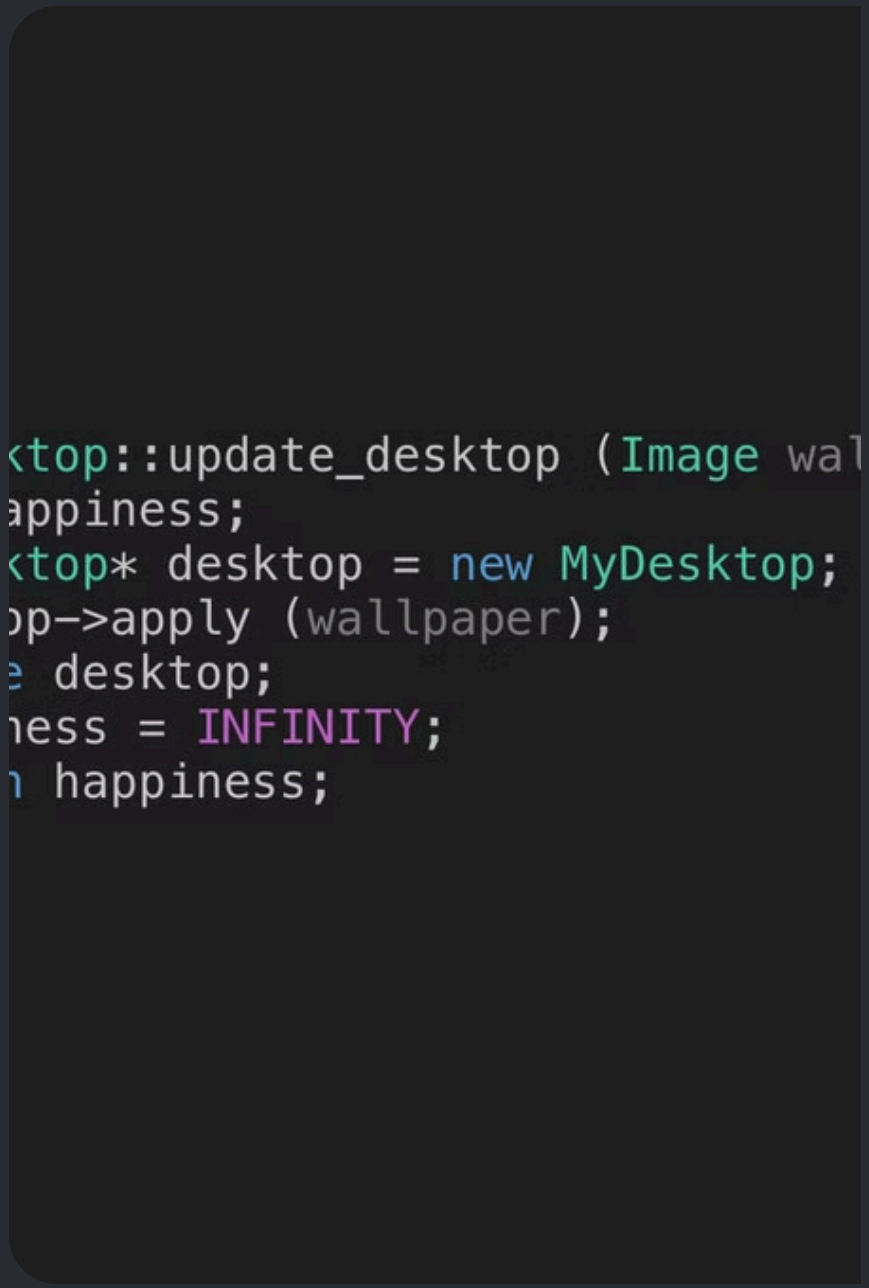


C++的核心：面向对象

本次讲解中，我们将深入探讨C++中的常考知识点，包括面向对象的三大特征、虚函数、struct与class的区别以及堆与栈的概念。

C++为什么叫C++

"++"在编程界被用作增量操作符，意味着比原来更进一步，这象征着C++相较于C的进化和功能增强。

起名C++的由来，不仅仅体现了C++语言是C语言的超集，更加重了它在功能上的增强。比尔·斯特劳斯特鲍普通过添加面向对象编程的特性，以及多种高级特性如模板和异常处理机制，让C++成为了一个高效而强大的编程语言。

C和C++之间的主要区别

面向对象编程

C++通过类和对象引入了面向对象编程，而C则没有这一概念，是更为基础的过程式语言。

标准库

C++拥有更加丰富的标准库，涵盖了数据结构、算法、输入输出等多方面的支持。

内存管理

C++的内存管理相对自动，使用了构造函数和析构函数，而C须由开发者手动管理内存。

具体区别-面试快问快答

C++引入 new/delete 运算符，取代了C中的 malloc/free 库函数；

C++引入引用的概念，而C中没有；

C++的标准库更加丰富

C++引入类的概念，而C中没有；

C++引入函数重载的特性，而C中没有

面向对象的三大特征

封装

封装（Encapsulation）是信息隐藏的艺术

继承

继承（Inheritance）是代码复用的机制

多态

多态（Polymorphism）允许同一操作应用于不同的对象

面向对象的基本特性

封装

信息隐藏的艺术

继承

代码复用的机制

多态

允许同一操作应用于不同的对象

抽象

通过抽象类和接口定义对象的行为，忽略具体的实现细节。

一、封装

定义

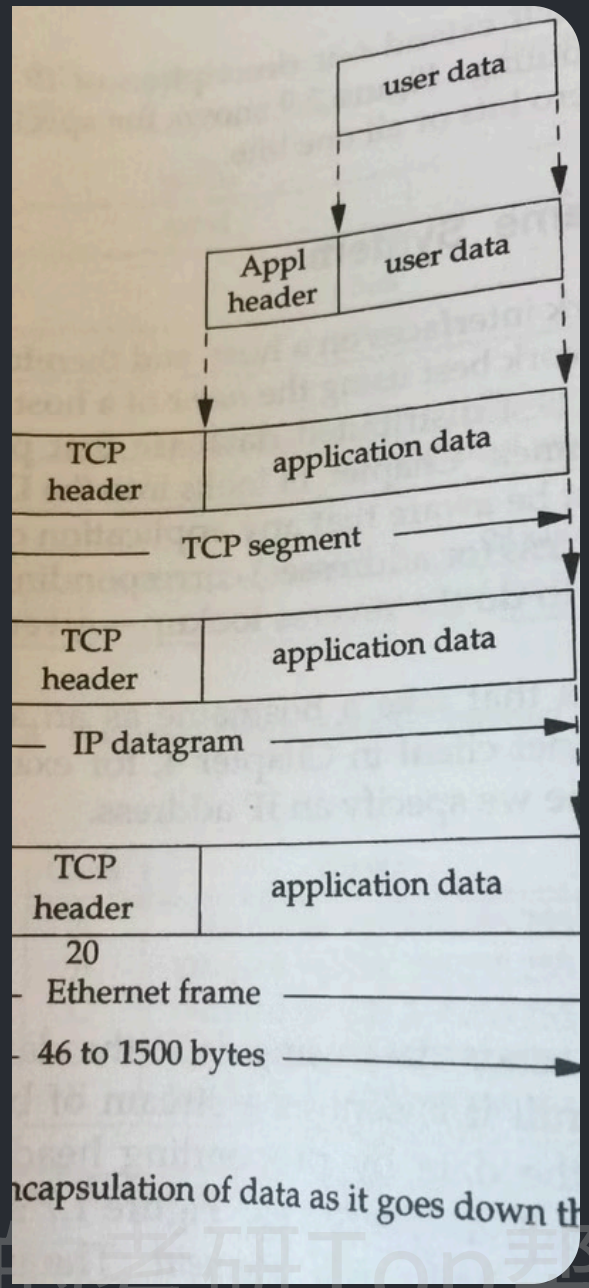
在C++中，封装是面向对象程序设计的核心原则之一，允许将数据（属性）和行为（方法）捆绑在单个工作单元内。

特点

通过关键字如private、protected、public来控制类的外部对其成员变量和成员函数的访问权限，增强数据安全性。

好处

封装简化了代码的管理，隐藏了复杂性，提高了类与类之间的接口质量和可维护性。



代码示例

```
class Character {
protected:
    int health; // 只能由Character本身及其派生类访问
private:
    std::string secretIdentity; // 只能由Character自己访问
public:
    void setHealth(int h); // 为health设置值的公共接口
    virtual void attack(Character& target) = 0; // 抽象攻击方法，所有角色都可以执行攻击动作
};
```

访问修饰符

public

公共访问权限

protected

受保护访问权限

private

私有访问权限

public

`public` 修饰符表示类中的成员可以被**类的外部**访问。这意味着你可以在类外直接使用对象访问 `public` 成员变量和成员函数。

```
class Character {
public:
    void setHealth(int h); // 设置 health 的公共接口
    virtual void attack(Character& target) = 0; // 抽象方法，需要子类实现
};
```

在 `Character` 类中，`setHealth()` 和 `attack()` 是 `public`，所以可以在类的外部直接调用：

```
Traveler traveler;
traveler.setHealth(100); // 可以直接调用 setHealth
traveler.attack(enemy);  // 可以直接调用 attack（由 Traveler 实现）
```

private

`private` 修饰符表示类中的成员只能在**类的内部**访问，无法在类的外部或者派生类中直接访问。它用来隐藏类内部的实现细节，不允许其他类直接访问。

```
class Character {
private:
    std::string secretIdentity; // 私有成员变量，只能由 Character 自己访问
};
```

在 `Character` 类中，`secretIdentity` 是 `private`，这意味着它只能在 `Character` 类的内部使用，其他类和对象无法直接访问它。

```
Traveler traveler;
// traveler.secretIdentity; // 错误！secretIdentity 是私有的，无法直接访问
```

private

即使是 `Character` 的派生类（如 `Traveler`），也不能直接访问 `secretIdentity`：

```
class Traveler : public Character {
public:
    void revealSecret() {
        // secretIdentity; // 错误！无法访问基类的 private 成员
    }
};
```

protected

protected 修饰符表示类中的成员可以被**类的内部**和**派生类**访问，但不能在类的外部访问。它用来给子类提供访问基类数据的权限，但又不希望让外部访问。

```
class Character {
protected:
    int health; // 受保护的成员变量，只能由 Character 和它的子类访问
};
```

protected

在 `Character` 类中，`health` 是 `protected`，这意味着它不能在类的外部直接访问，但可以在 `Character` 类内部以及 `Character` 的派生类（如 `Traveler`）中访问：

```
class Traveler : public Character {
public:
    void takeDamage(int damage) {
        health -= damage; // 可以访问基类的 protected 成员 health
    }
};
```

但不能在类的外部直接访问 `protected` 成员：

```
Traveler traveler;
// traveler.health = 50; // 错误！无法直接访问 protected 成员
```

二、继承

1

基础概念

继承允许创建基于通用类的特殊类，从而建立类与类之间的层次关系。

2

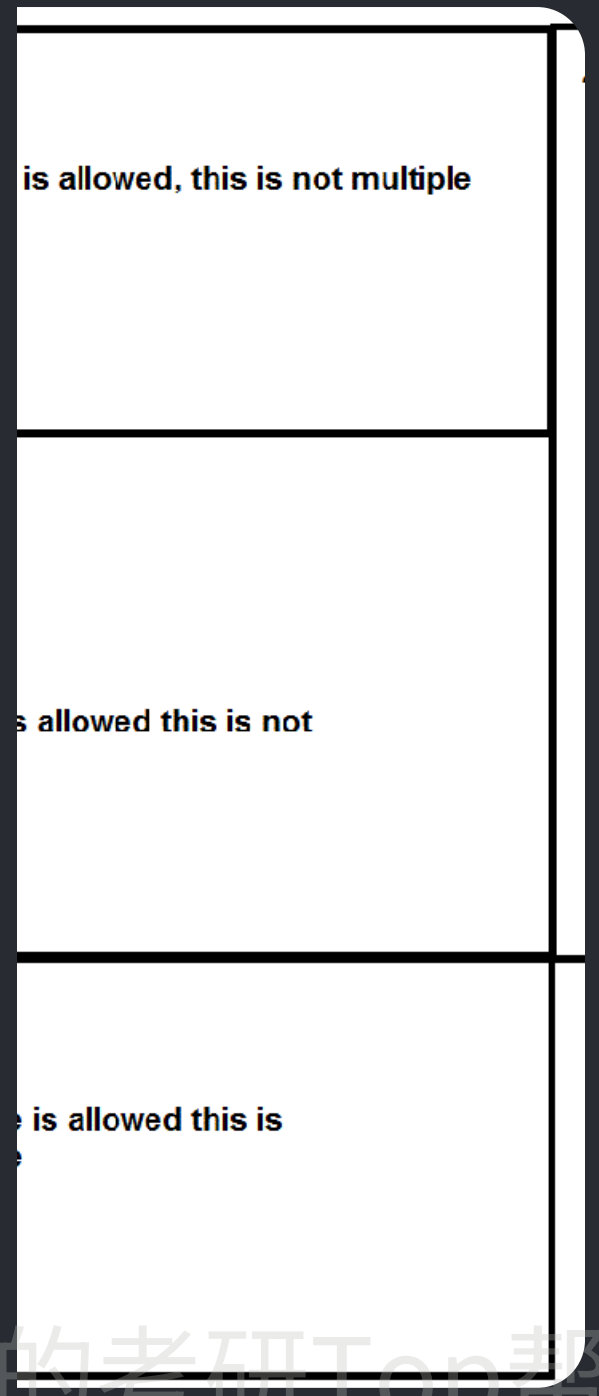
功能扩展

派生类继承基类特性的同时，还能够引入新的属性和方法，以实现功能扩展。

3

重写与重载

派生类能够重写（Override）基类的方法，或者重载（Overload）方法以实现特定功能。



子类示例

```
// 定义子类 Traveler 继承自 Character
class Traveler : public Character {
private:
    std::string element; // 新增一个私有成员变量，表示旅行者的元素属性

public:
    // 子类构造函数需要调用基类构造函数初始化共同的属性
    Traveler(int health, const std::string& identity, const std::string& travelerElement)
        : Character(health, identity), element(travelerElement) {}

    // 实现父类的抽象方法attack
```

代码示例

```
    void attack(Character& target) override {
        // 这里是旅行者特有的攻击逻辑，例如造成风元素伤害
        // ...
        std::cout << "Traveler is attacking with the power of " << element << "!\n";
    }

    // 子类新增的方法，比如展示旅行者的元素属性
    std::string getElement() const {
        return element;
    }

    // 重写getName方法，返回旅行者的名字
    std::string getName() const override {
        return "Traveler";
    }
};
```

三、多态

1 **概述**

多态性让C++程序有更高的通用性和可扩展性，它使得相同的消息可以根据发送对象的不同引发不同的行为。

2 **静态多态**

通过函数重载和模板类实现，编译器在编译期间确定调用的函数版本。

3 **动态多态**

通常通过虚函数实现，允许在运行时决定调用哪个函数版本，这也称为晚绑定（Late Binding）。

虚函数

1

基本定义

虚函数基于一个核心思想：在基类中定义一个函数接口，并允许在派生类中重写该接口的具体实现。

2

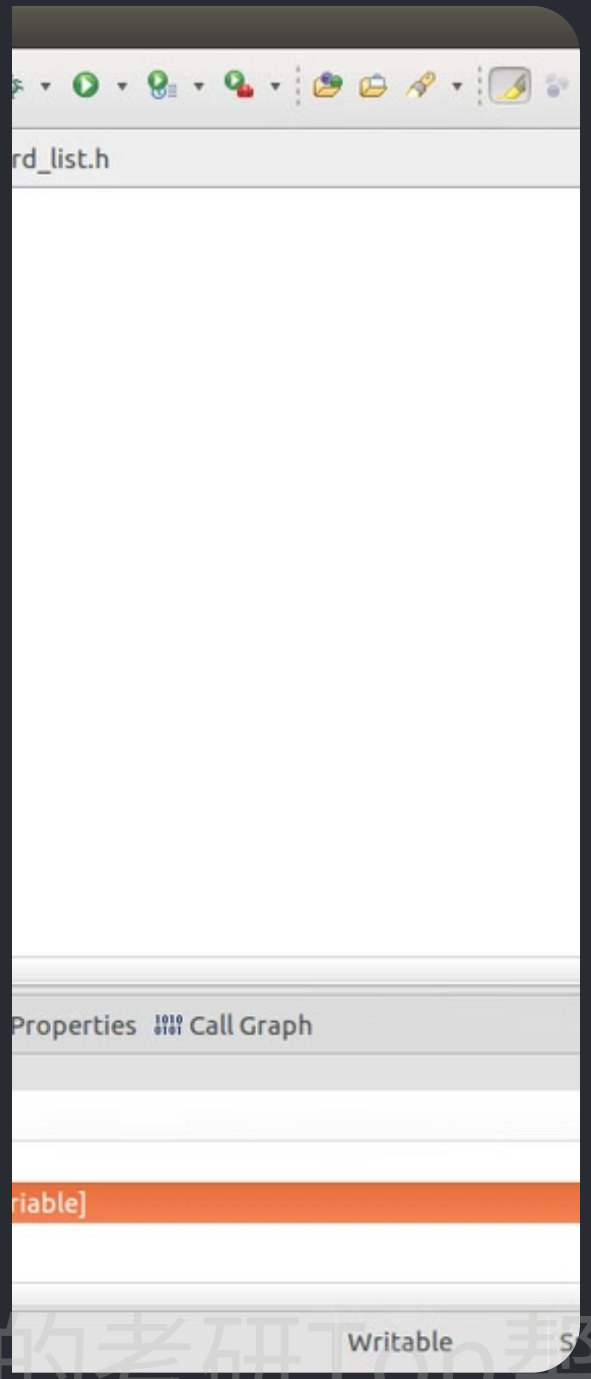
动态绑定

虚函数使得程序在运行时能够动态地决定应该调用对象的哪个函数版本。

3

应用场景

在设计框架和组件库时虚函数是多态性实现的重要手段，它增强了代码间的解耦和灵活性。



代码示例-基类

```
class Character {
protected:
    int health; // 只能由 Character 及其派生类访问
private:
    std::string secretIdentity; // 只能由 Character 自己访问

public:
    // 普通成员函数
    void setHealth(int h) {
        health = h;
        std::cout << "Character's health is set to " << health << std::endl;
    }

    // 虚函数：派生类可以重写此方法
    virtual std::string getName() const {
        return "Character";
    }
};
```

代码示例-子类

```
// 派生类 Traveler 继承自 Character
class Traveler : public Character {
private:
    std::string element; // 私有成员变量，表示旅行者的元素属性

public:
    // 重写普通成员函数
    void setHealth(int h) {
        health = h;
        std::cout << "Character's health is set to " << health << std::endl;
    }

    // 重写 getName 方法（虚函数）
    std::string getName() const override {
        return "Traveler";
    }
};
```

代码示例-对比

```
int main() {
    // 创建 Traveler 对象，并使用基类指针指向它
    Character* character = new Traveler(100, "Secret Traveler", "Wind");

    // 使用普通成员函数（不会多态调用）
    character->setHealth(80);  // 调用的是基类的 setHealth，不是 Traveler 的

    // 使用虚函数（支持多态调用）
    std::cout << "Character name: " << character->getName() << std::endl;  // 调用的是 Traveler 的
getName

    // 使用纯虚函数（必须在派生类中实现）
    character->attack(*character);  // 调用的是 Traveler 的 attack

    delete character;
    return 0;
}
```

纯虚函数的定义

1

纯虚函数定义

纯虚函数是类中声明的一个没有实现的虚函数，它通常用于定义接口规范。

2

抽象类

拥有纯虚函数的类被称为抽象类，它不能被实例化，只能作为基类用于派生。

3

派生类实现

所有非抽象派生类必须实现基类中的纯虚函数，以确保对象的完整性。

纯虚函数

```
class Character {
protected:
    int health; // 只能由Character本身及其派生类访问
private:
    std::string secretIdentity; // 只能由Character自己访问
public:
    void setHealth(int h); // 为health设置值的公共接口
    virtual void attack(Character& target) = 0; // 抽象攻击方法，所有角色都需要执行攻击动作
};
```

代码示例

```
    void attack(Character& target) override {
        // 这里是旅行者特有的攻击逻辑，例如造成风元素伤害
        // ...
        std::cout << "Traveler is attacking with the power of " << element << "!\n";
    }

    // 子类新增的方法，比如展示旅行者的元素属性
    std::string getElement() const {
        return element;
    }

    // 重写getName方法，返回旅行者的名字
    std::string getName() const override {
        return "Traveler";
    }
};
```

抽象

- 抽象是将一类事物的共同特征提取出来，形成抽象类或接口。抽象类和接口定义了一组抽象方法，具体的实现由子类完成。
- 例如，定义一个交通工具接口，包含行驶方法。汽车类、火车类等可以实现这个接口，各自实现行驶方法以体现不同交通工具的行驶方式。

虚函数原理是什么？

```
int main() {
    Character* c = new Traveler();
    c->attack(); // 输出：Traveler attacks with sword!
    delete c;
}
```

编译器如何确定 c->attack() 应该调用 Character 的 attack() 还是 Traveler 的 attack()?

虚函数原理

虚函数表

虚函数表，或称为VTable，是存储类中所有虚函数地址指针的结构。

运行时类型识别

RTTI运行时类型信息是C++中用于识别对象类型的机制。

动态绑定过程

运行时，程序通过vptr指针访问对象的虚函数表，以确定调用的正确方法。

总结

1. **普通成员函数**：无 virtual，编译时确定调用哪个类的函数，不支持多态。适用于不需要派生类多态行为的函数。
2. **虚函数**：使用 virtual，在子类中可以被重写，支持多态。运行时根据实际对象的类型调用子类重写的版本。
3. **纯虚函数**：是虚函数的一种特殊形式，必须在派生类中实现。基类不能提供具体实现，通常用于定义接口或抽象行为。

区别总结

特性	普通成员函数	虚函数	纯虚函数
关键字	无 `virtual` 关键字	使用 `virtual` 关键字	使用 `virtual` 和 `= 0`
多态性支持	不支持（静态绑定）	支持（动态绑定，运行时决定调用哪个函数）	支持多态性，必须在派生类中重写
实现要求	子类可以选择重写	子类可以选择重写	子类**必须重写**
调用效率	编译时确定（更快）	运行时通过虚函数表查找（稍慢）	同虚函数，需在派生类中实现并运行时调用
用途	没有多态需求时使用	需要多态性（不同子类有不同实现）时使用	为基类提供接口，强制派生类实现其功能

struct

struct是一种用户定义的数据类型，用于将不同类型的数据组合在一起

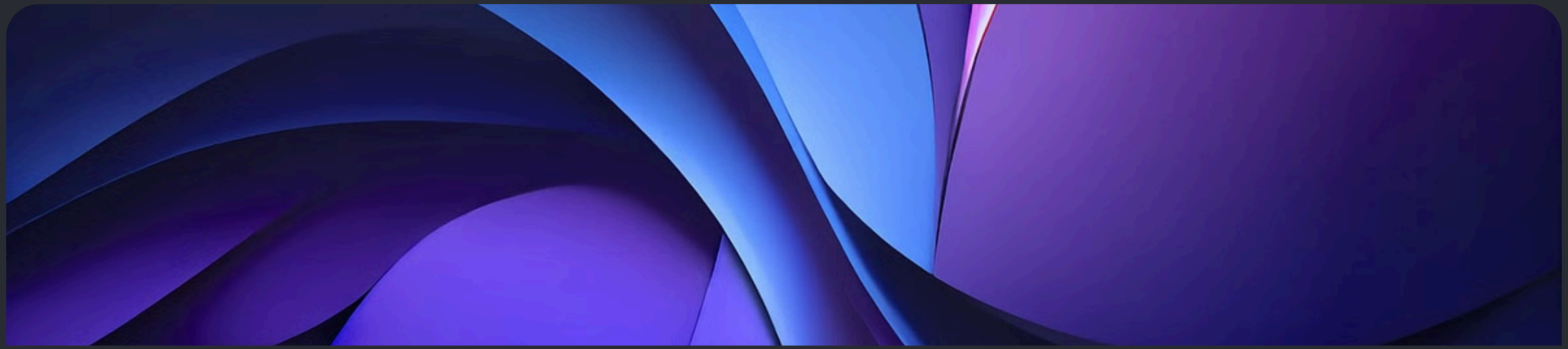
```
// 定义角色属性结构体
struct CharacterAttributes {
    int health;   // 生命值
    int attack;   // 攻击力
    int defense;  // 防御力

    // 成员函数：显示属性
    void display() const {
        std::cout << "生命值: " << health
                  << ", 攻击力: " << attack
                  << ", 防御力: " << defense << std::endl;
    }
};
```

struct和class的区别

特性	struct	class
默认访问权限	public	private
使用习惯	数据结构	面向对象类
成员函数与特性	支持	支持

关于面向对象的相关问题



常见C++面试问题

顺带讲一些课上没讲的琐碎知识点

堆和栈的区别

栈（Stack）

栈是一种**后进先出**（LIFO）的内存区域，在函数调用时自动分配和释放。通常用来存储**局部变量**和**函数调用信息**（包括**函数参数**、**返回地址**、**返回值**，以及**函数的局部变量**）。

堆（Heap）

堆是一块大的自由内存区域，内存分配是通过显式调用来完成的，例如使用 new 或 malloc。堆内存由**程序员手动管理**，需要手动分配和释放。

代码示例

```
int main() {
    // 栈上存储
    int stackVar = 5; // 局部变量，自动分配和释放

    // 堆上存储
    int* heapVar = new int(10); // 动态创建整型变量，需手动释放
    delete heapVar;

    // 或者使用 malloc/free
    int* cStyleHeapVar = (int*)malloc(sizeof(int)); // C风格动态分配内存
    *cStyleHeapVar = 20;
    free(cStyleHeapVar);

    // 在栈上动态分配，但仅限当前作用域有效
    void* stackAllocated = alloca(sizeof(int)); // 使用alloca在栈上动态分配，离开作用域自动释放
    *(int*)stackAllocated = 30;

    return 0;
}
```

什么是堆栈溢出（Stack Overflow）？如何避免?

回答：

- **堆栈溢出（Stack Overflow）**：当程序使用的栈内存超过其分配的大小时，发生堆栈溢出，通常导致程序崩溃。常见原因包括过深的递归调用和过大的局部变量。
- **如何避免**：
  - **限制递归深度**：确保递归函数有明确的终止条件，避免无限递归。
  - **使用迭代代替递归**：在可能的情况下，使用循环结构代替递归调用。
  - **避免在栈上分配过大的变量**：对于大型数据结构，使用堆内存分配。

```
// 避免
void func() {
    int largeArray[1000000]; // 可能导致栈溢出
}
// 改为
void func() {
    int* largeArray = new int[1000000];
    // 使用后记得释放内存
    delete[] largeArray;
}
```

什么是调用栈？函数调用时调用栈是如何变化的？

回答： 调用栈（Call Stack）是程序运行时用于管理函数调用和返回的内存结构。当一个函数被调用时，系统会在调用栈中为该函数分配一个栈帧，存储函数的参数、局部变量和返回地址。

变化过程：
1.函数调用：将返回地址、参数和局部变量压入栈中，创建新的栈帧。
2.函数执行：在栈帧中执行函数体。
3.函数返回：弹出当前栈帧，返回控制权到调用点。

什么是调用栈？函数调用时调用栈是如何变化的？

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

int main() {
    int sum = add(5, 10); // 调用add函数，调用栈变化
    return 0;
}
```

在调用`add(5, 10)`时，调用栈会：

1. 压入`main`函数的返回地址。
2. 压入`add`函数的参数`a=5`和`b=10`。
3. 创建`add`函数的栈帧，执行函数体。
4. 返回结果，弹出`add`的栈帧，恢复`main`的执行。

什么是作用域？C++中有哪些作用域？

回答： 作用域（Scope）指变量或函数在程序中可见和可访问的范围。C++中主要有以下几种作用域：

- **全局作用域**：在所有函数外部声明的变量，整个文件内可见。

```
int globalVar = 10; // 全局变量

int main() {
    std::cout << globalVar << std::endl;
}
```

- **局部作用域**：在函数或代码块内部声明的变量，仅在声明的函数或代码块内可见。

```
int main() {
    int localVar = 5; // 局部变量
    std::cout << localVar << std::endl;
}
```

什么是作用域？C++中有哪些作用域？

- **块作用域**：在花括号{}内声明的变量，仅在该块内可见。

```
if (true) {
    int blockVar = 20; // 块作用域变量
    std::cout << blockVar << std::endl;
}
// blockVar在这里不可访问
```

- **命名空间作用域**：在命名空间内部声明的变量或函数，仅在该命名空间内可见，除非使用using声明。

```
namespace MyNamespace {
    int nsVar = 30;
}

int main() {
    std::cout << MyNamespace::nsVar << std::endl;
}
```

C++中如何使用const修饰函数参数？有什么作用？

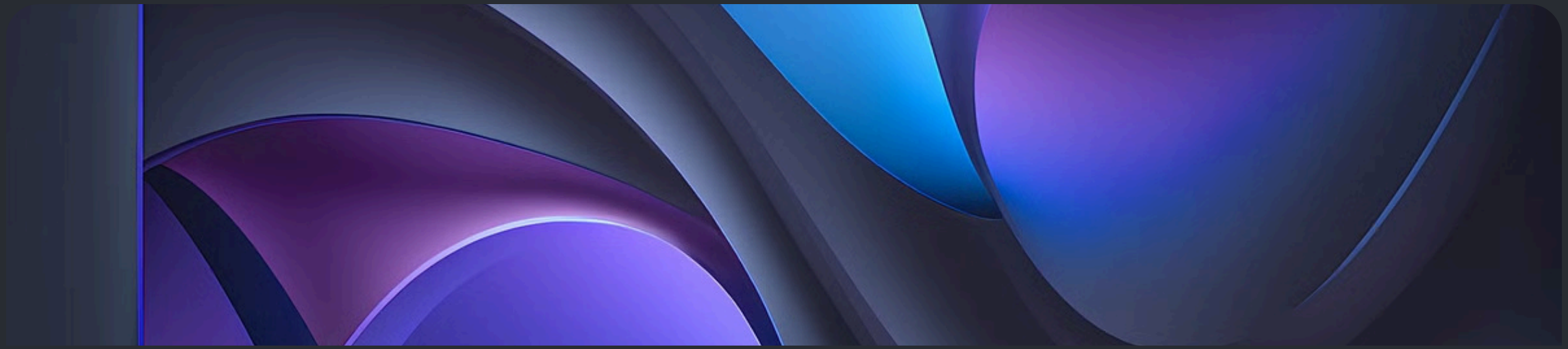
回答： 在函数参数前使用const修饰，表示函数内部不能修改该参数的值，增强代码的安全性和可读性。

- **示例**：

```
void displayMessage(const std::string &message) {
    std::cout << message << std::endl;
    // message = "New Message"; // 编译错误，无法修改
}

int main() {
    std::string msg = "Hello, C++!";
    displayMessage(msg);
}
```

- **作用**：
    - 防止函数内部意外修改参数的值。
    - 允许传递常量对象或临时对象，提高函数的通用性。



谢谢大家

更多问题可以参看课程文档～