第1课 C++常考概念

C++为什么叫C++

C++最初是由比尔·斯特劳斯特鲍普（Bjarne Stroustrup）在20世纪80年代早期开发的。他扩展了C语言，添加了面向对象编程（Object-Oriented Programming）的支持，这是C++最显著的特性之一

此外，C++还引入了许多其他特性，如模板、异常处理、标准模板库（STL）等，使得C++成为一种功能强大且多用途的编程语言。"++"的符号强调了C++相对于C语言的增强和扩展

C和C++之间的主要区别：

简略版本（老师问的时候快问快答）

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

详细区别（老师详细问的回答）

1. 面向对象编程（OOP）支持：
    - C：C是一种过程式编程语言，不直接支持面向对象编程。它使用函数来组织代码，并且数据和函数通常分离。
    - **C++：C++是一种支持面向对象编程的语言。它引入了类和对象的概念，允许开发人员使用封装、继承和多态等OOP特性来组织代码。**
2. 标准库：
    - C：C语言的标准库相对较小，主要包括基本的输入输出函数、字符串处理函数等。
    - C++：C++的标准库更加丰富，包括了C语言的标准库，并添加了大量用于容器、算法、输入输出流、字符串处理、文件操作等方面的标准库。
3. 语法：
    - C：C语言的语法相对较简单，更接近底层计算机操作。
    - C++：C++语言在C的基础上引入了更多的语法元素，包括类、对象、构造函数、析构函数、运算符重载等。
4. 内存管理：

- C：C语言提供手动内存管理，开发人员需要自行分配和释放内存。
- C++：C++引入了构造函数和析构函数，使得对象的内存管理更加自动化。通过使用类和对象，可以更容易地管理资源。

5. 编程风格：
   - C：C语言通常更适合编写系统级或嵌入式编程，注重性能和控制。
   - C++：C++更适合开发大型和复杂的应用程序，特别是需要使用OOP概念的项目。
6. 兼容性：
   - C：C语言代码通常可以直接在C++中使用，因为C++是C的超集。
   - C++：C++引入了一些新的关键字和语法，因此不是所有的C代码都能够直接在C++中编译通过，但大多数C代码可以很容易地转换为C++。

C++重要概念整理

下述是一些C++面试中容易问到的概念，但我们项目中并未涉及到，大家可以有空的时候阅读理解下述概念。

`private`、`protected`和`public`

在C++面向对象编程中，`private`、`protected`和`public`是类的访问权限修饰符，它们用来控制类成员（包括数据成员和成员函数）对外部世界以及其他类或派生类的可见性。这些区别体现在以下几个方面：

1. **private（私有）**
   - `private`成员只能被本类的成员函数访问。
   - 它们不能被外部代码直接访问，即使是在同一个文件中的其他函数或者不同的类中也不能直接访问。
   - 对于一个类来说，它的private部分体现了封装原则，即隐藏内部实现细节，只提供公共接口供外界使用。

2. **protected（保护）**

- `protected`成员可以被该类自身以及任何从该类派生出的子类的成员函数访问。
- 类似于private，protected成员也不能被类之外的普通代码直接访问，但比private更开放，因为它允许继承链上的子类访问这些成员。

- protected成员常用于设计需要在继承体系中共享但不希望对所有用户公开的数据和方法。

3. **public（公共）**

- `public` 成员可以被任何代码自由地访问，无论是同一类内的函数，还是类外的函数，或者是从该类派生出来的子类的函数。
- 公共成员定义了类的接口，是类与外界交互的主要方式。

以游戏为例，假设我们有一个基础角色类 `Character`：

```
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
};
```

在这种情况下，`health` 属性对于所有继承自 `Character` 的类如 `PlayerCharacter` 和 `Enemy` 都是可访问的，但是 `secretIdentity` 属性只有 `Character` 类本身可以访问。同时，所有的角色都有一个公共的方法 `setHealth()` 来调整血量，确保了对外提供的API一致性，而具体的实现细节则被隐藏在类的内部。

Static 关键字

`static` 是C++中一个非常重要的面试常考的关键字，它在不同上下文中有多种用途。重在理解

1. 静态局部变量（Static Local Variables）

静态局部变量是在函数内部声明并带有 `static` 修饰的变量。它的特点是：

- 生命周期：从程序开始运行至程序结束，即使函数已经返回，该变量依然存在。
- 初始化：只初始化一次，后续函数调用时保留上次的值。

示例代码：

```cpp
void func() {
    static int count = 0; // 静态局部变量
    std::cout << "This function has been called " << ++count << " times.\n";
}

int main() {
    for (int i = 0; i < 5; ++i) {
        func();
    }
    return 0;
}
```

在此例子中，尽管 `func()` 被调用了五次，但 `count` 只会被初始化为0一次，之后每次调用都会递增。

（所以叫他静态就是因为和动态相对，不会因为函数调用完了这个变量就被销毁了）

2. 静态全局变量（Static Global Variables）

在文件作用域内声明的 `static` 全局变量具有以下特点：

- 可见性：仅在当前编译单元（通常是一个源文件）内可见，不会与其它文件中的同名静态全局变量冲突。
- 生命周期：在整个程序运行期间持续存在。

重要异同辨析

1. 全局变量：
    - 作用域: 全局变量在整个程序（即所有源文件）中都是可见的，只要包含声明该变量的头文件，任何函数都可以直接访问它。
    - 存储区域: 存储在内存的全局数据区或静态存储区。
    - 生命周期: 从程序开始运行时创建，直到程序结束时销毁。
    - 初始化: 如果没有明确初始化，全局变量会被自动初始化为0（对于基本类型）或其默认构造函数（对于类对象）。
    - 特性: 全局变量在程序的所有部分共享同一份数据，因此不同函数修改全局变量会影响到其他函数的行为。
2. 静态全局变量：

- 作用域: 静态全局变量的作用域仅限于定义它的编译单元（通常是单个源文件），在其他源文件中不可见，除非通过extern关键字进行声明。
- 存储区域: 同样存储在全局数据区或静态存储区。
- 生命周期: 和全局变量一样，从程序开始运行到程序结束，生命周期贯穿整个程序执行过程。
- 初始化: 如果没有手动初始化，静态全局变量也会被初始化为0。
- 特性: 虽然作用域相对受限，但静态全局变量同样在整个程序执行过程中只有一份实例，并且不会因函数调用而重新分配或销毁。

3. 静态局部变量：
   - 作用域: 静态局部变量在其所在的函数内部有效，只有在定义它的那个函数内可以访问。
   - 存储区域: 仍然存储在全局数据区或静态存储区。
   - 生命周期: 它是在第一次进入该函数时创建，而不是每次函数调用时都创建新的变量。因此，即使函数执行完毕返回，静态局部变量的值仍然保留，下次调用该函数时继续使用上次的值。
   - 初始化: 静态局部变量只初始化一次，不论函数调用多少次。如果没有初始化，则同样会被初始化为0。
   - 特性: 静态局部变量提供了在函数内部保持持久化状态的能力，而不影响其他函数或模块的数据独立性。

3. 静态成员变量（Static Member Variables）

类中的静态成员变量属于类本身，不属于任何对象实例，且所有对象共享同一份拷贝。特点如下：

- 访问：可以通过类名直接访问，无需创建类的对象。
- 生命周期：从程序开始到结束。

示例代码：

```
class MyClass {
public:
    static int sharedValue; // 静态成员变量

    void printSharedValue() {
        std::cout << "The shared value is: " << MyClass::sharedValue << "\n";
    }
};

int MyClass::sharedValue = 0;
```

```
11
12 int main() {
13     MyClass obj1, obj2;
14     MyClass::sharedValue = 10;
15     obj1.printSharedValue(); // 输出: 10
16     obj2.printSharedValue(); // 输出: 10
17     return 0;
18 }
```

4. 静态成员函数（Static Member Functions）

静态成员函数属于类，不与类的任何对象关联，因此不能访问非静态成员。常用于操作静态成员变量或进行与类相关的不需要具体对象的操作。

异同辨析（有印象理解即可）

- **静态成员函数**：
  - **不与任何对象实例关联，通过类名直接调用。**
  - 可访问静态数据成员，不能直接访问非静态数据成员（需通过对象引用或指针）。
  - 用于实现与类相关而非特定对象的逻辑。

- **非静态成员函数**：
  - **与对象实例绑定，通过对象或指针调用。**
  - 可访问所有成员（静态和非静态），通常处理对象的具体业务逻辑和状态变化。
  - 调用时隐含传递 `this` 指针，指向当前对象。

示例代码：

```
1  #include <iostream>
2
3  class BankAccount {
4  private:
5      // 非静态数据成员
6      int accountNumber;
7      double balance;
8
9      // 静态数据成员
10     static int totalAccounts;
11
12 public:
13     // 非静态构造函数，与每个对象实例关联
```

```cpp
    BankAccount(int num, double initialBalance) : accountNumber(num), balance(initialBalance) {
        ++totalAccounts;  // 非静态成员函数中访问并修改静态数据成员
    }

    // 非静态成员函数，操作单个对象的状态
    void deposit(double amount) {
        this->balance += amount;
        std::cout << "Deposited " << amount << " into account #" << this->accountNumber << ".\n";
    }

    // 非静态成员函数，操作单个对象的状态
    double getBalance() {
        return this->balance;
    }

    // 析构函数，减少账户总数
    ~BankAccount() {
        --totalAccounts;
    }

    // 静态成员函数，与类相关，不依赖于任何特定对象
    static void displayTotalAccounts() {
        std::cout << "Total number of accounts created: " << totalAccounts << "\n";
        // 注意：这里不能直接使用 this 或任何非静态数据成员
    }

    // 初始化静态数据成员
    static int getTotalAccounts() {
        return totalAccounts;
    }

    // 静态数据成员初始化
    static int totalAccounts = 0;
};

int main() {
    BankAccount acc1(1, 1000.0);  // 创建第一个账户
    BankAccount acc2(2, 2000.0);  // 创建第二个账户

    // 使用非静态成员函数操作单个对象
    acc1.deposit(500);
    std::cout << "Balance of account #1: " << acc1.getBalance() << "\n";

    // 直接通过类名调用静态成员函数，它不会作用于任何具体对象
```

```
    BankAccount::displayTotalAccounts();  // 输出创建的所有账户数量
    std::cout << "Total Accounts via getter: " <<
BankAccount::getTotalAccounts() << "\n";

    return 0;
}
```

综上所述，`static`关键字在C++中的应用丰富多样，涵盖了生命周期管理、作用域控制以及类级别的数据和方法等方面

面向对象的三大特征

一、封装

- **定义**：封装是将客观事物抽象成类的过程，通过隐藏内部实现细节和暴露必要的接口来控制对数据和方法的访问权限。
- **特点**：
  - 类可以对外公开（暴露出）特定的数据成员和成员函数给可信的类或对象使用。
  - 同时，对于不可信的对象，可以通过访问限制（如private/protected）进行信息隐藏，确保数据的安全性和完整性。

```
class Animal {
private: // 隐藏内部实现
    std::string species; // 私有成员变量

public:
    // 公开访问接口（getter 和 setter）
    void setSpecies(const std::string& s) {
        species = s;
    }
    std::string getSpecies() const {
        return species;
    }

    // 公开方法
    virtual void makeSound() const {
        std::cout << "The animal makes a sound.\n";
    }
};

```

```
int main() {
    Animal cat;
    cat.setSpecies("Cat"); // 通过公开接口设置私有变量
    std::cout << cat.getSpecies(); // 输出 "Cat"
    cat.makeSound(); // 使用公开方法
    return 0;
}
```

二、继承

- **定义**：继承是一种复用现有类功能的方式，通过派生新类来扩展原有类的功能，而无需重新编写原始代码。
- **特点**：
  - 新创建的派生类自动拥有基类的所有属性和方法，并可以根据需要增加新的属性和重写原有的方法。

```
class Animal { // 基类
public:
    virtual ~Animal() {}
    virtual void makeSound() const {
        std::cout << "The animal makes a sound.\n";
    }
};

class Dog : public Animal { // 派生类
public:
    void makeSound() const override {
        std::cout << "The dog barks.\n";
    }
};

class Cat : public Animal { // 派生类
public:
    void makeSound() const override {
        std::cout << "The cat meows.\n";
    }
};

int main() {
    Animal* animalPtr = new Dog();
    animalPtr->makeSound(); // 输出 "The dog barks."

```

```
    animalPtr = new Cat();
    animalPtr->makeSound(); // 输出 "The cat meows."

    delete animalPtr; // 释放内存
    return 0;
}
```

三、多态

- **概述**：多态是指一个类的不同实例在不同情况下表现出不同的行为特性，使得具有不同内部结构的对象能够共享相同的外部接口，从而达到灵活和统一处理的目的。
- **分类**：
  - **静态多态（编译时多态）**：
    - 实现方式：通过函数重载（Overloading）和模板技术（Templates）实现。
    - 特点：编译器在编译期间就能确定调用哪个函数版本。

    函数重载（Function Overloading）是指在同一作用域内，可以定义多个同名的函数，但这些函数的参数列表不同（包括参数个数、类型或顺序的不同）。编译器会根据调用函数时传递的实际参数来决定调用哪个函数实现。

```
#include <iostream>
// 函数重载：计算两个整数之和
int sum(int a, int b) {
    return a + b;
}
// 函数重载：计算两个浮点数之和
double sum(double a, double b) {
    return a + b;
}
// 函数重载：计算三个整数之和
int sum(int a, int b, int c) {
    return a + b + c;
}
int main() {
    std::cout << "Sum of two integers: " << sum(3, 5) << std::endl; // 输出8
    std::cout << "Sum of two floats: " << sum(3.5, 2.7) << std::endl; // 输出6.2
    std::cout << "Sum of three integers: " << sum(1, 2, 3) << std::endl; // 输出6
    return 0;
}
```

- **动态多态（运行时多态）：**
  - 实现方式：通过虚函数（Virtual Functions）和继承关系实现。
  - 特点：编译器不能在编译阶段决定具体调用哪个函数，而是在运行期根据实际对象类型执行动态绑定，选择合适的函数版本。

```
// 动态多态的例子，继续使用上面继承部分的代码

int main() {
    Animal* animals[2] = {new Dog(), new Cat()};

    for (int i = 0; i < 2; ++i) {
        animals[i]->makeSound(); // 根据实际类型调用不同的函数版本，动态绑定
    }

    // 释放所有动物对象的内存
    for (Animal* animal : animals) {
        delete animal;
    }

    return 0;
}
```

虚函数

虚函数的定义

1. 基本定义：

在C++中，一个成员函数被声明为 `virtual` 时，我们就称它为“虚函数”。虚函数允许派生类重写基类中的函数行为，并且通过基类指针或引用调用时能够调用派生类的版本（动态绑定）。

```
class Base {
public:
    virtual void display() {
        std::cout << "This is Base's display.\n";
    }
};

class Derived : public Base {
public:
```

```
    void display() override { // 使用override表明该函数覆盖了基类的虚函数
        std::cout << "This is Derived's display.\n";
    }
};
```

2. 纯虚函数：

纯虚函数是在基类中声明但没有提供实现的虚函数，必须在所有非抽象派生类中实现。纯虚函数使用 `= 0` 来标识：

```
class AbstractBase {
public:
    virtual void doSomething() = 0; // 纯虚函数，不允许实例化AbstractBase对象
};
```

虚函数原理

原理核心：

虚函数的工作基于运行时类型信息（RTTI）和虚函数表（VTable）。每个具有虚函数的类都会有一个隐藏的虚函数表，其中包含了指向虚函数地址的指针数组。当创建一个对象时，编译器会在对象内部添加一个指向其对应类的虚函数表的指针（vptr）。当通过基类指针调用虚函数时，实际上是通过这个vptr找到正确的虚函数表，然后执行对应的函数。

函数映射与代码示例

下面是一个完整的例子，展示如何通过基类指针调用实际派生类的虚函数：

```
#include <iostream>

// 定义基类
class Animal {
public:
```

```cpp
    virtual ~Animal() {} // 通常需要给基类提供虚析构函数以保证正确销毁派生类对象

    virtual void makeSound() const {
        std::cout << "The animal makes a sound.\n";
    }
};

// 定义派生类
class Dog : public Animal {
public:
    void makeSound() const override {
        std::cout << "The dog barks.\n";
    }
};

class Cat : public Animal {
public:
    void makeSound() const override {
        std::cout << "The cat meows.\n";
    }
};

int main() {
    Animal* animalPtr;
    Dog myDog;
    Cat myCat;

    animalPtr = &myDog;
    animalPtr->makeSound(); // 输出 "The dog barks."

    animalPtr = &myCat;
    animalPtr->makeSound(); // 输出 "The cat meows."

    return 0;
}
```


在这个例子中：

- `Animal` 类包含一个虚函数 `makeSound()`。
- `Dog` 和 `Cat` 类从 `Animal` 派生并重写了 `makeSound()` 函数。
- 当我们通过 `Animal` 类型的指针调用 `makeSound()` 时，根据指针所指向的实际对象类型（运行时信息），会调用相应的 `makeSound()` 实现。

当然，以下是关于虚函数、纯虚函数以及虚函数表等相关问题的整理和解析，我也会提供相应的代码示例进行说明。

相关问题

虚函数表是针对类的还是针对对象的？

虚函数表是针对类的。同一个类的所有对象共享同一个虚函数表。每个对象内部都保存一个指向该类虚函数表的指针 `vptr`。虽然每个对象的 `vptr` 地址不同，但它们都指向同一个虚函数表。

为什么基类的构造函数不能定义为虚函数？

构造函数不可以是虚函数。原因是虚函数的调用依赖于虚函数表，而虚函数表的指针 `vptr` 需要在对象的构造函数中初始化。在构造函数执行之前，`vptr` 还未被初始化，因此无法使用虚函数机制。

为什么基类的析构函数需要定义为虚函数？

当使用基类指针指向派生类对象时，为了正确调用派生类的析构函数来释放资源，基类的析构函数需要定义为虚函数。如果析构函数不是虚函数，那么在删除派生类对象时，可能无法正确调用派生类的析构函数，导致资源泄漏。

```
class Base {
public:
    virtual ~Base() {} // 虚析构函数
};

class Derived : public Base {
    // ...
};
```

构造函数和析构函数能抛出异常吗？

- **构造函数**：从语法上讲，构造函数可以抛出异常。但从逻辑和风险控制的角度来说，应尽量避免在构造函数中抛出异常。如果构造函数抛出异常，可能会导致资源（如已分配的内存）无法正确释放，从而引起内存泄漏。

- **析构函数**：析构函数不应该抛出异常。如果析构函数中抛出异常，可能会导致程序中断或无法正确释放资源。特别是在异常处理过程中，如果另一个异常被抛出（而当前异常还未处理完毕），程序可能会直接终止。因此，析构函数应该捕获并处理其内部可能产生的任何异常，而不是抛出它们。

以下是一个展示如何在构造函数和析构函数中处理异常的示例代码：

```
class Example {
public:
    Example() {
        try {
            // 构造函数中的代码
        } catch (...) {
            // 处理构造过程中的异常
            throw; // 重新抛出异常
        }
    }

    ~Example() {
        try {
            // 析构函数中的代码
        } catch (...) {
            // 处理析构过程中的异常
            // 不要重新抛出异常
        }
    }
};
```

在这个例子中，构造函数中的异常被重新抛出以便调用者可以捕获，而析构函数中的异常被捕获并处理，但不重新抛出，以避免可能的程序中断。

`struct` 和 `class` 区别

在C++中，`struct` 和 `class` 关键字用于定义类，两者在功能上几乎完全相同，主要区别在于成员的默认访问权限和使用习惯：

1. 默认访问权限：
   - `struct`：成员默认为 `public`。
   - `class`：成员默认为 `private`。
2. 使用习惯：
   - `struct` 通常用于定义数据结构，强调数据的聚合。
   - `class` 通常用于实现更复杂的对象，包含封装、继承、多态等特性。
3. 功能等同：
   - `struct` 和 `class` 在C++中功能上是等同的，都可以拥有构造函数、析构函数、成员函数、继承、多态等特性。

代码示例

```cpp
// 使用struct定义一个简单的数据结构
struct Point {
    int x, y; // 默认为public

    Point(int x, int y) : x(x), y(y) {} // 构造函数

    void move(int dx, int dy) {
        x += dx;
        y += dy;
    }
};

// 使用class定义一个类
class Rectangle {
private: // 默认为private
    Point topLeft;
    int width, height;

public:
    Rectangle(const Point& topLeft, int width, int height)
        : topLeft(topLeft), width(width), height(height) {}

    int area() const {
        return width * height;
    }
};
```

```
27
```

在这个示例中，`Point` 使用 `struct` 定义，其成员默认为 `public`，而 `Rectangle` 使用 `class` 定义，其成员默认为 `private`。但是两者都可以包含成员函数和构造函数，都可以用于实现面向对象的特性。

面试回答：使用习惯上，struct用于定义数据结构，class用于实现封装与多态性，但实际功能上只有默认权限不同，`struct` 定义成员默认为 `public`，`class` 定义，成员默认为 `private`，技术上两者完全可以互换

（这个复试要是抽到这么答可以装到，大多同学不知道struct和class底层几乎没有区别，struct也可以实现成员继承等等多态性）

堆和栈的区别

在C++编程中，数据结构存储在堆和栈上的区别主要体现在内存管理、分配方式以及对对象生命周期的影响上：

1. **栈（Stack）**：
   - 栈上存储的数据主要包括：局部变量（包括基本类型和内置数组等）、函数参数、返回地址以及编译器自动分配的临时变量。（简单说基本就是new之外的局部变量都在这里）
   - 存储特点：
     - 自动分配和释放：栈空间由编译器自动管理，当作用域结束时，栈上的变量会自动销毁，无需程序员手动释放。
     - 空间有限且固定：栈的大小一般在程序启动时由系统预先设定，并且有限制。如果栈上分配的空间过大，可能导致栈溢出。
     - 无碎片区分：栈上存储的数据连续，不会产生内存碎片。

```
1 void someFunction() {
2     int stackVariable = 10; // 这个变量存储在栈上
3 }
```

2. 堆（Heap）：

- 堆上存储的数据主要包括：通过 `new` 操作符动态分配的对象、数组或其他数据结构。
- 存储特点：

- 动态分配和释放：使用 `new` 申请内存后，需要通过 `delete` 来释放，否则会导致内存泄漏；同样， `malloc` 与 `free` 配合使用也是相同道理。
- 大小灵活可变：堆内存空间可以根据需要动态调整，没有固定的上限，但受限于系统资源。
- 可能产生内存碎片：由于频繁地分配和释放不同大小的内存块，可能会导致堆内存中存在无法利用的小块内存，即内存碎片。

```
int* heapVariable = new int(20); // 这个对象存储在堆上
delete heapVariable;
```

更多例子：

```
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;
}
```

静态存储区：

- 全局变量、静态变量和常量存储在静态存储区。
- 全局变量在整个程序运行期间始终存在，静态局部变量在第一次初始化后其值会在函数调用间保持不变。

例如：

```cpp
int globalVariable = 30; // 全局变量存储在静态存储区
void anotherFunction() {
    static int staticLocalVariable = 40; // 静态局部变量也存储在静态存储区
}
```

在这个例子中，`stackVar`是在栈上分配的，而`heapVar`是通过`new`在堆上分配的。`alloca`分配的内存也在栈上，但它不是标准C++的一部分，在一些编译器中可用，其生命周期仅限于当前的作用域内。

迭代器和指针的区别（问的频率不高，了解即可）

1. 抽象层次
   - 指针：低级内存访问工具，可以直接操作内存地址。
   - 迭代器：高级抽象接口，专为STL容器设计，模仿指针行为，提供安全、一致的方式来访问容器内元素。
2. 功能与特性
   - 指针：支持算术运算、比较和解引用，无特定容器相关限制或保护。
   - 迭代器：同样支持类似操作，并根据容器类型分为单向、双向和随机访问等不同迭代器。符合STL算法要求，适用于泛型编程。
3. 安全性
   - 指针：不当使用可能导致越界访问或悬挂指针，安全性较低。
   - 迭代器：通常受到容器边界检查保护，超出范围的行为未定义，但可能提供额外安全保障。
4. 封装性
   - 指针：不具备容器特性和功能。
   - 迭代器：可能封装特定于容器的功能，如反向迭代等。

总结：

- 指针是通用的内存访问工具，而迭代器是针对容器优化的访问机制。
- 迭代器增强了指针的安全性和一致性，使代码更易于维护和扩展，更适合现代C++编程中的容器遍历和算法应用

常见C++问题

下面的问题很多课上没有讲的比较琐碎的知识点，大家可以通过问题的方式了解

问题1：什么是作用域？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。

```cpp
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。

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

- **块作用域**：在花括号 `{}` 内声明的变量，仅在该块内可见。

```cpp
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。

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

```
4
5 int main() {
6     std::cout << MyNamespace::nsVar << std::endl;
7 }
```

问题2：什么是调用栈？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。

- **示例**：

```
1 int add(int a, int b) {
2     return a + b;
3 }
4
5 int main() {
6     int sum = add(5, 10); // 调用add函数，调用栈变化
7     return 0;
8 }
```

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。

问题3：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：**

```
1 void displayMessage(const std::string &message) {
2     std::cout << message << std::endl;
3     // message = "New Message"; // 编译错误，无法修改
4 }
5
6 int main() {
7     std::string msg = "Hello, C++!";
8     displayMessage(msg);
9 }
```

- **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。

问题4:堆栈的区别

回答：

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。

- **局部变量**：当你在函数内声明变量（例如基本类型 `int a` 或 `double x`），它们一般会存储在栈中。它们的生命周期由函数的作用域决定，函数执行完后，变量会被自动释放。

- **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，这些都会存储在栈上。

- **栈内存分配**：速度快、管理简单，因为是编译器自动分配和释放的。栈的大小通常比较小，有固定的大小限制，容易发生栈溢出（stack overflow）。

示例：

```
void foo() {
    int a = 10; // 栈上的局部变量
    double b = 20.5; // 栈上的局部变量
}
```

在上面的 `foo` 函数中，变量 `a` 和 `b` 都被分配在栈上。当 `foo` 结束时，`a` 和 `b` 会被自动释放。

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。

- **动态分配的对象**：当你使用 `new` 或 `malloc` 来分配内存时，内存被分配在堆上，需要手动使用 `delete` 或 `free` 来释放。例如，动态数组、动态对象，或者需要跨越函数作用域的变量。
- **更灵活但更慢**：与栈相比，堆内存更灵活，因为你可以在运行时动态分配任意大小的内存。然而，分配和释放内存的速度较慢，而且如果忘记释放内存会导致内存泄漏。

示例：

```
void foo() {
    int* a = new int(10); // 动态分配的变量，在堆上
    double* b = new double[100]; // 动态分配的数组，在堆上

    // 使用完之后需要手动释放
    delete a;
    delete[] b;
}
```

在 `foo` 函数中，`a` 和 `b` 都是指针，指向堆上的内存。需要手动调用 `delete` 或 `delete[]` 释放内存，否则会导致内存泄漏。

总结对比

特性	栈（Stack）	堆（Heap）
典型数据	局部变量、函数参数、返回地址	动态分配的对象、动态数组
分配方式	编译器自动分配和释放	手动分配（`new` / `malloc`）和手动释放（`delete` / `free`）
分配速度	快（静态分配）	慢（动态分配，内存碎片化可能降低效率）
生命周期	作用域结束时自动释放	需要显式释放，否则会导致内存泄漏
大小限制	通常有固定的大小限制，较小	大小取决于系统内存，可分配大量内存
性能	较高，适合短期使用的对象	较低，适合跨函数作用域的大对象或可变大小的对象

问题5：什么是面向对象编程（OOP）？C++如何支持OOP的基本特性？

回答：

面向对象编程（OOP）是一种编程范式，通过将数据和操作数据的函数封装在对象中，来模拟现实世界中的实体和行为。OOP的基本特性包括封装、继承、多态和抽象。

- **封装**：将数据和方法绑定在一起，隐藏内部实现细节，保护数据不被外部直接访问。
- **继承**：允许一个类（子类）继承另一个类（父类）的属性和方法，实现代码的重用和扩展。
- **多态**：允许不同类的对象以相同的接口调用不同的实现，提高代码的灵活性和可扩展性。
- **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。

C++通过类和对象、访问修饰符（public、protected、private）、虚函数和纯虚函数等机制来支持OOP的这些特性。

问题6：什么是类和对象？它们之间有什么区别？

回答：

- **类（Class）**：类是对现实世界中某一类事物的抽象描述，定义了对象的属性（成员变量）和行为（成员函数）。它是一个模板，用于创建具体的对象。

- **对象（Object）**：对象是类的实例，是具有具体属性和行为的实体。通过类可以创建多个对象，每个对象都有独立的属性值。

区别：

- 类是抽象的模板，定义了对象的结构和行为。
- 对象是类的具体实例，拥有类中定义的属性和行为。

问题7：什么是继承？C++中支持哪些类型的继承？

回答：

- **继承（Inheritance）**：继承是面向对象编程中的一个重要特性，允许一个类（子类）继承另一个类（父类）的属性和方法，实现代码的重用和扩展。

- **C++中支持的继承类型**：

  - **公有继承（Public Inheritance）**：父类的public和protected成员在子类中分别成为public和protected成员。公有继承表示“是一个”关系。

  - **保护继承（Protected Inheritance）**：父类的public和protected成员在子类中分别成为protected成员。较少使用。

  - **私有继承（Private Inheritance）**：父类的public和protected成员在子类中分别成为private成员。表示“基于实现”关系。

示例：

```cpp
class Base {
public:
    void display() { }
protected:
    int value;
private:
    int secret;
};

class Derived : public Base { // 公有继承
public:
    void show() {
        display(); // 可访问
        value = 10; // 可访问
        // secret = 5; // 不可访问
    }
};
```

问题7：什么是多态？C++如何实现多态？

回答：

- **多态（Polymorphism）**：多态是指同一个接口可以有不同的实现，允许不同类的对象通过同一接口调用不同的函数实现，提高代码的灵活性和可扩展性。

- **C++中实现多态的方式**：

  - **虚函数（Virtual Functions）**：通过在基类中声明虚函数，允许子类重写这些函数，实现运行时多态。

  - **纯虚函数和抽象类（Pure Virtual Functions and Abstract Classes）**：基类中声明纯虚函数，子类必须实现这些函数，形成抽象类。

问题8：什么是纯虚函数？C++中如何定义纯虚函数？

回答：

- **纯虚函数（Pure Virtual Functions）**：纯虚函数是在基类中声明但不提供实现的虚函数，强制要求子类必须实现该函数，使基类成为抽象类。

- **定义纯虚函数的语法：**

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

- **作用：**

- 定义接口：基类提供接口规范，子类必须实现具体的功能。
- 创建抽象类：无法实例化抽象类，只能用于继承。

问题9：什么是指针（Pointer）？C++中使用指针的注意事项有什么？

回答：

- **指针（Pointer）**：指针是存储内存地址的变量，用于间接访问和操作其他变量。指针可以指向任何类型的数据，包括基本类型、对象、函数等。

注意事项：

- **初始化**：指针在使用前应初始化，避免野指针。
- **内存管理**：手动管理动态分配的内存，防止内存泄漏。
- **类型匹配**：指针类型应与所指向的数据类型匹配。

问题10：C++中什么是堆栈溢出（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;
    }
```

问题11：什么是抽象类？C++中如何定义和使用抽象类？

回答：

- **抽象类（Abstract Class）**：抽象类是包含至少一个纯虚函数的类，不能实例化，只能作为基类使用。它用于定义接口和行为规范，强制子类实现特定的功能。

- **定义方法**：

- 在类中声明至少一个纯虚函数。

示例：

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

class Dog : public Animal {
public:
    void speak() override {
        std::cout << "Dog barks" << std::endl;
```

```
    }
};

int main() {
    // Animal a; // 错误，无法实例化抽象类
    Animal* pet = new Dog();
    pet->speak(); // 输出：Dog barks
    delete pet;
}
```

特点：

- **不可实例化**：只能通过派生类来实现具体功能。

- **定义接口**：基类定义了子类必须实现的接口和行为。

应用场景：

- 定义抽象接口，确保子类实现特定的功能。
- 实现多态，允许通过基类指针或引用调用子类的实现。

问题12：什么是静态成员函数？它与普通成员函数有什么区别？

回答：

- **静态成员函数（Static Member Functions）**：

- 属于类本身，而不是类的某个对象。
- 不能访问类的非静态成员变量和非静态成员函数。
- 可以通过类名直接调用，无需实例化对象。

- **区别**：

  - **访问权限**：静态成员函数只能访问静态成员变量和其他静态成员函数。

  - **调用方式**：静态成员函数可以通过类名调用，而普通成员函数需要通过对象调用。

  - **绑定**：静态成员函数在编译时绑定，不能被虚函数机制支持。

示例：

```
class MyClass {
public:
    static int staticVar;

    static void staticFunc() {
        std::cout << "Static Function, staticVar: " << staticVar << std::endl;
    }

    void normalFunc() {
        std::cout << "Normal Function" << std::endl;
    }
};

int MyClass::staticVar = 0;

int main() {
    MyClass::staticVar = 10; // 访问静态成员变量
    MyClass::staticFunc(); // 调用静态成员函数

    MyClass obj;
    obj.normalFunc(); // 调用普通成员函数
}
```

输出：

```
Static Function, staticVar: 10
Normal Function
```


问题13：什么是虚析构函数？为什么需要使用虚析构函数？

回答：

- **虚析构函数（Virtual Destructor）**：在基类中将析构函数声明为 `virtual`，确保通过基类指针删除派生类对象时，能够正确调用派生类的析构函数，释放派生类特有的资源。

- **为什么需要使用虚析构函数**：

 - **防止资源泄漏**：确保派生类的资源能够被正确释放。

 - **实现正确的对象销毁**：通过基类指针删除派生类对象时，先调用派生类析构函数，再调用基类析构函数。

示例：

```
class Base {
public:
    virtual ~Base() { // 虚析构函数
        std::cout << "Base Destructor" << std::endl;
    }
};

class Derived : public Base {
public:
    ~Derived() {
        std::cout << "Derived Destructor" << std::endl;
    }
};

int main() {
    Base* obj = new Derived();
    delete obj; // 输出: Derived Destructor -> Base Destructor
}
```

输出：

```
Derived Destructor
Base Destructor
```

注意事项：

- 如果类有虚函数，则应有虚析构函数。
- 即使类本身不打算被继承，良好的设计习惯也是为析构函数添加 `virtual`。

问题14：C++中如何实现动态绑定？它的作用是什么？

回答：

- **动态绑定（Dynamic Binding）**：在运行时根据对象的实际类型决定调用哪个函数，实现运行时多态。

- **实现方法**：
  - 使用虚函数（`virtual`关键字）在基类中声明函数。
  - 通过基类指针或引用调用虚函数，编译器根据对象的实际类型选择合适的函数实现。

作用：

- **实现多态**：允许通过基类接口调用不同子类的实现，增强代码的灵活性和可扩展性。
- **接口编程**：通过基类定义接口，不依赖具体的子类实现，便于代码维护和扩展。