**C++学习笔记——函数与可调用对象**

**1. 函数声明与定义**

标准的C++开发流程是：将函数声明放在头文件中，而将函数定义（实现）放在源文件（.cpp/.cxx）中，并且源文件包含头文件。

编译时，编译器会将函数实现的代码复制到头文件当中。

部分较为简短的函数（例如getter/setter函数），会直接把声明和定义都写在头文件中，这种情况称为内联（inline）函数。

**2. 函数的默认参数**

除了众所周知的默认参数规则外，特别需要注意的是：

(1) 函数的声明和实现中都可以指定默认参数，但二者不能同时指定。一般情况下，最佳编程实践是，在函数声明中写默认参数，而函数实现中不写。

少数情况下，会选择声明和实现中都不写默认参数，而在调用之前临时声明。

(2) 绝对不要让派生类重写基类虚函数的默认参数。如果一定要这样做，那么代码可以通过编译，但这种重写不会产生任何效果，最后生效的仍然是基类函数的默认参数。这样的代码与预期不符，会增加维护和理解难度。

示例代码：

图形用户界面, 文本, 应用程序

描述已自动生成



原因很简单：默认参数需要在编译时确定。

重写基类虚函数的默认参数的初衷显然是，希望能够触发动态绑定，根据实际类型，应用每个派生类的函数的不同默认参数。然而动态绑定发生在运行时，而默认参数需要在编译时确定。编译时，编译器只会获取基类函数的默认参数，所以任何派生类试图重写默认参数的行为，都是无效的。

**3. 可调用对象（callable objects）**

在C++中，"可调用对象（callable objects）"是一个可以像函数一样被调用的对象。包括：

(1) 函数：这是最基本的callable objects。

(2) 函数指针：指向函数的指针也是callable objects。例如：

void (\*foo\_ptr)() = &foo; // 后续可以通过foo\_ptr()来调用。

(3) 函数对象（Functor）：重载了operator()的类的对象。例如：

struct Foo {

void operator()() { }

};

Foo foo;

foo(); // Call operator()

(4) Lambda表达式：C++11引入的一种特殊的函数对象。例如：

auto lambda = []() { };

lambda();。

(5) 成员函数指针：指向类的成员函数的指针。需要通过对象来调用。例如：

struct Foo {

void bar() { }

};

void (Foo::\*bar\_ptr)() = &Foo::bar;

Foo foo;

(foo.\*bar\_ptr)(); // Call member function

(6) std::function：C++11引入的一种通用、可复制的函数包装器。它可以存储任何类型的callable objects。例如：

std::function<void()> func = foo; // foo can be any callable objects

func(); // Call the function

**4. 类的成员函数声明与实现中的关键字**

对于类的非内联成员函数，只写在函数声明而不写在实现中的关键字：

- virtual

- override

- final

在函数声明和实现中都要写的关键字：

- constexpr

- const

- noexcept

- throw

注意：在现代C++编译器中，throw()和noexcept等价，均表示不抛出异常。编译器在预处理时，会将二者统一成noexcept。

**5. 友元函数**

一个类的友元（friend）函数是一个非成员函数，但可以访问该类的private成员。友元函数通常在类内声明，在类外实现。

友元函数的作用：

(1) 在不需要进行代码重构的前提下，让一个类对特定的非成员函数，开放private成员。比起精心重构代码，使用友元函数大大降低了开发成本。

(2) 友元函数可以直接访问私有成员变量，这比通过外部接口访问内部成员（如getter函数）的开销更低。但这点收益通常抵不上破坏封装性带来的代价。

总之，**友元函数以部分牺牲类的封装性，以及降低代码的可维护性为代价，换取了开发的便利性和更高效的访问。**之所以说是“部分”，是因为类的封装性只对有限个友元函数失效，而对其他外部函数来说，封装性仍然存在。

开发实践中，应当做到非必要不使用友元函数。