Effective C++ 建议总结

视 C++ 为一个语言联邦

- **解释**：C++是一个多范式编程语言，其主要包括C、面向对象的C++、模板C++（Template C++）和STL（标准模板库）。每个子语言有自己的规则。
- **例子**：使用STL容器来管理资源而非手动管理指针。

用编译器替换预处理器

- **解释**：推荐使用 `const`、`enum` 和 `inline` 替代 `#define`。
- **例子**：使用 `const` 定义常量，而不是 `#define`。

尽可能使用 const

- **解解释**：const 可以提高代码的可读性和健壮性。
- **例子**：将不会修改的成员函数声明为 `const`。

确保对象被使用前已被初始化

- **解释**：构造时初始化（通过构造函数或初始化列表）比默认构造后赋值更高效。
- **例子**：使用初始化列表初始化成员变量。

了解 C++ 默默编写并调用哪些函数

- **解释**：C++ 编译器会为类自动生成默认构造函数、拷贝构造函数、拷贝赋值操作符和析构函数。
- **例子**：定义自己的拷贝构造函数和拷贝赋值操作符，以控制对象的复制行为。

若不想使用编译器自动生成的函数，就应该明确拒绝

- **解释**：如果你不需要或不希望编译器自动生成的成员函数，应该将它们声明为 `private` 并且不提供实现。
- **例子**：将拷贝构造函数和拷贝赋值操作符声明为私有，防止自动生成。

为多态基类声明 virtual 析构函数

- **解释**：如果一个类有任何 `virtual` 函数，它应该有一个 `virtual` 析构函数。
- **例子**：

```
class Base {
public:
    virtual ~Base() {}
};
```

别让异常逃离析构函数

- **解释**：析构函数应该捕获并处理异常，而不是抛出它们。如果析构函数发出异常，应该终止程序或吞下异常。
- **例子**：

```
class MyClass {
public:
    ~MyClass() {
        try {
            // 清理代码
        } catch(...) {
            // 处理异常
        }
    }
};
```

绝不在构造和析构过程中调用 virtual 函数

- **解释**：在构造和析构期间，对象的类型会被视为构造或析构的那个类类型，而不是派生类类型。
- **例子**：

```
class Base {
public:
    Base() { callFunc(); }
    virtual void callFunc() {}
};

class Derived : public Base {
public:
    void callFunc() override { /* ... */ }
};
// Derived 的构造函数调用 Base::callFunc，而不是 Derived::callFunc
```

令 operator= 返回一个 reference to *this

- **解释**：赋值操作符应该返回一个指向当前对象的引用，以支持连锁赋值。
- **例子**：

```
MyClass& operator=(const MyClass& rhs) {
    if (this == &rhs) return *this;
    // 赋值操作
    return *this;
}
```

在 operator= 中处理 “自我赋值”

- **解释**：赋值操作符应该能正确处理自我

赋值的情况。

- **例子**：如上。

赋值对象时应确保复制 “对象内的所有成员变量” 及 “所有 base class 成分”

- **解释**：确保拷贝所有成员变量，并且在派生类的赋值操作符中调用基类的赋值操作符。
- **例子**：

```
class Derived : public Base {
    int x;
public:
    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs); // 调用基类赋值操作符
        x = rhs.x; // 复制所有成员
        return *this;
    }
};
```

以对象管理资源

- **解释**：使用对象来管理资源，例如使用智能指针管理内存。
- **例子**：使用 `std::unique_ptr` 或 `std::shared_ptr` 而不是裸指针。

在资源管理类中小心 copying 行为

- **解释**：如果一个类管理资源（如动态分配的内存），需要特别注意其拷贝行为。
- **例子**：使用智能指针的拷贝语义来管理资源，或自定义拷贝行为。

在资源管理类中提供对原始资源的访问

- **解释**：资源管理类应该提供方法来访问它所管理的资源。
- **例子**：智能指针提供 `get()` 方法来访问原始指针。

成对使用 new 和 delete

- **建议**：如果在 `new` 中使用了 `[]`，则在对应的 `delete` 中也应该使用 `[]`。
- **例子**：

```
int* array = new int[10];  // 使用 new[]
// ... 使用 array
delete[] array;            // 使用 delete[]
```

独立语句置入智能指针

- **建议**：在将 `new` 表达式结果存入智能指针时，使用独立的语句，以防编译器优化导致的资源泄漏。
- **例子**：

```
std::shared_ptr<int> smartPtr(new int(42)); // 独立语句
```

接口容易正确使用，不易被误用

- **建议**：设计接口时，应易于正确使用且不易被误用。
- **例子**：创建类型以表示特定的概念，如 `UserID` 替代 `int`。

设计 class 犹如设计 type

- **建议**：在设计类时，考虑对象的创建、销毁、初始化、赋值等。
- **例子**：为类定义合适的构造函数、析构函数和赋值操作符。

宁以 pass-by-reference-to-const 替换 pass-by-value

- **建议**：除非类型小且易于复制（如内置类型），否则最好以传递常量引用的方式传递参数。
- **例子**：

```
void process(const Widget& w);
```

必须返回对象时，别妄想返回其 reference

- **建议**：绝不要返回局部栈对象、堆分配对象或局部静态对象的引用或指针。
- **例子**：

```
Widget getWidget() {
    Widget w;
    return w; // 返回一个副本
}
```

将成员变量声明为 private

- **建议**：为了封装、一致性和精确控制，将成员变量声明为私有。
- **例子**：使用 `private` 成员变量并提供公共访问函数。

宁以 non-member、non-friend 替换 member 函数

- **建议**：非成员、非友元函数可提高封装性、弹性和扩展性。
- **例子**：为类实现非成员 `operator<<` 来支持输出。

若所有参数皆须要类型转换，请为此采用 non-member 函数

- **建议**：如果所有参数都需要类型转换，最好使用非成员函数。
- **例子**：实现非成员函数以实现两种不同类型对象的相互转换。

考虑写一个不抛异常的 swap 函数

- **建议**：为自定义类型提供一个不抛异常的 `swap` 函数。
- **例子**：

```
class Widget {
public:
    void swap(Widget& other) noexcept {
        // 交换成员
    }
};
```

尽可能延后变量定义式的出现时间

- **建议**：为了清晰和效率，尽量在需要变量时才定义它。
- **例子**：

```
for (int i = 0; i < n; ++i) {
    // 使用 i
}
```

尽量少做转型动作

- **建议**：尽量避免转型，如果必须转型，优先使用 C++ 风格的转型。
- **例子**：使用 `static_cast`、`dynamic_cast`、`const_cast` 和 `reinterpret_cast` 而非 C 风格转型。

避免使用 handles 指向对象内部

- **建议**：避免使用指针、引用或迭代器指向对象内部，以增强封装性和安全性。
- **例子**：返回对象副本而不是对象内部数据的引用。

为“异常安全”而努力

- **建议**：编写异常安全的代码，确保即使发生异常也不会泄露资源或破坏数据结构。
- **例子**：使用 RAII（资源获取即初始化）模式管理资源。

透彻了解 inlining 的里里外外

- **建议**：理解 `inline` 的行为，它是一种请求而非命令，其效果取决于编译器。
- **例子**：合理使用 `inline` 函数，避免过度使用导致的代码膨胀。

将文件间的编译依存关系降至最低

- **建议**：使用前向声明和将定义与声明分离，以减少编译依赖。
- **例子**：在头文件中只包含必要的类声明，而将定义放在源文件中。

确定你的 public 继承塑模出 is-a 关系

- **建议**：确保公有继承表示“是一个”的关系，派生类应能够替代基类。
- **例子**：

```
class Bird : public Animal { /* ... */ };
```

避免遮掩继承而来的名字

- **建议**：使用 `using` 声明或转交函数，以避免派生类遮掩基类的名字。
- **例子**：

```
class Derived : public Base {
public:
    using Base::someFunction;
};
```

区分接口继承和实现继承

- **建议**：区分仅继承接口（使用纯虚函数）和继承接口及其默认实现（使用非纯虚函数）。
- **例子**：

```
class Base {
public:
    virtual void func() = 0; // 纯虚函数
};
```

考虑 virtual 函数以外的其他选择

- **建议**：考虑使用其他技术如模板方法模式、函数指针等替代 virtual 函数。
- **例子**：

```
class Strategy {
public:
    virtual void execute() = 0;
};
```

绝不重新定义继承而来的 non-virtual 函数

- **建议**：不要在派生类中重新定义非虚函数。

绝不重新定义继承而来的缺省参数值

- **建议**：不要在派生类的虚函数中改变默认参数值。

通过复合塑模 has-a 或 “根据某物实现出”

- **建议**：使用组合来实现 has-a 关系或基于某物的实现。
- **例子**：

```
class Car {
    Engine engine; // Car has-a Engine
};
```

明智而审慎地使用 private 继承

- **建议**：私有继承意味着“根据某物实现出”，应谨慎使用。
- **例子**：

```
class Timer : private Clock { /* ... */ };
```

明智而审慎地使用多重继承

- **建议**：多重继承比单一继承更复杂，应谨慎使用。

了解隐式接口和编译期多态

- **建议**：理解类和模板如何提供接口和多态。

了解 typename 的双重意义

- **建议**：在模板编程中正确使用 `typename` 关键字。

学习处理模板化基类内的名称

- **建议**：在派生模板类中正确处理基类模板的成员名称。

将与参数无关的代码抽离 templates

- **建议**：将模板中独立于类型参数的代码提取出来，避免不必要的代码重复。

运用成员函数模板接受所有兼容类型

- **建议**：使用成员函数模板来接受所有兼容类型的参数。
- **例子**：

```
class Widget {
public:
    template<typename T>
    void process(const T& arg);
};
```

需要类型转换时请为模板定义非成员函数

- **建议**：对于模板类相关的支持所有参数隐式类型转换的函数，定义为类内的友元函数。
- **例子**：

```
template <typename T>
class Rational {
    // ...
public:
    friend const Rational operator*(const Rational& lhs, const Rational&
rhs) {
        // ...
    }
};
```

使用 traits classes 表现类型信息

- **建议**：使用特征类（traits classes）在编译期提供类型相关信息。
- **例子**：

```
template <typename T>
struct numeric_limits {
    static const bool is_specialized = false;
    // ...
};

template <>
struct numeric_limits<int> {
    static const bool is_specialized = true;
    static const int min() { return INT_MIN; }
    static const int max() { return INT_MAX; }
};
```

认识 template 元编程

- **建议**：模板元编程可以在编译期执行复杂的计算，从而提高运行时效率。

了解 new-handler 的行为

- **建议**：使用 `std::set_new_handler` 定义内存分配失败时的处理逻辑。
- **例子**：

```
void outOfMemHandler() {
    std::cerr << "无法分配内存" << std::endl;
    std::abort();
}

std::set_new_handler(outOfMemHandler);
```

了解 new 和 delete 的合理替换时机

- **建议**：在特定情况下替换 `new` 和 `delete` 以提高性能或获得特殊行为。

编写 new 和 delete 时需固守常规

- **建议**：自定义 `operator new` 和 `operator delete` 时，遵守一定的规则，如处理0字节申请，对于 `operator delete` 忽略空指针等。

写了 placement new 也要写 placement delete

- **建议**：如果提供了 placement `new`，也应提供对应的 placement `delete` 以防止内存泄漏。

不要轻忽编译器的警告

- **建议**：认真对待编译器的警告，它们可以帮助你发现代码中的潜在问题。

熟悉包括 TR1 在内的标准程序库

- **建议**：熟悉 C++ TR1（Technical Report 1）和 C++11 标准库，它们提供了许多有用的功能和工具。

熟悉 Boost（准标准库）

- **建议**：熟悉 Boost 库，它提供了广泛的 C++ 库，许多功能最终成为 C++ 标准的一部分。

More Effective C++ 建议总结

仔细区别 pointers 和 references

- **建议**：使用指针时可能更改指向，而引用则保持恒定。选择合适的类型以表达意图。
- **例子**：

```
void processValue(int& ref);  // 使用引用
void processPointer(int* ptr); // 使用指针
```

最好使用 C++ 转型操作符

- **建议**：使用 `static_cast`、`const_cast`、`dynamic_cast`、`reinterpret_cast` 替代 C 风格的转型。
- **例子**：

```
const_cast<int&>(x); // 去除 const 属性
dynamic_cast<Derived*>(basePtr); // 安全向下转型
```

避免多态方式处理数组

- **建议**：多态和指针算术不应混用，尤其在处理数组时。

非必要不提供 default constructor

- **建议**：只在需要的时候提供默认构造函数，避免无意义的初始化。

对定制的类型转换函数保持警觉

- **建议**：谨慎使用类型转换函数，使用 `explicit` 关键字避免非预期的类型转换。
- **例子**：

```
class Widget {
public:
    explicit Widget(int initVal);
    // ...
};
```

区别 increment/decrement 操作符的前置和后置形式

- **建议**：前置形式返回引用，后置形式返回值。
- **例子**：

```
class Counter {
public:
    Counter& operator++();    // 前置
    Counter operator++(int);  // 后置
};
```

千万不要重载 &&，|| 和 , 操作符

- **建议**：避免重载这些操作符，以免改变它们原有的语义。

了解 new 和 delete 的不同意义

- **建议**：理解 `new` 和 `delete` 操作符的不同形式和使用场景。

利用 destructors 避免泄漏资源

- **建议**：在析构函数中释放资源，确保异常安全。
- **例子**：

```
class ResourceHolder {
public:
    ~ResourceHolder() {
        delete resource;
    }
private:
    ResourceType* resource;
};
```

在 constructors 内阻止资源泄漏

- **建议**：使用智能指针管理构造过程中可能分配的资源，以避免异常导致的泄漏。
- **例子**：

```
class Widget {
public:
    Widget() : resource(new ResourceType) {}
private:
    std::unique_ptr<ResourceType> resource;
};
```

禁止异常流出 destructors 之外

- **建议**：确保析构函数不抛出异常。

了解异常处理的成本

- **建议**：理解异常处理的成本，仅在必要时使用。

谨记 80-20 法则

- **建议**：大多数性能问题通常由代码的一小部分造成，应该重点优化这些部分。

考虑使用 lazy evaluation

- **建议**：延迟计算的执行，直到真正需要结果。

分期摊还预期的计算成本

- **建议**：当计算结果经常被需要时，可以提前计算并存

储这些结果。

运用成员函数模板接受所有兼容类型

- **建议**：使用成员函数模板来接受所有兼容的类型，以提高类的通用性。
- **例子**：

```
template <typename T>
class SmartPtr {
public:
    SmartPtr(T* realPtr);
    // ...
};
```