these tools in scenarios where you need callable objects, function pointers, callbacks, or operations on functions (partial application, type erasure, etc.).
- **Why**: They are important because they enhance code flexibility, reusability, and readability, allow for functional programming paradigms, provide an easy way to handle stateful or state-agnostic functions, and make event handling or callback-based programming easier and more expressive.