第九章的内容涉及到成员变量指针和成员函数指针的使用。在C++中，成员变量指针和成员函数指针提供了一种方法来访问和操作类对象中的特定成员，而不必直接使用对象。以下是详细的讲解：

1. 指向成员变量的指针（Pointers to Data Members）

用途：
成员变量指针用于在运行时访问类的特定成员变量，尤其是在我们希望通过同样的代码操作不同的对象时，它为我们提供了一种更灵活的方法。例如，在一个存储了学生信息的类中，可以使用成员变量指针遍历和访问学生的姓氏、名字、学号等不同的数据成员。

语法：
要声明一个指向类成员变量的指针，可以使用以下语法：

```
class MyClass {
public:
    int x;
    double y;
};

// 声明一个指向MyClass成员x的指针
int MyClass::* ptrX = &MyClass::x;
```

- `MyClass::* ptrX`： 表示这个指针 `ptrX` 指向 `MyClass` 类的一个 `int` 类型成员变量。
- `&MyClass::x`： 是获取成员变量 `x` 的地址。

使用方法：
要通过对象访问成员变量，可以通过以下方式解引用成员变量指针：

```
MyClass obj;
obj.x = 10;
int MyClass::* ptrX = &MyClass::x;
std::cout << obj.*ptrX << std::endl;  // 输出：10
```

此处， `obj.*ptrX` 解引用指针并获取对象 `obj` 的成员变量 `x` 的值。

意义：
成员变量指针提供了一种可以在运行时动态选择访问哪个数据成员的方式，这种灵活性在很多动态编程场景中都非常有用。

2. 指向成员函数的指针（Pointers to Member Functions）

用途:
成员函数指针用于在运行时调用类的特定成员函数，而不是在编译时确定调用哪个函数。这在需要处理不同类的对象时特别有用。它允许我们动态地选择要调用的成员函数。

语法:
要声明一个指向成员函数的指针，可以使用以下语法：

```
class MyClass {
public:
    void myFunc() {
        std::cout << "MyClass::myFunc" << std::endl;
    }
};

// 声明一个指向MyClass成员函数myFunc的指针
void (MyClass::* ptrFunc)() = &MyClass::myFunc;
```

- `void (MyClass::* ptrFunc)()`：表示 `ptrFunc` 是一个指向 `MyClass` 类中返回类型为 `void` 的成员函数的指针。
- `&MyClass::myFunc`：是获取成员函数 `myFunc` 的地址。

使用方法:
通过对象来调用成员函数指针：

```
MyClass obj;
void (MyClass::* ptrFunc)() = &MyClass::myFunc;
(obj.*ptrFunc)();  // 输出：MyClass::myFunc

void (MyClass::* FunPtrFunc()) () {
    return &MyClass::myFunc;
}
```

- `obj.*ptrFunc` 解引用函数指针并调用 `myFunc`。

意义:
成员函数指针让我们可以在运行时根据条件选择要调用的成员函数，而不是在编译时确定函数调用。这种灵活性为实现如回调函数、事件处理程序等模式提供了可能。

示例

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

class Calculator {
public:
    void add(int a, int b) {
        cout << "Sum: " << a + b << endl;
    }

    void subtract(int a, int b) {
        cout << "Difference: " << a - b << endl;
    }

    // 返回指向add或subtract的指针
    void (Calculator::*getOperation(bool isAdd)) (int, int) {
        return isAdd ? &Calculator::add : &Calculator::subtract;
    }
};

int main() {
    Calculator calc;

    // 通过getOperation函数返回成员函数指针
    void (Calculator::*opPtr)(int, int) = calc.getOperation(true);
    (calc.*opPtr)(10, 5);  // 调用add函数

    opPtr = calc.getOperation(false);
    (calc.*opPtr)(10, 5);  // 调用subtract函数

    return 0;
}
```

3. 成员指针的具体场景应用

在一些复杂场景中，成员变量指针和成员函数指针可以与函数模板结合使用，使代码更加灵活。例如，使用模板函数处理不同的类对象和成员：

```
template<typename T, typename M>
void accessMember(T& obj, M T::* member) {
    std::cout << "Member value: " << obj.*member << std::endl;
}

MyClass obj;
obj.x = 42;
accessMember(obj, &MyClass::x);  // 输出：Member value: 42
```

这种模式在处理大规模对象的动态访问和操作时非常有用，特别是在需要统一处理不同类的对象和成员时。

要更好地理解显示成员指针以及返回指向成员函数指针的函数，我们可以从一些实际应用场景出发，逐步展示它们的用法和好处。

4. 成员函数指针静态与动态赋值：

在基类与派生类的多态性设计中，成员函数指针可以被静态或动态地赋值。

静态赋值（编译时确定）

静态赋值通常是直接把基类的成员函数指针赋值为某个具体的函数。如下所示：

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

class BaseClass {
public:
    void display() {
        cout << "BaseClass display" << endl;
    }
};

int main() {
    // 静态绑定：直接使用BaseClass的成员函数指针
    void (BaseClass::*funcPtr)() = &BaseClass::display;

    BaseClass base;
    (base.*funcPtr)();  // 调用BaseClass的display函数

    return 0;
}
```

输出：

```
BaseClass display
```

动态赋值（运行时确定）

动态赋值通过派生类对象来动态决定调用哪个版本的成员函数，特别是在多态的场景下。

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

class BaseClass {
public:
    // 虚函数，允许派生类重写
    virtual void display() {
        cout << "BaseClass display" << endl;
    }
};

class DerivedClass : public BaseClass {
public:
    // 重写基类的 display 函数
    void display() override {
        cout << "DerivedClass display" << endl;
    }
};

int main() {
    BaseClass base;
    DerivedClass derived;

    // 定义指向成员函数的指针，指向基类的 display 函数
    void (BaseClass::*funcPtr)() = &BaseClass::display;

    // 通过基类对象调用 display 函数
    (base.*funcPtr)();  // 输出: BaseClass display

    // 通过派生类对象调用 display 函数
    (derived.*funcPtr)();  // 输出: DerivedClass display （动态绑定）

    // 使用基类指针指向派生类对象
    BaseClass* ptr = &derived;
    (ptr->*funcPtr)();  // 输出: DerivedClass display （动态绑定）

    return 0;
}
```

输出:

```
BaseClass display
DerivedClass display
DerivedClass display
```

尽管 `ptr` 的静态类型是 `BaseClass*`，但由于 `display` 函数是虚函数，最终调用的是 `DerivedClass` 的 `display` 函数。这就是多态性在运行时的表现。

4. 设计的好处：

成员指针（尤其是函数指针）与返回函数指针的设计模式有很多好处：

- **多态性支持**：通过成员函数指针，程序可以根据不同的对象类型在运行时调用不同的函数，而不需要在编译时确定。
- **灵活性**：可以动态决定调用哪个成员函数，极大地增加了程序的灵活性和动态性，特别是在实现类似事件驱动或回调机制时。
- **封装与抽象**：返回函数指针的设计模式常见于反射机制或元编程中，这样可以避免在程序中显式地列出所有函数，提供了一种更加抽象的调用方式。

5. 成员函数指针在PPT中的应用：

- 正如你所分享的 PPT 中提到的，指向成员函数的指针让我们能够延迟调用成员函数，并且能够结合 `std::function` 和 `std::mem_fn` 将成员函数对象化。这不仅可以方便地构建回调系统，还能实现类似策略模式（Strategy Pattern）中的函数选择。

```
function<void()> func1 = bind(&BaseClass::display, &base);
function<void()> func2 = bind(&BaseClass::display, &derived);
function<void()> func3 = bind(&DerivedClass::display, &derived);

function<void(BaseClass &)> func1 = mem_fn(&BaseClass ::display);
function<void(BaseClass &)> func2 = mem_fn(&BaseClass ::display);
function<void(DerivedClass &)> func3 = mem_fn(&DerivedClass ::display);

func1(base);
func2(derived);
func3(derived);
```

- mem_fn(&BaseClass::display)：std::mem_fn 是一种将成员函数转换为函数对象的工具。它能够生成一个可调用的对象，该对象可以传递给 std::function，并能被调用。这里 mem_fn(&BaseClass::display) 获取的是 BaseClass 类的成员函数 display 的指针。

总结：成员变量指针和成员函数指针提供了一种在运行时动态访问和调用类成员的机制。这使得C++程序具有更高的灵活性，能够处理多态、回调等高级编程需求。理解并熟练使用这些指针是掌握C++编程中面向对象和动态特性的关键。

As you point out, they are different tools and the “pointer to data member” is used rarely. One thing to keep in mind is that an accessor (getter/setter) acts directly on a member of a specific object whereas the pointer to data member separates the member selection step (am I getting the first name, last name, street address?) from the actual object access (which person is it?). An example where that may be used is data binding in a user interface. When the UI is being set up, the various controls (drop-down lists, input fields, etc.) would be bound to members of the class – not members of a specific object, since those may not even exist yet. Later the UI would need to actually render a view of an object, so it would dereference the pointers to data members using the given object.