

目录

目录

Q&A

`char*[]` 和 `char**` 有什么区别?

指针和引用有什么区别?

常量指针和指针常量之间有什么区别?

常量指针, 常引用, 常量之间有什么关系?

关于复制构造函数

对象间用等号赋值为什么不导致复制构造函数被调用?

引用指向的对象如果被销毁,那么这时候对于原引用的引用怎么办?

课上例题的解释说明(列表初始化)

在函数中定义了静态变量, 在递归调用该函数时,这个变量会有什么变化?

作业里出现的问题(类)

如果函数返回值的类型是常引用.引用.指针.变量,请对比他们的区别

关于常引用作为函数的返回值

为什么说赋值运算符 = 只能重载为成员函数?使用中文回复,通过代码给出例子,如果重载成普通函数会怎么样?

关于指针的一些疑惑(I)

关于指针的一些了解 (II)

复合关系中的循环定义是什么?如何避免?

C风格字符串和C++风格字符串

关于第二次上机的一道题目的讨论

C++补充知识

命令行参数

 命令行界面

 命令行重定向

位运算

负整数的表示方式

位运算

掩码

函数指针

 定义和使用

c语言的输入输出语句

```
int scanf(const char *,...);
```

```
int printf(const char *, ...);
```

```
char* gets(char * s)
```

C到C++

动态内存分配

引用

常引用

`const` 关键字的用法

内联函数

函数重载

函数缺省参数

面向对象编程(类)

类的介绍

访问权限

构造函数(Constructor)

 通用构造函数

 复制构造函数(拷贝构造函数)

 类型转换构造函数

析构函数(Destructor)

各种构造函数,析构函数的调用时机

- 通用构造函数（普通构造函数）
- 复制构造函数
- 类型转换构造函数
- 析构函数
- 赋值运算符函数
- 成员函数重载和参数默认
- this指针
 - c语言中类的实现
 - c++中的this
- 静态 `static`
- 常量成员函数
 - `mutable` 成员变量
- 成员对象和封闭类
 - 成员对象和封闭类
 - 函数的调用顺序
 - 封闭类的复制构造函数
- 友元
- 运算符重载
 - 自定义数据类型和运算符重载
 - 自定义数据类型
 - 运算符重载
 - 赋值运算符 `=` 重载
 - 关于重载运算符`=`的探寻
 - 运算符重载为友元
 - 流操作符的重载(插入,输出)
 - `operator[]` 的重载
 - 重载类型转化运算符
 - 自增自减运算符的重载
 - 运算符重载的注意事项
 - 运算符重载的应用
 - 作为类传入其他函数
- 继承和派生
 - 基本概念
 - 派生类的内存空间
 - 复合与继承
 - 权限说明符: `protected`
 - 生成与消亡

Q&A

`char*[]` 和 `char**` 有什么区别?

`char*[]` 是一个指向字符指针数组的指针，它可以看作是一个指针数组，每个元素都是一个指向字符的指针。

而 `char**` 是一个指向字符指针的指针，它可以看作是一个指向字符指针的指针，也可以表示一个指向字符串数组的指针。

因为 `char[]` 是一个指向字符数组头部的指针。

简单来说，`char*[]` 表示一个字符指针数组，而 `char**` 表示一个字符指针的指针。

指针和引用有什么区别？

引用和指针是C++中两种不同的概念，它们在使用和语法上有一些重要区别：

1. 定义和声明：

- 指针使用 `*` 符号来声明，例如：`int* ptr;`
- 引用使用 `&` 符号来声明，例如：`int& ref = var;`

2. 初始化：

- 指针可以在声明后再指向其他变量或空指针，例如：`int* ptr = nullptr;`，`ptr = &var;`
- 引用在声明时必须初始化，并且不能再指向其他变量，例如：`int& ref = var;`

3. 操作符：

- 指针通过 `*` 操作符来访问所指向的对象，例如：`*ptr = 10;`
- 引用直接访问所引用的对象，不需要使用操作符，例如：`ref = 10;`

4. 空值：

- 指针可以指向空值（`nullptr`），表示不指向任何对象。
- 引用必须在初始化时指向一个有效的对象，不能引用空值。

5. 重定义：

- 指针可以被重新赋值指向其他对象或空指针。
- 引用在初始化后不能再引用其他对象，它始终引用同一个对象。

6. 地址：

- 指针存储对象的地址，需要使用取地址运算符 `&` 获取地址。
- 引用在初始化时绑定到对象，不存储地址。

总的来说，指针提供了更多灵活性，可以指向不同的对象或空值，可以动态改变其指向的对象；而引用提供了更简洁的语法和更直接的访问方式，但不能重新绑定到其他对象。在选择使用指针还是引用时，可以根据具体情况和需求来决定。

常量指针和指针常量之间有什么区别？

常量指针和指针常量是两种不同的概念，它们之间有一定的区别。下面分别介绍常量指针和指针常量的概念和区别：

1. 常量指针（Constant Pointer）：

- 常量指针是指一个指针，其指向的对象是常量，不能通过该指针修改指向的对象的值。
- 声明常量指针时需要在指针前面加上 `const` 关键字。
- 常量指针本身的值（即指向的地址）是可以改变的，但指向的对象的值是不能改变的。

示例：

```
int x = 10;
const int* ptr = &x; // 常量指针，指向的对象是常量
// *ptr = 20; // 无法修改 ptr 指向对象的值
x = 20; // 可以修改 x 的值
```

2. 指针常量（Pointer Constant）：

- 指针常量是指一个指针本身是常量，即指针的值（即指向的地址）是不可修改的。
- 声明指针常量时需要在指针后面加上 `const` 关键字。
- 指针常量不能修改指针本身指向的地址，但可以通过该指针修改指向的对象的值。

示例：

```
int x = 10;
int* const ptr = &x; // 指针常量，指针本身是常量
*ptr = 20; // 可以修改 ptr 指向对象的值
// ptr = nullptr; // 无法修改 ptr 的值
```

综上所述，常量指针和指针常量的区别在于：常量指针指向的对象是常量，不能修改其值；而指针常量指针本身是常量，不能修改指针的值。在使用指针时，根据具体需求选择合适的类型可以帮助我们更好地管理内存和保证数据的安全性。

常量指针，常引用，常量之间有什么关系？

在C++中，常量指针、常引用和常量是三个涉及常量性（constness）的不同概念。它们之间的关系在于它们都用于限定数据的可修改性和访问方式，但应用的上下文和具体含义有所不同。

1. 常量（Constant）：

常量是指一旦初始化后其值就不能改变的变量。在C++中，可以使用 `const` 关键字来声明常量。例如：

```
const int constantValue = 10;
```

这里 `constantValue` 被初始化为10，之后它的值就不能被改变了。

2. 常量指针（Pointer to Constant）：

常量指针是指向常量的指针，这意味着通过这个指针不能改变它指向的值，但是指针本身的值（即存储的地址）可以改变，它可以指向别的地址。声明常量指针的语法如下：

```
const int* ptrToConstant = &constantValue;
```

这里 `ptrToConstant` 是一个指针，指向一个 `const int` 类型的数据。你 cannot 通过 `ptrToConstant` 来改变这个数据的值，但是 `ptrToConstant` 可以指向另一个 `const int` 类型的数据。

3. 指针常量（Constant Pointer）：

这是一个指针，它的值（即它所指向的地址）不可以改变，但是指向的数据的值可以改变（除非它指向的也是一个常量）。声明指针常量的语法如下：

```
int value = 5;
int* const constantPtr = &value;
```

这里 `constantPtr` 必须初始化，且之后不能再指向其他的地址，但是 `constantPtr` 指向的 `value` 是可以改变的。

4. 常引用（Constant Reference）：

常引用是对常量的引用，或者是对变量的只读引用。它不允许通过引用来修改它绑定的对象。常引用在声明时必须初始化，并且不能被重新绑定到另一个对象。声明常引用的语法如下：

```
const int& refToConstant = constantValue;
```

这里 `refToConstant` 是一个引用，它引用了一个 `const int` 类型的数据。通过 `refToConstant` 不能改变这个数据的值。

这三个概念的共同点是它们都利用 `const` 关键字来限制修改数据。区别在于指针常量（Constant Pointer）关注的是指针自身值的不变性，常量指针（Pointer to Constant）关注的是指针所指向的数据的不变性，而常引用（Constant Reference）则是引用的一种特殊形式，确保通过引用不能修改绑定的数据。

在实际编程实践中，这些概念用于提高代码安全性，防止非预期的修改，以及在接口设计中表达函数对输入参数的修改意图。

关于复制构造函数

Q:在函数列表和返回值中出现了类的对象是否会调用复制构造函数?如果会,那么如果我在类中自定义了复制构造函数,使得复制的结果并不等于复制原本,那么这时候调用复制构造函数来生成副本显然会出现错误,我的推测是否正确?

在C++中，当函数列表中出现类的对象时，拷贝构造函数可能被调用。但是，编译器会尽可能地通过优化来避免额外的拷贝。这种优化通常称为（Named）Return Value Optimization（(N)RVO）或Copy Elision。在C++17标准中，这种优化在某些情况下是强制性的。

当你通过值将对象传递给函数时，拷贝构造函数会被调用来创建参数的本地副本。例如：

```
void foo(MyClass obj) {  
    // ...  
}  
  
MyClass myObj;  
foo(myObj); // 调用 MyClass 的拷贝构造函数来创建 foo 的参数 obj 的副本
```

如果参数是通过引用传递的，无论是`const`引用还是非`const`引用，都不会调用拷贝构造函数。

当函数返回一个对象时，理论上也会调用拷贝构造函数来创建返回值的副本。但是，实际上编译器通常会应用 (N)RVO，从而避免这些额外的拷贝。在C++17及更高版本中，这种优化是保证的，在某些情况下拷贝构造函数的调用可以省略。

如果你在类中自定义了拷贝构造函数，并且使得复制的结果与原对象不同，这通常是不推荐的设计。拷贝构造函数的预期行为是创建一个与原对象状态相同的新对象。如果你违反了 this 预期，那么当拷贝构造函数被调用时（如在函数参数传递或返回值时），可能会出现逻辑错误或不一致的行为。

例如，如果你的自定义拷贝构造函数故意改变了成员变量的值，那么在函数调用中创建的副本将与原始对象有所不同。这可能导致程序的其它部分出现不可预料的行为，因为它们可能依赖于对象的拷贝与原对象具有相同的状态。

这里是一个示例：

```

class MyClass {
public:
    int value;

    MyClass(int val) : value(val)    // 自定义拷贝构造函数, 故意改变了 value 的值
    MyClass(const MyClass& other) : value(other.value + 1)};

    void foo(MyClass obj) {
        // 如果 MyClass 的实例是通过拷贝传递的, 则 obj.value 将不同于传入对象的 value
    }

    MyClass a(5);
    foo(a); // foo 中的 obj.value 将会是 6, 而不是 5
}

```

在这个例子中, `foo` 函数接收的参数并不反映传入对象 `a` 的实际状态, 这可能导致错误或者混乱。

在设计类时, 拷贝构造函数应该保持对象状态的一致性, 确保对象的拷贝是原对象的精确副本。如果你需要不同的复制行为, 你应该考虑使用不同的函数或操作符来明确这种行为, 而不是改变拷贝构造函数的语义。这样做可以确保代码清晰和可预测, 同时符合其他程序员的期望。

对象间用等号赋值为什么不导致复制构造函数被调用?

在C++中, 对象间的等号赋值操作并不会导致拷贝构造函数被调用, 而是会触发拷贝赋值操作符(Copy Assignment Operator)的调用。这两者是不同的概念和操作。

拷贝构造函数 的调用场景包括:

1. 当使用一个已存在的对象来初始化一个新对象时 (例如, `MyClass obj1 = obj2;`)。
2. 当对象作为参数通过值传递给函数时 (尽管编译器优化可能会省略这个过程)。
3. 当函数返回一个对象时 (同样, 编译器优化可能会省略这个过程)。

拷贝赋值操作符 的调用场景是:

1. 当一个已经存在的对象被用来给另一个已经存在的对象赋值时 (例如, `obj1 = obj2;`)。

以下是两者的一个简单示例来说明区别:

```

class MyClass {
public:
    int value;

    // 构造函数
    MyClass(int val) : value(val)    // 拷贝构造函数
    MyClass(const MyClass& other) : value(other.value) {
        // 这里是拷贝构造函数的实现, 它会在新对象初始化时被调用
    }

    // 拷贝赋值操作符
    MyClass& operator=(const MyClass& other) {
        if (this != &other) { // 防止自赋值
            this->value = other.value;
        }
        return *this;
        // 这里是拷贝赋值操作符的实现, 它会在对象之间的赋值时被调用
    }
}

```

```
};

int main() {
    MyClass a(10);           // 构造函数被调用
    MyClass b = a;           // 拷贝构造函数被调用, 因为初始化新对象b
    MyClass c(20);           // 构造函数被调用
    c = a;                   // 拷贝赋值操作符被调用, 因为c已存在, 现在只是赋新值
    return 0;
}
```

在这个示例中:

- `MyClass b = a;` 使用了 `a` 来初始化 `b`, 所以这里调用了拷贝构造函数。
- `c = a;` 是赋值操作, 因为 `c` 已经被构造了, 所以这里调用了拷贝赋值操作符。

这种区分是很重要的, 因为它影响着类的设计, 特别是当涉及到资源管理(如动态内存分配)时。如果你有需要管理的资源, 你需要确保正确实现拷贝赋值操作符和拷贝构造函数(以及移动构造函数和移动赋值操作符, 如果你的类需要支持移动语义)。

引用指向的对象如果被销毁,那么这时候对于原引用的引用怎么办?

引用一旦被初始化就不能再引用其他的对象了,即完全绑定

在C++中, 引用本身不能被显式销毁。引用一旦被初始化后, 就会一直绑定到它最初被绑定的对象上, 直到这个引用的作用域结束。引用的生命周期是由它所引用的对象的作用域决定的。当对象的作用域结束时, 对象会被销毁, 但是引用本身并没有销毁的概念——它只是失去了有效的绑定对象。

考虑下面的示例:

```
int main() {
    int a = 5;
    int& ref_a = a; // ref_a 是对变量 a 的引用
    // ...
} // ref_a 的作用域在这里结束, 但我们不需要(也不能)显式销毁它
```

当 `main` 函数结束执行时, `a` 和 `ref_a` 的作用域都结束了。`a` 作为一个局部变量会被自动销毁。由于 `ref_a` 是 `a` 的引用, 所以在 `a` 被销毁时, `ref_a` 自然也就没有了它所引用的对象。

引用和它所引用的对象之间的这种关系是由编译器在编译时期就决定的, 而不是运行时可以改变的。因此, 没有一个操作或函数可以用来“销毁”一个引用, 正如你不能改变一个引用一旦绑定后所指向的对象一样。

在某些高级语言中, 如Java或C#, 引用可以被赋予新的对象或者赋值为 `null` 来“断开”引用, 但在C++中这是不可能的。如果你需要更灵活的引用关系, 可以考虑使用指针, 指针可以被重新赋值来指向另一个对象, 或者赋值为 `nullptr` 来表示它不指向任何对象。

悬垂引用 (Dangling Reference) 是指引用已经释放或者失效的内存的引用。当对象被销毁或者释放后, 指向它的引用仍然存在, 但是再通过这个引用访问对象就会导致未定义的行为, 因为引用的内存地址已经不再有效或者可能已经被重新分配给其他用途。

这里有几个导致悬垂引用的常见情况:

1. **局部变量的引用返回:** 如果你从一个函数返回一个局部变量的引用, 那么当函数退出时, 局部变量的存储空间将被释放, 任何保留的引用都将成为悬垂引用。

```
int& func() {
    int local = 42;
    return local; // 返回局部变量的引用，会产生悬垂引用
}
```

2. 释放动态内存后的引用：如果你释放了一个对象的动态分配的内存（例如使用 `delete`），但是保留了一个指向该内存的引用，该引用也将成为悬垂引用。

```
int* ptr = new int(42); // 动态分配内存
int& ref = *ptr;         // 创建指向动态分配内存的引用
delete ptr;              // 释放内存
// 'ref' 现在成为悬垂引用
```

3. 指向对象成员的引用：如果你有一个引用指向对象的一个成员，然后删除了整个对象，那么指向成员的引用也会变成悬垂引用。

```
class MyClass {
public:
    int value;
    MyClass(int val) : value(val) {}

    MyClass* myObject = new MyClass(42);
    int& ref = myObject->value;
    delete myObject;
    // 'ref' 现在指向已经被释放的内存
```

4. 迭代器失效：在C++的STL容器中，迭代器可能因为容器的修改操作（如插入、删除元素）而失效。失效的迭代器可以被视为一种悬垂引用。（可以通过为迭代器重新赋值使其指向一个有效的位置来修正这个错误）

```
std::vector<int> vec = {1, 2, 3};
auto it = vec.begin(); // 'it' 是指向 vec 第一个元素的迭代器
vec.clear();           // 清空 vec, 'it' 现在变成了悬垂迭代器
```

避免悬垂引用：

- 不要返回局部变量的引用或指针。
- 确保在引用或指针失效前不再使用它们。
- 当使用动态内存时，合理安排对象的生命周期，并确保在对象的生命周期结束后不再使用其引用。
- 使用智能指针（如 `std::unique_ptr` 或 `std::shared_ptr`）来管理动态内存，以避免手动管理内存的复杂性。

悬垂引用是C++编程中一个常见的问题，它可以导致程序崩溃或不可预测的行为，因此理解它们并知道如何避免是非常重要的。

课上例题的解释说明(列表初始化)

```
#include <cstdint>
#include <iostream>

using namespace std;
```



```

class A {
    int v;

public:
    A(int n) { v = n; }
    A() { cout << "Build Function Called" << endl; }
};

int main() {
    A *arr[4] = {new A(), NULL, new A()};
    return 0;
}

```

Q:这是一种什么样的类对象构造方式?请为我详细介绍这种构造方式,并解释为什么只创建了两个对象。

A:这是一种使用列表初始化的方式来构造类对象数组。在这个例子中,我们创建了一个指向类 A 对象的指针数组 `arr`,数组的大小为4。通过列表初始化,我们可以在数组创建时直接指定每个元素的初始值。

让我详细介绍这种构造方式:

1. 列表初始化:

- 在C++11中引入了列表初始化的概念,允许使用花括号 `{}` 来初始化数组、向量等容器。
- 通过在花括号内提供初始值列表,可以在创建数组时直接指定每个元素的初始值。
- 列表初始化可以用于各种类型的数组,包括内置类型数组和对象数组。

```

#include <iostream>
using namespace std;
class A {
    int v;

public:
    A(int n) {
        v = n;
        cout << "Build Function Called" << endl;
    }
    A() { cout << "Build Function Called" << endl; }
};

int main() {
    A arr[4] = {1, 2}; // 创建了4个对象
    return 0;
}

```

2. 指针数组:

- 在这个例子中,我们创建了一个指向类 A 对象的指针数组 `arr`。
- 数组的每个元素都是一个指向 A 对象的指针,而不是 A 对象本身。
- 通过列表初始化,我们可以在创建数组时直接指定每个指针元素的初始值。

3. 对象创建:

- 在列表初始化中,我们使用 `new` 操作符来动态创建 A 类的对象。
- `new A()` 会调用 A 类的默认构造函数来创建一个新的对象,并返回指向该对象的指针。
- 我们将这些指针作为初始值赋给指针数组 `arr` 的相应元素。

4. 初始化列表:

- 在这个例子中,初始化列表为 `{new A(), NULL, new A()}`。
- 第一个元素 `new A()` 会创建一个新的 `A` 对象,并将其指针赋给 `arr[0]`。
- 第二个元素 `NULL` 是一个空指针,表示 `arr[1]` 不指向任何对象。
- 第三个元素 `new A()` 会创建另一个新的 `A` 对象,并将其指针赋给 `arr[2]`。

5. 未初始化的元素:

- 在这个例子中,数组 `arr` 的大小为4,但初始化列表中只提供了3个值。
- 对于未显式初始化的元素 `arr[3]`,它会被默认初始化为 `nullptr`,即空指针。(同`arr[1]`)

现在,让我解释为什么只创建了两个对象:

- 在初始化列表中,我们只使用 `new A()` 创建了两次 `A` 类的对象。
- 第一个 `new A()` 创建了一个对象,并将其指针赋给 `arr[0]`。
- 第二个 `new A()` 创建了另一个对象,并将其指针赋给 `arr[2]`。
- 初始化列表中的 `NULL` 表示 `arr[1]` 不指向任何对象,它只是一个空指针。
- 未显式初始化的 `arr[3]` 被默认初始化为 `nullptr`,也不指向任何对象。

因此,虽然数组 `arr` 的大小为4,但实际上只有 `arr[0]` 和 `arr[2]` 指向了动态创建的 `A` 类对象,而 `arr[1]` 和 `arr[3]` 为空指针。所以,只有两个 `A` 类对象被创建。

需要注意的是,使用动态分配的对象时,需要注意的是,使用动态分配的对象时,我们有责任在不再需要这些对象时手动释放内存。在上述例子中,我们使用 `new` 操作符动态创建了对象,但没有使用 `delete` 操作符释放内存,这可能会导致内存泄漏。

为了正确管理动态分配的对象,我们应该在适当的时候调用 `delete` 操作符来释放内存。例如:

```
int main() {
    A *arr[4] = {new A(), NULL, new A()};

    // 使用 arr 数组进行操作

    // 释放动态分配的对象
    delete arr[0];
    delete arr[2];

    return 0;
}
```

在这个例子中,我们在使用完 `arr` 数组后,通过 `delete arr[0]` 和 `delete arr[2]` 释放了动态分配的对象。这样可以避免内存泄漏。

另外,还有一些其他的注意事项:

1. 初始化顺序:

- 列表初始化按照初始值的顺序来初始化数组元素。
- 在上述例子中, `arr[0]` 首先被初始化为 `new A()` 创建的对象指针,然后 `arr[1]` 被初始化为 `NULL`,以此类推。

2. 默认构造函数:

- 在创建 `A` 类的对象时,我们使用了默认构造函数 `A()`。

- 默认构造函数没有参数,它会输出一条消息表示构造函数被调用。
- 如果类没有显式定义默认构造函数,编译器会自动生成一个默认构造函数。

3. 构造函数的调用:

- 当我们使用 `new A()` 创建对象时,会调用 `A` 类的默认构造函数。
- 在上述例子中,默认构造函数输出了一条消息,表示构造函数被调用。
- 由于我们创建了两个对象,所以默认构造函数被调用了两次,输出了两条消息。

总结起来,这种使用列表初始化构造类对象数组的方式允许我们在创建数组时直接指定每个元素的初始值。通过使用 `new` 操作符,我们可以动态创建类的对象,并将其指针赋给数组元素。需要注意的是,我们要负责在适当的时候释放动态分配的对象,以避免内存泄漏。同时,列表初始化按照初始值的顺序来初始化数组元素,未显式初始化的元素会被默认初始化为空指针。

在函数中定义了静态变量，在递归调用该函数时,这个变量会有什么变化?

在C++函数中定义的静态变量具有以下特性:

1. 它们在程序执行期间只被初始化一次。
2. 它们的生命周期持续到程序结束。
3. 它们的作用域仅限于包含它们的函数，但是它们在函数调用之间保持其值。

当一个函数包含一个静态变量并且这个函数被递归调用时，无论这个函数被调用多少次，静态变量都不会重新初始化。它将保留上一次函数调用结束时的值。这意味着递归调用中的每个函数实例都会看到相同的静态变量，而不是每个调用都有自己的独立副本。

这个特性可以在需要保持跨多个递归调用的状态时非常有用，比如计数器或累加器。

下面是一个简单的例子，说明了递归函数中静态变量的行为：

```
#include <iostream>

void recursiveFunction(int n) {
    static int count = 0; // 静态变量只初始化一次
    count++;
    std::cout << "In call " << count << ": n = " << n << std::endl;

    if (n > 0) {
        recursiveFunction(n - 1);
    }
}

int main() {
    recursiveFunction(3);
    return 0;
}
```

输出将会是：

```
In call 1: n = 3
In call 2: n = 2
In call 3: n = 1
In call 4: n = 0
```

如你所见，即使函数 `recursiveFunction` 被多次调用，变量 `count` 也只在被初始化一次，并在每次递归调用中累加。每个调用都不会重置 `count`，而是从上一次调用保留的值开始计数。这是因为静态局部变量在栈帧间是共享的，而不是每个栈帧一个独立的副本。

作业里出现的问题(类)

```
#include <iostream>
#include <string>
using namespace std;
class Base {
public:
    int k;
    Base(int n) : k(n) {}
};
class Big {
public:
    int v;
    Base b; // 在提供了只有参数的构造函数的情况下, 为什么这里b能被创建?
    Big(int n) : v(n), b(n) {} // 为什么这种形式能让它重新初始化?
};
int main() {
    int n;
    while (cin >> n) {
        Big a1(n);
        Big a2 = a1;
        cout << a1.v << ", " << a1.b.k << endl;
        cout << a2.v << ", " << a2.b.k << endl;
    }
}
```

如果函数返回值的类型是常引用.引用.指针.变量,请对比他们的区别

在C++中,函数返回值可以是常引用、引用、指针或变量。下面我将对比它们的区别:

1. 常引用 (const reference)

- 返回一个常引用意味着返回值是一个对常量的引用。
- 常引用返回的是一个不可修改的引用,无法通过该引用修改原始对象。
- 常引用可以引用临时对象、字面值等。
- 常引用可以避免不必要的对象拷贝,提高性能。

2. 引用 (reference)

- 返回一个引用意味着返回值是一个对变量的引用。
- 通过返回的引用可以直接修改原始对象。
- 引用必须绑定到一个已存在的对象,不能引用临时对象或字面值。
- 引用可以避免对象拷贝,提高性能。

3. 指针 (pointer)

- 返回一个指针意味着返回一个指向对象的内存地址。
- 通过返回的指针可以间接访问和修改原始对象。
- 指针可以为空 (`nullptr`),需要在使用前进行空指针判断。

- 指针可以表示对象的所有权转移,但需要手动管理内存。

4. 变量 (variable)

- 返回一个变量意味着返回值是一个独立的对象副本。
- 返回的变量与原始对象是完全独立的,对返回值的修改不会影响原始对象。
- 返回变量时会发生对象拷贝,可能会有一定的性能开销。
- 返回变量适用于返回小的、简单的对象或内置类型。

选择函数返回值类型取决于具体的需求和设计考虑:

- 如果返回值不需要被修改,且可能引用临时对象,则使用常引用。
- 如果返回值需要被修改,且引用的对象在函数外部有效,则使用引用。
- 如果返回值表示对象的所有权转移或可能为空,则使用指针。
- 如果返回值是一个独立的副本,且对象较小或是内置类型,则使用变量。

需要根据实际情况权衡性能、安全性和语义的需求来选择合适的返回值类型。

关于常引用作为函数的返回值

如果一个函数的返回值类型为常引用,但引用的是函数内部新建的变量,那么:

1. 这个常引用在函数结束后是否会成为垂悬引用?
2. 如果不会成为垂悬引用,是否意味着引用的变量在被使用之后函数才会结束?
3. 这样做与直接返回变量本身有什么区别?

如果一个函数返回一个在函数内部新建变量的常引用,那么这个引用在函数结束后确实会成为**垂悬引用 (dangling reference)**。这是因为当函数返回时,函数内部的局部变量会被销毁,而返回的引用仍然指向那个已经被销毁的变量。

访问垂悬引用是未定义行为 (undefined behavior),可能会导致程序崩溃或产生意想不到的结果。即使返回的是常引用,也无法避免这个问题,因为常引用只是禁止通过引用修改对象,并不能延长对象的生命周期。

示例:

```
const int& getNumber() {  
    int num = 42;  
    return num; // 返回局部变量的常引用  
}  
  
int main() {  
    const int& ref = getNumber(); // ref 成为了垂悬引用  
    // 使用 ref 会导致未定义行为  
    return 0;  
}
```

正确的做法是直接返回变量本身,而不是返回引用。当函数返回一个变量时,会创建一个临时对象来存储返回值,而这个临时对象在表达式的生命周期内是有效的

修改后的示例:

```
int getNumber() {
    int num = 42;
    return num; // 返回变量本身
}

int main() {
    int result = getNumber(); // 获取函数返回的值
    // 使用 result 是安全的
    return 0;
}
```

直接返回变量自身有以下区别和优点:

1. 避免了垂悬引用的问题,因为返回的是一个独立的对象副本。
2. 返回值的生命周期与表达式的生命周期相同,不依赖于函数内部的局部变量。
3. 返回值可以被直接使用和修改,不受常量性的限制。

总之,在函数内部动态创建的变量,应该直接返回变量本身,而不是返回引用,以避免垂悬引用的问题。返回常引用的目的是为了

在刚才的问题之中,当返回常引用之后,函数是否已经结束,并且销毁了函数中生成的临时变量?

当函数返回常引用后,函数确实已经结束执行,并且销毁了函数中生成的临时变量。

在C++中,当函数返回时,会执行以下步骤:

1. 计算返回值表达式。
2. 销毁函数内部的局部变量。
3. 将返回值传递给调用方。
4. 函数执行结束。

当函数返回常引用时,第一步会计算返回值表达式,即获取要返回的引用。但是,这个引用所引用的对象(函数内部新建的变量)在第二步中就会被销毁。然后,函数将这个引用返回给调用方,但实际上引用已经指向了一个不再存在的对象。

因此,在函数返回之后,通过常引用访问函数内部新建的变量是危险的,因为该变量已经被销毁,访问它会导致未定义行为。

示例:

```
const int& getNumber() {
    int num = 42;
    return num; // 返回局部变量的常引用
}

int main() {
    const int& ref = getNumber(); // ref 成为了垂悬引用
    // 使用 ref 会导致未定义行为,因为 num 已经被销毁
    return 0;
}
```

在上面的示例中,当 `getNumber()` 函数返回时,局部变量 `num` 已经被销毁,但 `ref` 仍然引用了 `num`。在函数返回后访问 `ref` 会导致未定义行为。

总之,函数返回常引用后,函数已经结束,并销毁了函数内部生成的临时变量。返回的常引用如果引用了函数内部的变量,就会成为垂悬引用,访问它会导致未定义行为。因此,应该避免返回函数内部新建变量的引用,而是直接返回变量本身。

如果是返回变量,那么创建副本是在什么环节创建的?

你在之前的回复中提到

在C++中,当函数返回时,会执行以下步骤:

1. 计算返回值表达式。
2. 销毁函数内部的局部变量。
3. 将返回值传递给调用方。
4. 函数执行结束。

创建副本是否在这四个环节中的其中一个,还是说这个流程的表述是不准确的?

你提出了一个很好的问题。我在之前的回复中给出的函数返回步骤描述确实不够准确,没有明确指出创建副本的环节。让我来更正和补充说明。

在C++中,当函数返回非引用类型(如变量)时,会执行以下步骤:

1. 计算返回值表达式。
2. 在调用方的空间创建一个临时对象,用于存储返回值。
3. 将返回值表达式的结果复制或移动到临时对象中。
4. 销毁函数内部的局部变量。
5. 函数执行结束,返回临时对象。

创建副本的环节发生在第2步和第3步之间。当函数返回一个变量时,会在调用方的空间创建一个临时对象,然后将返回值表达式的结果复制或移动到这个临时对象中。这个临时对象就是返回值的副本。

示例:

```
int getNumber() {
    int num = 42;
    return num; // 返回变量本身
}

int main() {
    int result = getNumber(); // 获取函数返回的值
    // 使用 result 是安全的
    return 0;
}
```

在上面的示例中,当 `getNumber()` 函数返回时,会在 `main()` 函数的空间创建一个临时对象,并将 `num` 的值复制到这个临时对象中。然后,函数销毁局部变量 `num`,并将临时对象作为返回值传递给 `result`。

需要注意的是,在某些情况下,编译器可能会进行返回值优化(RVO)或命名返回值优化(NRVO),以避免不必要的复制。这些优化会在可能的情况下直接在调用方的空间构造返回值,而不是创建临时对象。但即使进行了优化,返回的也是一个独立的对象,而不是函数内部的变量。

总之,当函数返回变量时,会在调用方的空间创建一个临时对象,并将返回值复制或移动到该临时对象中。这个临时对象就是返回值的副本,它的生命周期与表达式的生命周期相同。访问该副本是安全的,不会导致垂悬引用的问题。

为什么说赋值运算符 = 只能重载为成员函数?使用中文回复,通过代码给出例子,如果重载成普通函数会怎么样?

在C++中，赋值运算符 `=` 必须重载为成员函数，这是因为赋值运算符至少需要一个操作数是调用它的对象本身（即左值必须是对象本身），而成员函数可以保证第一个操作数（即 `*this`）是一个类的实例。如果将赋值运算符重载为非成员函数（即全局函数），那么左值可能不是类的实例，这与赋值操作的基本要求相违背。

下面是一个简单的类 `MyClass`，其中赋值运算符被重载为成员函数的示例：

```
class MyClass {
public:
    int value;

    MyClass(int val) : value(val)    // 赋值运算符重载为成员函数
    MyClass& operator=(const MyClass& other) {
        if (this != &other) { // 避免自赋值
            this->value = other.value;
        }
        return *this;
    }
};

int main() {
    MyClass a(1);
    MyClass b(2);
    a = b; // 调用赋值运算符
    return 0;
}
```

如果尝试将赋值运算符重载为非成员函数，例如：

```
class MyClass {
public:
    int value;

    MyClass(int val) : value(val)    // 假设我们尝试将赋值运算符重载为非成员函数
};

// 错误的尝试：赋值运算符作为非成员函数
MyClass operator=(MyClass& lhs, const MyClass& rhs) {
    lhs.value = rhs.value;
    return lhs;
}
```

上面的尝试是不允许的，编译器会报错，因为赋值运算符必须是成员函数。事实上，如果你尝试这么做，编译器会提供类似于以下的错误消息：

```
error: 'MyClass MyClass::operator=(const MyClass&)' must be a non-static member function
```


这意味着赋值运算符必须是非静态的成员函数。因此，在C++中，赋值运算符只能被重载为成员函数。这样设计的原因之一是，赋值操作涉及到对象状态的改变，而成员函数直接对类的内部状态具有访问权限，这使得成员函数成为管理对象状态变化的合适方式。

关于指针的一些疑惑(I)

如果在 `cout<<ptr<<endl;`

这里的 `ptr` 是一个字符类型的指针,则在这里它会被认为是一个字符数组的首地址并尝试访问这个数组从而完成输出操作。

是的，当 `ptr` 是一个字符类型的指针并且使用 `cout << ptr` 进行输出时，C++ 的 `iostream` 库会将 `ptr` 解释为指向以空字符 `\0` 结尾的 C 风格字符串的指针。`cout` 会从指针指向的地址开始输出字符直到遇到空字符为止。

例如：

```
char* ptr = new char[10];
strcpy(ptr, "Hello");
cout << ptr << endl;
```

这段代码会输出字符串 "Hello" 后跟一个换行符。这是因为 `strcpy` 函数将一个以 `\0` 结尾的字符串复制到了 `ptr` 指向的内存中，而 `cout` 检测到 `ptr` 是一个 `char*` 类型，就尝试将它作为 C 风格字符串输出。

重要的是要确保 `ptr` 确实指向一个有效的以 `\0` 结尾的字符数组。如果没有空字符来标记字符串的结束或 `ptr` 指向的内存没有被正确初始化，那么 `cout` 将会继续输出内存中的连续字符，直到偶然遇到一个 `\0`，这可能导致未定义行为，包括输出垃圾数据或导致程序崩溃。

对于指针使用[]操作符访问,则编译器会将这个指针认为是一个数组,并尝试访问索引位置的元素,这个说法正确吗?

是的，这个说法基本上是正确的。在 C++ 中，当你对指针使用下标操作符（`[]`），编译器会按照数组访问的方式来处理该指针。具体来说，表达式 `ptr[index]` 等价于 `*(ptr + index)`。

当你写下 `ptr[index]`，编译器会计算出从 `ptr` 开始，向前移动 `index` 个元素的位置，并解引用该位置的指针，以得到存储在那里的值。这里的 `index` 通常是一个整数值。

这种行为适用于任何类型的指针，包括字符指针。例如：

```
int* ptr = new int[10];
ptr[3] = 5; // Sets the fourth element (index 3) to 5
```

在这个例子中，`ptr` 是一个指向 `int` 的指针，指向一个动态分配的包含 10 个整数的数组。`ptr[3] = 5;` 这行代码实际上是将 `ptr` 所指向的数组的第四个元素（从 0 开始计数）设置为 5。

同样的概念也适用于字符指针，这也是为什么可以将字符指针当作指向字符串的数组来处理。但请记住，这种数组访问方式并不提供越界检查，因此你需要确保你的下标操作是在有效的范围内。

类似的将指针视作是数组的操作还有哪些？

在C++中，指针被广泛地用作数组，因为数组名在大多数表达式中会被转换为指向其第一个元素的指针。这里有一些常见的将指针视作数组的操作：

1. 下标访问（Subscripting）：

如前所述，使用 `ptr[index]` 可以访问指针 `ptr` 指向的数组中的第 `index` 个元素。

2. 指针算术 (Pointer Arithmetic) :

- 使用 `ptr + index` 可以获得指针 `ptr` 向后移动 `index` 个元素的位置。
- 使用 `ptr - index` 可以获得指针 `ptr` 向前移动 `index` 个元素的位置。
- 两个指针可以相减, `ptr2 - ptr1` 会返回两个指针之间的元素数量。

3. 迭代 (Iteration) :

通过指针可以遍历数组。例如, 使用循环可以通过指针递增来遍历数组的每个元素:

```
for (int* p = arr; p != arr + n; ++p) {  
    // 使用 *p 访问元素  
}
```

其中 `arr` 是数组名, `n` 是数组中的元素数量。

4. 与C标准库函数的交互 (Interaction with C Standard Library Functions) :

许多C标准库函数 (如 `memcpy`, `strcmp`, `strcpy` 等) 需要指针参数来指代数组。

5. 动态内存管理 (Dynamic Memory Management) :

使用 `new[]` 和 `delete[]` 运算符分配和释放内存时, 返回的是指向分配数组的指针。

6. 函数参数 (Function Parameters) :

当数组作为参数传递给函数时, 它被转换为指向其第一个元素的指针。函数内部可以使用该指针遍历数组元素。

7. 返回数组 (Returning Arrays from Functions) :

由于C++不允许直接返回局部数组, 所以通常会返回指向数组的指针。

请记住, 在使用指针作为数组时, 你必须保证指针指向的内存是有效的, 且你的操作不会造成越界, 这是因为C++不会自动进行边界检查。对于越界的内存访问, 程序可能会崩溃或产生未定义行为。

关于指针的一些了解 (II)

请告诉我 `int*` 和 `int*` 和 `int**` 的区别和使用方法, 以此类推, 总结不同类型指针的使用方法

为我说明它们的内存释放方式, 假设在我没有指明它指向的内存大小的时候, 以及多级指针指向的指针指向内存大小未知的时候

在C或C++语言中, `int*`、`int**` 和 `int***` 是指针类型, 它们分别是指向整数、指向整数指针和指向整数指针的指针。

这里简要解释一下它们的含义和基本用法:

1. `int*` 是一个指向整型的指针。它可以存储一个整型变量的地址。

```
int a = 10;  
int* p = &a; // p 是一个指向整数的指针, 存储了 a 的地址
```

2. `int**` 是一个指向 `int*` 的指针, 即它可以存储一个整型指针的地址。

```
int* p = &a;  
int** pp = &p; // pp 是一个指向指针的指针, 存储了 p 的地址
```

3. `int***` 是一个指向 `int**` 的指针, 即它可以存储一个指向整型指针的指针的地址。

```
int** pp = &p;
int*** ppp = &pp; // ppp 是一个指向指针的指针的指针，存储了 pp 的地址
```

使用这些指针时，你需要考虑以下两点：

- 指针的解引用（dereference）：通过使用解引用操作符 `*` 你可以访问指针指向的值。对于 `int**` 或 `int***` 等多级指针，你需要多次解引用来获取最终的值。
- 指针的内存管理：分配给指针的内存需要适当地释放，以防止内存泄露。

内存释放方式取决于内存是如何分配的。如果你使用 `malloc` 或 `new` 分配了内存，你必须使用 `free` 或 `delete` 来释放它。对于多级指针，你需要释放指向的每一级分配的内存。

例如：

```
int* p = new int;           // 分配一个整数
int** pp = new int*;        // 分配一个指针
*pp = new int;              // 分配另一个整数，并让 pp 指向它

delete *pp; // 释放 pp 所指向的整数
delete pp;   // 释放指针本身
delete p;    // 释放第一个整数
```

如果你没有分配固定大小的内存，那么你必须要知道内存的大小以正确地释放它。对于动态分配的数组，你应该使用 `delete[]` 而不是 `delete`。

```
int* p = new int[10]; // 分配10个整数的数组
// ...
delete[] p; // 释放整个数组
```

当涉及到多级指针，尤其是指向动态分配数组的指针时，释放内存变得更加复杂。你需要首先释放最内层的指针指向的内存，然后依次向外释放：

```
int** pp = new int*[10]; // 分配10个指针
for (int i = 0; i < 10; ++i) {
    pp[i] = new int[10]; // 分配10个整数的数组给每个指针
}
// ...
for (int i = 0; i < 10; ++i) {
    delete[] pp[i]; // 释放每个指针指向的数组
}
delete[] pp; // 最后释放指针数组本身
```

在多级指针的情况下，如果你不知道指向的指针指向的内存大小，就没有通用的方法来释放内存。你必须跟踪分配的大小，这样才能正确地释放它。通常，这涉及到记录分配的大小或使用数据结构（如动态数组、链表等）来管理动态分配的内存。

复合关系中的循环定义是什么?如何避免?

在C++中，如果两个类相互引用对方，且至少一个类需要知道另一个类的完整定义，那么会出现循环定义的问题。

这是因为在C++中，一个类的定义需要在使用该类的成员之前完全可见。

为了理解为什么使用指针可以避免循环定义，我们需要先理解C++中类的声明（forward declaration）和定义的区别。

- **类的声明** 只是告诉编译器这个类的名字，而不需要知道类的细节。
- **类的定义** 描述了类的全部结构，包括它的所有成员。

当你有两个类**A**和**B**，它们互相含有对方的实例时，会出现编译错误，因为编译器需要知道被包含对象的完整大小以分配内存。但是如果你使用指针，情况就不同了。

使用指针的关键优势是指针的大小在编译时是已知的，不管它指向的类的定义是否完整。这意味着你可以在类A中包含指向类B的指针，即使类B尚未完全定义，反之亦然。在这种情况下，你只需要在类A中前向声明类B，不需要其完整定义。

这里是一个简单的例子：

```
class B; // 前向声明，足够声明指向B的指针

class A {
    B* b; // 使用前向声明的类型的指针
public:
    void setB(B* b);
};

class B {
    A* a; // 此时A类已定义，可以使用指针
public:
    void setA(A* a);
};
```

在上面的代码中，即使A和B类互相引用，也没有循环定义的问题，因为它们使用了指针而不是直接包含对象。使用指针意味着你只需要知道被引用类的名字，而不需要知道它的大小或布局。这允许你在类的定义中包含对不完全类型的指针，从而避免了编译错误。

C风格字符串和C++风格字符串

C++字符串和C字符串在概念和用法上有很大的区别，下面我会分别介绍它们。

C字符串

在C语言中，字符串是以空字符 `'\0'` 结尾的字符数组。这意味着C字符串是一连串的字符，内存中排列在一起，最后以空字符作为结束标志。C字符串的操作通常通过标准库中的一系列函数来实现，比如 `strcpy()`、`strcat()`、`strlen()` 等。这些函数定义在 `<string.h>` 头文件中。

例如，定义和初始化一个C字符串可以这样写：

```
char c_string[] = "Hello, World!";
```

这行代码会在内存中创建一个数组，包含了字符串 `"Hello, World!"` 的每个字符和一个额外的空字符 `'\0'` 来标示字符串的结束。

操作C字符串需要格外小心，因为它们容易出现缓冲区溢出和其他安全问题。例如，当使用 `strcpy()` 复制字符串时，如果目标缓冲区大小不足以容纳源字符串，就会发生溢出。

C++字符串

C++提供了 `std::string` 类，它是C++标准库（STL）的一部分，定义在 `<string>` 头文件中。

`std::string` 管理和操作字符串的工作更加直观和安全。`std::string` 对象会自动管理内存，用户不需要担心内存分配和释放的问题。

例如，定义和初始化一个C++字符串可以这样写：

```
#include <string>

std::string cpp_string = "Hello, World!";
```

`std::string` 类提供了许多成员函数来执行常见操作，如 `append()`、`insert()`、`find()`、`substr()` 等，这些操作比C字符串的相应操作更加安全和方便。

此外，`std::string` 支持操作符重载，如 `+` 用于连接字符串，`==` 用于比较字符串等，使得代码更加清晰易懂。

总结

C字符串是基于字符数组的，必须以空字符结束，使用时需要手动管理内存和注意安全问题。C++字符串则是一个类，提供了大量方便的成员函数和操作符重载，自动管理内存，使用更加安全方便。在C++中，除非有特殊需要，通常推荐使用 `std::string` 来处理字符串。

关于第二次上机的一道题目的讨论

0208:编程填空：a+b+c问题

- 总时间限制:

1000ms

- 内存限制:

65536kB

- 描述

完善代码，使其能够按照指定方式输出

```
#include <iostream>
using namespace std;
// 在此处补充你的代码
int main() {
    int t;
    cin >> t;
    while (t--){
        int aa, bb, cc;
        cin >> aa >> bb >> cc;
        A a(aa);
        B b(bb);
        C c(cc);
        A* x = &a;
        A* y = &b;
        A* z = &c;
        cout << (x->get_value() + y->get_value() + z->get_value()) << "
";
        cout << ((*x) + y + z)->get_value() << endl;
    }
}
```

```
    return 0;
}
```

- 输入

第一行是数据组数t 每组数据1行, 为三个整数 a 和 b 和 c

- 输出

对每组数据, 输出 a+b+c, 连续输出两次中间空格隔开。(数据保证结果在int范围内) 每组数据输出占一行

- 样例输入

```
3 1 2 3 1 2 4 6 6 6
```

- 样例输出

```
6 6 7 7 18 18
```

```
#include <iostream>
using namespace std;
class A {
private:
    int val;

public:
    A() {}
    A(int n) { val = n; }
    int get_value() { return val; }
    friend A operator+(const A &a, A *b) {
        A temp(a.val + b->val);
        return temp;
    }
    friend A operator+(const A &a, const A &b) {
        A temp(a.val + b.val);
        return temp;
    }
    A *operator->() { return this; } // 很奇怪的一句代码
};

class B : public A {
public:
    B() {}
    B(int n) : A(n) {}
};

class C : public A {
public:
    C() {}
    C(int n) : A(n) {}
};

// 在此处补充你的代码
int main() {
    int t;
    cin >> t;
    while (t--) {
```

```

int aa, bb, cc;
cin >> aa >> bb >> cc;
A a(aa);
B b(bb);
C c(cc);
A *x = &a;
A *y = &b;
A *z = &c;
cout << (x->get_value() + y->get_value() + z->get_value()) << " ";
cout << ((*x) + y + z)->get_value() << endl;
}
return 0;
}

```

- 我是否可以理解为
- 如果类内重载了->操作符，则在遇到原始->操作符不满足代码需要的时候会调用重载之后的操作符，并尝试对这个操作符函数的返回值调用原始的->定义，如果不行则检查是否有对应的重载可以满足该需求，但是最后均会调用一次原始的->来结束

你的理解基本正确，但让我们稍微澄清和细化一下这个过程。特别是，“最后均会调用一次原始的->来结束”这部分需要具体化。实际上，这个过程更多关于对返回类型的 `->` 操作的连续调用，直到达到一个可以直接使用原生 `->` 操作的类型（通常是指针类型），而不是“结束”于一个原生 `->` 操作的调用。

当你重载 `->` 操作符时，编译器会在你尝试使用这个操作符时自动调用你定义的重载版本。如果重载版本返回的是一个对象（或对象的引用），编译器会检查这个对象是否也重载了 `->`。如果是，那么编译器将继续调用这个新返回的对象的 `->` 重载版本。这个过程会持续进行，直到某个 `->` 的调用返回了一个原始指针。在这一点上，编译器会使用这个指针来访问指向对象的成员，使用的是 `->` 的原始（非重载）语义。

关键点：

- **重载 `->` 操作符的类**：可以返回任何类型，但如果返回的是非指针类型的对象，这个对象也应该重载了 `->`，以便编译器可以继续访问成员。
- **终止条件**：这个连续调用链会一直进行，直到遇到一个返回原始指针的 `->` 调用。这时，就会使用原始指针的原生 `->` 功能来访问成员。

示例说明：

假设有一个类 `Wrapper` 重载了 `->` 并返回另一个 `Wrapper` 类的实例，这个过程可以无限进行，但实际上没有实际用途，除非链的某个环节返回一个原始指针，如下所示：

```

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

class Wrapper {
private:
    MyClass* ptr;
public:
    Wrapper(MyClass* p) : ptr(p)    MyClass* operator->() { return ptr; } //
    返回原始指针
}

```

```
};

int main() {
    MyClass obj;
    Wrapper wrap(&obj);
    wrap->foo(); // 这里会调用 Wrapper::operator->() 来获取 MyClass 的指针, 然后
               // 调用 MyClass::foo()
}
```

在这个示例中, `Wrapper` 类重载了 `->` 操作符, 并返回一个指向 `MyClass` 类实例的指针。当我们使用 `wrap->foo()` 时, 实际上首先调用了 `Wrapper` 的 `->` 重载, 这个重载返回 `MyClass` 类的一个指针, 然后通过这个指针调用 `MyClass::foo()` 方法。这个过程中只有一次 `->` 的重载调用, 之后就是直接使用原始指针的原始 `->` 功能。

- C++不允许定义新的运算符
- 所有的运算符重载都对于参数的顺序有严格要求
- 重载后运算符的含义应该符合日常习惯
 - `complex_a + complex_b`
 - `word_a > word_b`
 - `date_b = date_a + n`
- 运算符重载不改变运算符的优先级
- 以下运算符不能被重载:
 - `.` 访问运算符
 - `::` 作用域解析运算符

在 C++ 中, `::` 被称为作用域解析运算符。它不是用来执行传统意义上的运算, 而是用于指定一个标识符 (比如变量名、函数名、类型名等) 的作用域。这个运算符可以用在多种情境中, 包括但不限于:

1. 访问全局变量: 当局部作用域内有与全局变量同名的变量时, 可以使用 `::` 运算符来访问全局变量。

```
int x = 10; // 全局变量

void function() {
    int x = 20; // 局部变量
    cout << ::x; // 打印全局变量x, 结果为10
}
```

2. 指定类成员: 在类外部调用静态成员函数或访问静态成员变量时, 使用类名加 `::`。


```
class MyClass {
public:
    static int myStaticVar;
    static void myStaticFunction() };

int MyClass::myStaticVar = 0; // 定义静态成员变量

void function() {
    MyClass::myStaticFunction(); // 通过作用域解析运算符调用静态成员函数
}
```

3. 在类继承中解决命名冲突：当子类继承的多个父类有同名成员时，可以用作用域解析运算符明确指定使用哪一个父类的成员。

```
class Base1 {
public:
    void function() };

class Base2 {
public:
    void function() };

class Derived : public Base1, public Base2 {
public:
    void callFunction() {
        Base1::function(); // 明确调用Base1的function
    }
};
```

4. 访问命名空间中的成员：当使用命名空间组织代码时，可以用 `::` 来访问命名空间内的成员。

```
namespace MyNamespace {
    void myFunction() }

void function() {
    MyNamespace::myFunction(); // 使用作用域解析运算符访问命名空间中的函数
}
```

作用域解析运算符是 C++ 中用于明确指定作用域的重要工具，特别是在复杂的程序中，它有助于保持代码的清晰和有序。

- `?:` 三目运算符

在 C++ 中，`?:` 是条件运算符，也称为三元运算符。它是唯一的一个三元运算符，格式如下：

```
condition ? expression1 : expression2
```

工作原理如下：

- 首先，评估 `condition`（一个布尔表达式）。
- 如果 `condition` 为 `true`，则表达式的结果是 `expression1` 的值。

- 如果 `condition` 为 `false`，则表达式的结果是 `expression2` 的值。

这个运算符常被用来在单行内对简单条件进行判断，可以看作是 `if-else` 语句的简化形式。

示例

```
int a = 10, b = 20;
int max = (a > b) ? a : b; // 如果 a > b 成立，则 max = a，否则 max = b
```

在这个例子中，如果 `a` 大于 `b`，则 `max` 将被赋值为 `a` 的值，否则 `max` 将被赋值为 `b` 的值。这是一种非常紧凑的方式来写条件赋值语句。

条件运算符非常适合进行简单的条件判断，但如果条件非常复杂或者执行的操作比较多，使用标准的 `if-else` 语句会更清晰易懂。

- `sizeof` 计算内存大小的运算符
- 重载运算符 `()`, `[]`, `->` 或者赋值运算符 `=` 时, 运算符重载函数必须声明为类的成员函数
- 重载运算符是为了让它能作用于对象, 因此重载运算符不允许操作数都不是对象
- 关于 `->` 的重载(一般用于智能指针的实现)

在 C++ 中，`->` 操作符通常用于指针类型，用来访问指针指向的对象的成员。但是，在定义类或结构体时，你可以重载 `->` 操作符来改变它的标准行为。重载 `->` 操作符可以让你的自定义类型模拟指针的行为，这在实现智能指针、迭代器等需要对指针行为进行封装或扩展的场景中非常有用。

-> 操作符的重载方式

重载 `->` 操作符需要遵循一定的规则：

- 它必须是成员函数。
- 它不接受参数。
- 它可以返回任何类型，但返回的类型必须是能够通过 `->` 访问成员的类型（通常是指针或者是定义了 `->` 操作符的类的对象）。

这句话的意思是，在重载 `->` 操作符时，你可以让它返回任意类型的值，但这个返回的类型必须支持 `->` 操作。换句话说，你可以继续通过返回的对象使用 `->` 来访问成员。这通常意味着返回类型要么是一个原始指针（因为原始指针自然支持 `->` 操作），要么是某个类的实例，而这个类同样重载了 `->` 操作符。

示例：自定义智能指针

假设我们有一个自定义的智能指针类，这个类通过重载 `->` 操作符来管理某个对象。为了演示连续使用 `->` 的情况，我们让这个智能指针类返回一个指向自己的指针。这通常不是实际应用中的做法，但可以帮助理解这个概念。

```
#include <iostream>

class Test {
public:
    void method() const { std::cout << "Test method called" <<
        std::endl; }
```

```
};

template <typename T>
class SmartPointer {
private:
    T* ptr;
public:
    SmartPointer(T* p = nullptr) : ptr(p)    ~SmartPointer() { delete
    ptr; }

    // 重载->操作符
    SmartPointer* operator->() {
        std::cout << "SmartPointer -> operator called" << std::endl;
        return this;
    }

    // 提供一个访问ptr的方法，用来最终调用Test的方法
    T* get() const { return ptr; }
};

int main() {
    SmartPointer<Test> smartPtr(new Test());
    smartPtr->get()->method();    // 链式调用

    return 0;
}
```

在这个示例中，`SmartPointer` 类重载了 `->` 操作符，使得当我们使用 `smartPtr->` 时，实际调用的是 `SmartPointer` 类的 `->` 操作符重载函数。这个重载函数打印一条消息，并返回指向 `SmartPointer` 实例自身的指针。然后，我们可以继续使用 `->` 来调用 `SmartPointer` 实例的其他成员函数，比如 `get()`。`get()` 函数返回存储在智能指针中的原始指针，我们可以通过它访问 `Test` 实例的 `method()`。

注意，这个示例主要用于演示目的，展示了重载 `->` 操作符可以让你控制成员访问的过程，并且可以通过返回不同的类型来实现不同的行为。在实际应用中，智能指针类通常会直接返回原始指针指向的对象，以便可以直接通过智能指针访问该对象的成员，而不是像上面示例那样返回自身的指针。

一般使用场景

`->` 操作符通常在实现智能指针、迭代器或类似指针的抽象时重载。智能指针是一种封装了原始指针的类，提供了自动内存管理的功能，例如，`std::unique_ptr`、`std::shared_ptr`。通过重载 `->` 操作符，智能指针类可以使得对它的操作就像是对普通指针的操作一样。

代码示例

下面是一个简单的智能指针类的实现，它重载了 `->` 和 `*` 操作符，用来管理一个动态分配的对象。

```
#include <iostream>

template <typename T>
class SmartPointer {
```

```

private:
    T* ptr; // 原始指针

public:
    // 构造函数
    SmartPointer(T* p = nullptr) : ptr(p) // 析构函数
    ~SmartPointer() {
        delete ptr;
    }

    // 重载->操作符
    T* operator->() const {
        return ptr;
    }

    // 重载*操作符
    T& operator*() const {
        return *ptr;
    }
};

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

int main() {
    SmartPointer<MyClass> p(new MyClass());

    p->show(); // 通过重载的->操作符访问MyClass的成员函数
    (*p).show(); // 通过重载的*操作符访问MyClass的成员函数

    return 0;
}

```

在这个示例中，`SmartPointer` 类重载了 `->` 和 `*` 操作符。通过这种方式，`SmartPointer` 实例在使用时的行为就像是指针一样，可以直接访问其管理的对象的成员。这对于创建能够自动管理资源的类非常有用，因为它简化了内存管理，同时保持了代码的简洁性和易读性。

C++补充知识

命令行参数

命令行界面

- 命令行界面 Command Line Interface, CLI
 - 在图形用户界面得到普及之前使用最为广泛的用户界面
 - 用户可以通过键盘输入指令 计算机收到指令后予以执行
 - Windows 系统中常见的命令行界面：命令提示符 (cmd.exe),当然还有很多同类型的程序，比方说终端等

- 将用户在 CMD 窗口输入可执行文件名的方式启动程序时,跟在可执行文件名后面的那些字符串,称为**命令行参数**

```
:: 使用notepad打开sample.txt
notepad sample.txt

REM 使用copy命令复制file1.txt 到file2.txt
copy file1.txt file2.txt

REM 如果你确实需要在命令提示符中添加说明,你可以使用 & 运算符来分隔命令,并使用 REM 来添加注释

notepad sample.txt & REM 使用notepad打开sample.txt
copy file1.txt file2.txt & REM 使用copy命令复制file1.txt到file2.txt
```

在 CMD 批处理脚本中,通常使用以下方法进行注释:

1. 使用双冒号 `::` 来注释。这是非官方的注释方法,但在大多数情况下有效。一些情况下可能会出现问題,特别是在复杂的批处理脚本中。

```
:: 这是一个注释
```

2. 使用 `REM`。这是官方的注释命令,总是被解释为注释。

```
REM 这是一个注释
```

3. 在 CMD 中,注释必须占据一行的开始,并且 `REM` 命令通常独占一行。`::` 也是如此,它通常用于批处理文件中作为注释,而且通常独占一行。
4. 在命令提示符直接输入命令时,并不常见也不必要添加注释。注释主要是为了批处理文件(`.bat` 或者 `.cmd`)中的可读性和文档目的。在实际的命令行操作中,通常是执行命令而不添加注释的。

```
int main(int argc, char *argv[]) {
    return 0;
}
```

- 参数 `argc` 是启动程序时,命令行参数的个数
 - C/C++语言规定,可执行程序程序本身的文件名,也算一个命令行参数: `argc` 的值至少是 1
- 参数 `argv` 是指针数组,其中每个元素都是一个 `char*` 类型的指针,该指针指向一个字符串,这个字符串里就存放着**命令行参数**
 - `argv[0]` 指向的字符串就是第一个命令行参数,即可执行程序的文件名
 - `argv[1]` 指向第二个命令行参数
 - `argv[2]` 指向第三个命令行参数

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    for (int i = 0; i < argc; i++)
        printf("%s\n", argv[i]);
    return 0;
}
```

`char* argv[]` 和 `char *argv[]` 实际上是完全相同的，它们都表示一个指向字符指针的数组。在C语言中，声明数组时，指针符号(*)可以放在类型名前面或者后面，这并不影响数组的类型。

所以，不管是 `char* argv[]` 还是 `char *argv[]`，它们都表示一个包含字符指针的数组。

命令行重定向

- 重定向输出
 - `sample.exe > out.txt` 将**标准输出**重定向到out.txt 文件中
- 重定向输入
 - `sample.exe < in.txt` 将in.txt 文件中的内容重定向到**标准输入**中

位运算

负整数的表示方式

- 解决方案之一：设置符号位
 - 最左边一位(最高位)作为符号位：表示整数的正负
 - 符号位为0说明该整数是非负的
 - 符号位为1说明该整数是负的
 - 除符号位外的其余位,非负整数 = 绝对值,负整数(**补码**) = 绝对值取反再加 1(取反就是把 0 变成 1 把 1 变成 0)

位运算

- 用途
 - 对某个整数类型变量中的某一位 (bit) 进行操作
 - 判断某一位是否为 1
 - 只改变其中某一位 而保持其他位都不变
- 分类(参考真值表)
 - **&** 按位与:双目操作符,对两个对象的二进制形式的每一位做**且**的操作,返回运算结果
 - 将某变量中的某些位清0或保留某些位不变
 - **|** 按位或:双目操作符,对两个对象的二进制形式的每一位做**或**的操作,返回运算结果
 - 将某变量中的某些位置变为1或保留某些位不变
 - **^** 按位异或:双目操作符,对两个对象的二进制形式的每一位做**异或**的操作,返回运算结果
 - 将某变量中的某些位取反, 且保留其他位不变
 - 只有对应的两个二进制位**不相同**时, 结果的对应二进制位是1,否则为0
 - 异或运算的特点是: 如果 $a \wedge b = c$, 那么就有 $c \wedge b = a$ 以及 $c \wedge a = b$ (穷举法可证),此规律可以用来进行最简单的加密和解密
 - 可以用来交换两个变量
 - ```
a = a ^ b;
b = b ^ a;
a = a ^ b;
```
  - **~** 取反:单目操作符,对对象的二进制形式的每一位做**取反**的操作,返回运算结果

- `<<` 左移:双目操作符, `a<<b` 将a的二进制形式左移b位(左移时, 高位丢弃, 低位补 0)
- `>>` 右移:双目操作符, `a>>b` 将a的二进制形式右移b位(右移时, 移出最右边的位就被丢弃;如果原符号位为1,则右移时高位就补充 1;如果原符号位为0,则右移时高位就补充 0)

## 掩码

掩码 (Mask) 是一个用于指定哪些位是可操作的, 哪些位是不可操作的二进制数。在位运算中, 掩码用于选择性地保留 (set)、清除 (clear)、翻转 (toggle) 或检查 (check) 二进制数中的位。

掩码通常与以下位操作一起使用:

1. **按位与 (AND)** : 用来清除位或者按位复制值。
  - 清除位: 如果掩码在某位为0, 则结果在该位也为0, 不管原数在该位是什么值。
  - 复制位: 如果掩码在某位为1, 原数在该位的值将被复制到结果中。
2. **按位或 (OR)** : 用来设置位。
  - 设置位: 如果掩码在某位为1, 则结果在该位也为1, 不管原数在该位是什么值。
3. **按位异或 (XOR)** : 用来翻转位。
  - 翻转位: 如果掩码在某位为1, 原数在该位的值将被翻转 (0变1, 1变0)。
4. **按位非 (NOT)** : 用来取反所有的位。
  - 取反位: 每位都反转, 通常不需要掩码。

如何使用掩码:

- **设置位**: 要设置 (turn on) 第 `n` 位, 可以创建一个掩码, 其中只有第 `n` 位是1, 其他位都是0, 然后将它与原数进行按位或 (OR) 运算。

```
number |= (1 << n);
```

- **清除位**: 要清除 (turn off) 第 `n` 位, 可以创建一个掩码, 其中只有第 `n` 位是0, 其他位都是1, 然后将它与原数进行按位与 (AND) 运算。

```
number &= ~(1 << n);
```

- **翻转位**: 要翻转 (toggle) 第 `n` 位, 可以创建一个掩码, 其中只有第 `n` 位是1, 其他位都是0, 然后将它与原数进行按位异或 (XOR) 运算。

```
number ^= (1 << n);
```

- **检查位**: 要检查第 `n` 位是0还是1, 可以创建一个掩码, 其中只有第 `n` 位是1, 其他位都是0, 然后将它与原数进行按位与 (AND) 运算。如果结果不为0, 则第 `n` 位是1, 否则是0。

```
bool isSet = number & (1 << n);
```

掩码是二进制操作中非常有用的工具, 能够精确地控制位的状态, 无论是在低级的硬件操作中, 还是在需要位操作的算法中。

(2024/3/10上机课)

位运算是对于整数在位级别上的操作, 主要包括以下几种运算符:

- `&` (按位与)

- `|` (按位或)
- `^` (按位异或)
- `~` (按位取反)
- `<<` (左移)
- `>>` (右移)

要修改一个整型的特定位，可以使用这些位运算符来实现不同的操作：

#### 1. 设置 (Set) 位：

要设置 (置1) 整型中的特定位 (例如，第 `n` 位)，可以使用按位或运算符 (`|`) 结合一个只在第 `n` 位为1的掩码。

```
int setBit(int number, int n) {
 return number | (1 << n);
}
```

#### 2. 清除 (Clear) 位：

要清除 (置0) 整型中的特定位 (例如，第 `n` 位)，可以使用按位与运算符 (`&`) 结合一个只在第 `n` 位为0的掩码。

```
int clearBit(int number, int n) {
 return number & ~(1 << n);
}
```

#### 3. 切换 (Toggle/Flip) 位：

要切换整型中的特定位 (例如，第 `n` 位)，可以使用按位异或运算符 (`^`) 结合一个只在第 `n` 位为1的掩码。

```
int toggleBit(int number, int n) {
 return number ^ (1 << n);
}
```

#### 4. 检查 (Check) 位：

要检查整型中的特定位是否为1 (例如，第 `n` 位)，可以使用按位与运算符 (`&`) 结合一个只在第 `n` 位为1的掩码。

```
bool isBitSet(int number, int n) {
 return (number & (1 << n)) != 0;
}
```

#### 5. 修改 (Modify) 位：

如果你需要将第 `n` 位设置为 `v` (其中 `v` 是0或1)，可以先清除该位，然后根据 `v` 的值设置它。

```
int modifyBit(int number, int n, int v) {
 int mask = 1 << n;
 return (number & ~mask) | (-v & mask);
}
```



在上述函数中， $n$  表示位的索引，通常从0开始计算，即最低位的索引为0。这些运算通常假设整型是无符号的，但除了左移操作外，对有符号整型也是有效的。左移一个有符号整型可能会导致未定义的行为，如果被移出的位包含符号位。

记住，位运算对性能优化有好处，但它们也可能让代码更难以理解和维护。当你使用位运算时，应该确保它们的必要性，并在使用时添加适当的注释。

#### 描述

输入整数  $n$  ( $0 \leq n \leq 2^{30}$ )，以及整数  $i, j$  ( $0 \leq i, j < 31, i < j - 1$ )，输出整数  $k$  (按十六进制输出结果)， $k$  的第  $i$  位和  $n$  相同，第  $j$  位和  $n$  不同， $i, j$  之间的位是1，其他位都是0。这里提到的所有的位，指的都是二进制位，最右边算第0位。

#### 输入

第一行是整数  $t$ ，表示数据组数。

每组输入数据是一行，三个整数  $n, i$  和  $j$ 。

#### 输出

对每组输入数据，按十六进制输出结果。

#### 样例输入

```
2
23 3 5
7 0 2
```

#### 样例输出

```
30
3
```

#### 提示

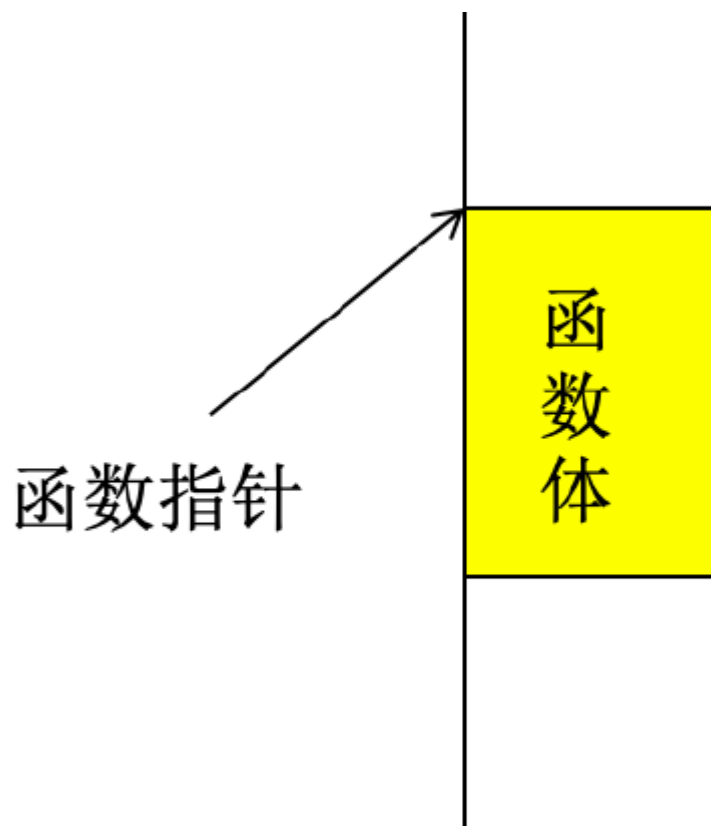
本题请写出完整的程序。

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
 int t = 0;
 cin >> t;
 int n, i, j;
 while (t--) {
 cin >> n >> i >> j;
 int k = 0;
 for (int tr = i + 1; tr < j; tr++) {
 k |= (1 << tr);
 }
 bool flag_i = n & (1 << i);
 bool flag_j = n & (1 << j);
 if (flag_i) {
 k |= (1 << i);
 }
 if (!flag_j) {
```

```
k |= (1 << j);
}
cout<<hex<<k<<endl;
}
return 0;
}
```

## 函数指针

- 在程序运行期间，每个函数都会占用一段连续的内存空间。
- 函数名即为该函数所占内存区域的**起始地址**，也即入口地址。
- 将函数的入口地址赋给一个指针变量，使该指针变量指向该函数。
- 通过指针变量就可以调用这个函数，这种指向函数的指针变量称为**函数指针**。



## 定义和使用

函数指针的定义一般形式如下：

```
返回值类型 (*指针变量名)(参数类型1, 参数类型2, ...);
```

其中：

- 返回值类型表示被指函数的返回值的类型。
- 参数类型1, 参数类型2, ... 分别表示函数的所有参数的类型。

举例：

```
int (*pf)(int, char);
```

上面例子中，`pf` 是一个函数指针，它指向的函数应有以下特征：

- 返回值类型为 `int`。
- 该函数应具有两个参数，第一个参数类型为 `int`，第二个参数类型为 `char`。

函数指针的使用方法如下：

1. 用一个原型匹配的函数的名字给一个函数指针赋值。
2. 要通过函数指针调用它所指向的函数，写法为：`函数指针名(实参表)`。
3. 下面是一个示例

```
#include <stdio.h>

void PrintMin(int a, int b) {
 if (a < b)
 printf("%d\n", a);
 else
 printf("%d\n", b);
}

int main() {
 void (*pf)(int, int);
 int x = 4, y = 5;
 pf = PrintMin;
 pf(x, y);
 return 0;
}
```

```
void qsort(void *base, int nelem, unsigned int width,
 int (* pfCompare)(const void *, const void *));
```

- `base` 是待排序数组的 起始地址
- `nelem` 是待排序数组的 元素个数
- `width` 是待排序数组的 每个元素的大小(以字节为单位)
- `pfCompare` 是一个 函数指针，指向一个**比较函数**

C语言的快速排序函数 `qsort` 需要传入一个**比较函数**的指针，`qsort` 函数的用法规定，比较函数 " 的原型应是：

```
int 函数名(const void * elem1, const void * elem2);
```

对于这个函数，你需要编写一个函数来比较两个元素的顺序。函数应该接受两个参数`elem1`和`elem2`，这两个参数是指向待比较的元素的指针。根据指针指向的实际元素值的大小关系，函数应该返回以下值：

- 1) 如果 `*elem1` 应该排在 `*elem2` 前面，则函数返回一个负整数。
- 2) 如果 `*elem1` 和 `*elem2` 在顺序上不确定，函数返回0。
- 3) 如果 `*elem1` 应该排在 `*elem2` 后面，则函数返回一个正整数。

下面是一个示例C语言函数的结构：

```
int compare_elements(int *elem1, int *elem2) {
 if (*elem1 < *elem2) {
 return -1;
 } else if (*elem1 > *elem2) {
 return 1;
 } else {
 return 0;
 }
}
```

这个函数比较了elem1和elem2指向的元素值的大小关系，并根据上述规则返回相应的值。你可以根据实际情况修改函数的数据类型和比较逻辑。

```
#include <stdio.h>
#include <stdlib.h>

#define NUM 5

int MyCompare(const void *elem1, const void *elem2) {
 unsigned int *p1 = (unsigned int *)elem1; // 对无类型的指针进行强制类型转换, 方便进行解引用
 unsigned int *p2 = (unsigned int *)elem2;
 return (*p1 % 10) - (*p2 % 10);
}

int main() {
 unsigned int an[NUM] = {8, 123, 11, 10, 4};
 qsort(an, NUM, sizeof(unsigned int), MyCompare);

 for (int i = 0; i < NUM; i++) {
 printf("%d ", an[i]);
 }

 return 0;
}
```

## c语言的输入输出语句

在c++中使用的话需要以下条件

```
#include<cstdio>
using namespace std;
```

```
int scanf(const char *,...);
```

- 关于为什么使用 `const char *` 指向字符串: c语言
- [参数可变](#)的函数
  - 第一个参数是**格式化字符串**
  - 后面的参数是**变量的地址**（注意**&**的使用）
- 函数作用：按照第一个参数指定的格式, 将数据读入后面的变量
- 返回值（int）：

- `>0` 成功读入的数据项个数
- `0` 没有项被赋值
  - 输入不匹配: 当 `scanf` 在读取输入时, 如果输入与格式化字符串不匹配, 则会返回0。例如, 如果格式化字符串中指定了要读取整数, 但用户输入了一个字符, 那么 `scanf` 将无法成功匹配, 返回0。
- `EOF` 第一个尝试输入的字符是EOF (结束)

**`int printf(const char *, ...);`**

- 参数可变的函数
  - 第一个参数是**格式字符串**
  - 后面的参数是**待输出的变量**
- 函数作用 :按照第一个参数指定的格式,将后面的变量在屏幕上输出
- `printf`返回值
  - 成功打印的字符数;
  - `<0` 输出出错;

note:对于POJ 上某些题, 返回值为 `EOF` ==>判断输入数据已经全部读完

参数可变的函数示例:

```
// 函数add() 计算可选参数之和
// 参数: 第一个强制参数指定了可选参数的数量, 可选参数为double类型
// 返回值: 和值, double类型
double add(int n, ...)
{
 int i = 0;
 double sum = 0.0;
 va_list argptr;
 va_start(argptr, n); // 初始化argptr
 for (i = 0; i < n; ++i) // 对每个可选参数, 读取类型为double的参数,
 sum += va_arg(argptr, double); // 然后累加到sum中
 va_end(argptr);
 return sum;
}
```

格式化字符串里的格式控制符号

- `%d`: 用于读入或输出 `int` 类型的整数。
- `%c`: 用于读入或输出单个字符 `char` 类型。
- `%f`: 用于读入或输出浮点数 `float` 类型。
- `%s`: 用于读入或输出字符串, 对应 `char *` 类型。
- `%lf`: 用于读入或输出双精度浮点数 `double` 类型。
- `%e`: 用于以科学计数法格式输出浮点数。
- `%x`: 用于以十六进制形式读入或输出 `int` 类型的整数。
- `%u`: 用于读入或输出无符号整数 `unsigned int` 类型。
- `%I64d`: 用于在Windows平台下读入或输出64位整数 `_int64` 类型。

- `%lld`: 用于在Linux平台下读入或输出长整型 `long long` 类型。
- `%p`: 用于输出指针的地址值。
- `%.5lf`: 用于输出 `double` 类型的浮点数, 精确到小数点后五位。

这些格式化占位符是C语言标准库函数 `printf` 和 `scanf` 及其相关函数的一部分。在使用这些占位符时, 您需要确保提供的变量类型与占位符匹配, 否则可能导致未定义行为。

`scanf()` 函数格式中, 如有非控制符也非空格字符, 且输入数据中相应位置也出现该字符, 则该字符会被跳过

### `char* gets(char * s)`

- 读取一行
- 从标准输入读取一行到字符串 `s`
  - 如果成功, 返回值就是 `s` 地址
  - 如果失败, 返回值是 `NULL`
  - 可以根据返回值是 `NULL` 判定输入数据已经读完

Note: 调用时要确保 `s` 指向的缓冲区足够大, 否则可能发生内存访问错误

- `int sscanf(const char * buffer, const char * format, ...)` 从字符串中读取数据
- `int sprintf(const char * buffer, const char * format, ...)` 向字符串中输出数据

## C到C++

### 动态内存分配

在C++中, `new` 运算符用于在程序运行时动态分配内存。相应地, `delete` 运算符用于释放那些通过 `new` 运算符分配的内存。这使得程序能够根据需要分配和释放恰当数量的内存。

分配单个对象的语法如下:

```
T* p = new T; // T是数据类型, p是指向T类型的指针
```

这段代码为类型 `T` 的对象分配内存, 并将内存的地址赋给指针 `p`。例如, 为 `int` 类型分配内存的代码会是这样:

```
int* pn = new int; // 动态分配一个int大小的内存空间
```

指针 `pn` 指向一块可以存储 `int` 值的内存。可以通过 `*pn` 来访问并修改这块内存。

初始化动态分配的对象(调用构造函数):

```
int* pn = new int(5); // 分配内存并初始化为5
```

动态分配数组:

```
T* p = new T[N]; // T是数据类型, p是指向T的指针, N是数组大小
```

这段代码为类型 `T` 的 `N` 个元素分配内存, 并将数组的起始地址赋值给 `p`。数组访问越界是个常见错误, 应当避免。

示例：

```
int* pn = new int[100]; // 分配100个int大小的数组
pn[0] = 10; // 正确
pn[100] = 30; // 一般可以过编译,但是错误,越界
```

程序从操作系统动态分配所得的内存空间,使用完后应该释放,交还操作系统,以便操作系统将这片内存空间分配给其他程序使用,否则即便程序运行结束,这部分内存空间仍然有可能不会被操作系统收回(取决于操作系统如何设计),从而成为白白浪费掉的内存垃圾,这种现象也称为内存泄漏

释放内存的操作：

```
delete pn; // 释放单个对象的内存,调用pn指向的动态分配对象的析构函数
```

在C++中,不能连续对同一个指针进行多次delete操作的原因是由于堆内存管理的机制。

当使用new运算符分配内存时,会在堆内存中创建一个对象,并返回该对象的地址。当我们使用delete运算符释放这块内存时,会将内存返回给系统,并且指针变为无效。如果之后再次使用delete来释放同一个指针指向的内存,由于该内存已经被释放并且指针已经无效,这样的操作就会导致未定义行为。

连续对同一个指针进行多次delete操作可能会导致以下问题：

1. 可能会破坏堆内存管理的数据结构,导致内存泄漏或程序崩溃。
2. 重复释放已经释放的内存可能会导致程序出现未定义行为,从而产生难以预测的结果。

因此,为了避免这种情况的发生,应该注意只对已经分配的内存进行一次delete操作,并且在释放内存后将指针设置为nullptr,以避免误操作导致对已经释放的内存进行操作。

释放数组：

```
delete[] pArray; // 释放数组内存
```

如果要求分配的空间太大,操作系统找不到足够的内存来满足,那么动态内存分配就会失败。保险做法是在进行较大的动态内存分配时,要判断一下分配是否成功

判断的方法是：如果new表达式返回值是NULL,则分配失败;否则分配成功

```
#include <iostream>

int main() {
 int* pn = new int[10000]; // Allocating memory for an array of 10 integers

 if (pn == nullptr) {
 std::cout << "内存分配失败" << std::endl; // Memory allocation failed
 } else {
 std::cout << "内存分配成功" << std::endl; // Memory allocation successful
 delete[] pn; // Don't forget to free the allocated memory
 }
 return 0;
}
```

正确地匹配new和delete以及new[]和delete[]是非常重要的,以防止内存泄漏和其他内存相关问题。

# 引用

下面的写法定义了一个引用，并将其初始化为引用某个变量。

类型名 & 引用名 = 某变量名;

```
int n = 4;
int &r = n; // r引用了n, r的类型是int &
```

- 某个变量的引用,等价于这个变量,相当于该变量的一个别名。

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

int main() {
 int n = 4;
 int &r = n;

 cout << "初始值: " << n << endl; // 输出初始值

 r = 4; // 通过引用 r 修改变量 n 的值为 4
 cout << "修改后的值: " << r << endl; // 输出修改后的值

 n = 5; // 直接修改变量 n 的值为 5, 由于 r 和 n 指向同一内存位置, r 的值也会被修改

 cout << "最终值: " << r << endl; // 输出最终值

 return 0;
}
```

- 定义引用时一定要将其初始化成引用某个变量。
- 初始化后，它就一直引用该变量(相当于绑定了)，不会再引用别的变量了。
- 引用只能引用变量(或者变量的引用)，不能引用常量和表达式。
- 使用引用可以用于交换函数的参数传递

```
void swap(int &a, int &b) {
 int tmp;
 tmp = a;
 a = b;
 b = tmp;
}
```

- 引用可以作为函数的返回值



```
int n = 4;

int& SetValue() {
 return n;
}

int main() {
 SetValue() = 40; // 修改引用返回的值, 但实际上是修改了 n 的值
 cout << n << endl; // 输出修改后的 n 的值
 return 0;
}
```

## 常引用

定义引用时, 前面加 `const` 关键字, 即为常引用

```
int n;
const int & r = n; // r的类型是 const int &
```

- 不能通过常引用去修改其引用的内容(只能通过修改常引用引用的值自身来使得常引用的值发生改变)

```
int n = 100;
const int & r = n;
r = 200; // 编译错
n = 300; // 没问题
```

在 C++ 中, 常引用 (`const T &`) 和非常引用 (`T &`) 是不同的类型, 它们之间不能直接进行初始化或赋值操作。常引用是指向不可修改的对象, 而非常引用是指向可修改的对象。

非常引用 (`T &`) 可以用来初始化常引用 (`const T &`), 因为常引用可以指向可修改的对象, 但反过来是不行的。

```
int value = 10;
const int constValue = 20;

int& ref = value; // 正确: 非常引用绑定到可修改的对象
const int& constRef = ref; // 正确: 常引用绑定到非常引用, 不会修改对象

const int& constRef2 = constValue; // 正确: 常引用绑定到常量对象
int& ref2 = constRef2; // 错误: 非常引用不能绑定到常引用
```

## const 关键字的用法

- 定义常量

```
const int MAX_VAL = 23
const string SCHOOL_NAME = "Peking University";
```

- 定义常量指针

- 不可通过常量指针修改其指向的内容
- 不能把常量指针赋值给非常量指针,反过来可以
- 函数参数为常量指针时,可避免函数内部不小心改变参数指针所指地方的内容
- **不能把常量指针赋值给非常量指针,反过来可以**

不能把常量指针赋值给非常量指针,反过来可以

这句话的原理是否和刚才的情况相似?

即

非常量指针可能在这样的赋值之后通过常量指针作为中介修改常量指针指向的位置。

你的理解完全正确。不能把常量指针赋值给非常量指针,而反过来是可以的,这个原理与常引用和非常引用的情况类似。

常量指针(`const T*`)是指向常量的指针,它不允许通过指针来修改指向的对象,但指针本身可以被修改。

非常量指针(`T*`)是指向可修改对象的指针,它允许通过指针来修改指向的对象,同时指针本身也可以被修改。

将常量指针赋值给非常量指针是不允许的,因为这样做会去除常量性,允许通过非常量指针来修改原本应该是常量的对象。这违反了常量指针的约束,可能导致意料之外的行为。

示例:

```
int value = 10;
const int constValue = 20;

const int* constPtr = &constValue; // 常量指针指向常量
int* ptr = constPtr; // 错误: 不能将常量指针赋值给非常量指针

int* ptr2 = &value; // 非常量指针指向可修改的对象
const int* constPtr2 = ptr2; // 正确: 可以将非常量指针赋值给常量指针
```

在上面的示例中,我们可以看到:

- 常量指针 `constPtr` 指向常量 `constValue`, 不能通过 `constPtr` 来修改 `constValue`。
- 将常量指针 `constPtr` 赋值给非常量指针 `ptr` 是错误的, 因为这样会去除常量性。
- 非常量指针 `ptr2` 指向可修改的对象 `value`, 可以通过 `ptr2` 来修改 `value`。
- 将非常量指针 `ptr2` 赋值给常量指针 `constPtr2` 是正确的, 因为常量指针可以指向可修改的对象, 但不能通过 `constPtr2` 来修改对象。

总之, 不能将常量指针赋值给非常量指针, 因为这样做会去除常量性, 允许通过非常量指针来修改原本应该是常量的对象。这与常引用和非常引用的情况类似。将非常量指针赋值给常量指针是可以的, 因为常量指针可以指向可修改的对象, 但不能通过常量指针来修改对象。

```
int n, m;
const int * p = &n;
* p = 5; // 编译出错
n = 4; // ok
p = &m; // //ok, 常量指针的指向可以变化

const int * p1;
int * p2;
```

```
p1 = p2; //ok
p2 = p1; //error
p2 = (int *) p1; //ok, 强制类型转换

void MyPrintf (const char * p){
strcpy(p, "this "); // 编译出错
printf("%s", p); //ok
}
```

- 定义常引用
  - 不能通过常引用修改其引用的变量

```
int n;
const int & r = n;
r = 5; // error
n = 4; // ok
```

## 内联函数

- 函数调用是有时间开销的。如果函数本身只有几条语句，执行非常快，而且函数被反复执行很多次，相比之下调用函数所产生的这个开销就会显得比较大。
- 为了减少函数调用的开销，引入了**内联函数**机制。编译器处理对内联函数的调用语句时，是将整个函数的代码插入到调用语句处，而不会产生调用函数的语句。

```
inline int Max(int a,int b){
if(a > b) return a;
return b;
}
```

### 内联成员函数

- 在成员函数前面加上**inline**关键字之后,成员函数就会成为内联成员函数
- 将整个函数写在类定义的内部,函数也会成为内联成员函数

## 函数重载

- 一个或多个函数，名字相同，然而**参数个数**或**参数类型**不相同，这叫做**函数的重载**。
- 函数重载使得函数命名变得简单。
- 编译器根据调用语句的中的实参的个数和类型判断应该调用哪个函数。
- 以下三个函数是重载关系：

```
int Max (double f1,double f2) {...}
int Max int n1,int n2) {...}
int Max int n1,int n2,int n3) {...}
```

在函数重载中，函数的返回值类型通常不会被用来区分重载的函数。函数重载仅仅通过函数名称和参数列表来区分不同的函数。因此，即使函数的返回值类型不同，只要函数名称相同,参数列表(参数个数和类型)不同，这些函数就会被认为是重载的。

```
int func(int a);
float func(double a); // 参数列表不同，可以重载
void func(int a, double b); // 参数列表不同，可以重载
double func(int a, int b, int c); // 参数列表不同，可以重载
```

## 函数缺省参数

在C++中，可以为函数的参数提供默认值，这样在调用函数时，如果不提供相应的参数，就会使用这些默认值。但是有一些规则需要遵守：

1. 只有最右边的连续若干个参数可以有默认值。这意味着一旦某个参数有了默认值，其右侧的所有参数也必须有默认值。
2. 调用函数时，不能跳过中间的参数只提供最右边的参数。

基于您提供的 `func` 函数定义：

```
void func(int x1, int x2 = 2, int x3 = 3) {
 // ...
}
```

下面是这个函数的几种调用方式：

```
func(10); // 等效于 func(10, 2, 3)
func(10, 8); // 等效于 func(10, 8, 3)
// func(10, , 8); // 这是错误的，不能跳过参数
```

对于最后一种情况，如果您想调用 `func` 并只指定 `x1` 和 `x3`，不能使用 `func(10, , 8)` 的形式，因为这违反了语法规则。如果需要指定 `x3` 但使用 `x2` 的默认值，您必须显式传递 `x2` 的默认值：

```
func(10, 2, 8); // 显式传递x2的默认值
```

这样，即使您想使用 `x2` 的默认值，也需要在调用时指定它。

- 函数参数可缺省的目的在于提高程序的可扩充性。
- 即如果某个写好的函数要添加新的参数，而原先那些调用该函数的语句，未必需要使用新增的参数，那么为了避免对原先那些函数调用语句的修改，就可以使用缺省参数。下面给出了解释
  - 这句话的意思是，如果你已经有一个在多处调用的函数，然后你想要为这个函数添加一个新的参数，但不想修改所有已经存在的调用代码，你可以给这个新参数一个默认值。这样，原有的代码在调用这个函数时就不需要改动，因为它们会自动使用这个新参数的默认值。而新的代码可以使用这个新参数来调用函数，如果需要的话。

下面是一个具体的例子：

假设原先有一个函数：

```
// 原先的函数
void displayMessage(std::string message) {
 std::cout << message << std::endl;
}

// 在代码的多个地方调用这个函数
displayMessage("Hello, World!");
displayMessage("Welcome to C++ programming!");
```

现在，你想要添加一个新的参数 `bool addNewLine`，以便有选择地添加换行符。如果直接添加这个参数，你将不得不修改所有调用这个函数的地方。为了避免这种情况，你可以给新参数一个缺省值：

```
// 修改后的函数，为新参数提供了缺省值false
void displayMessage(std::string message, bool addNewline = false) {
 if (addNewline) {
 std::cout << message << std::endl; // 添加换行符
 } else {
 std::cout << message; // 不添加换行符
 }
}

// 现有的调用不需要修改，会使用缺省值false
displayMessage("Hello, World!");
displayMessage("Welcome to C++ programming!");

// 新的调用可以使用新参数
displayMessage("Hello, New World!", true);
displayMessage("No newline here", false);
```

在这个例子中，即使 `displayMessage` 函数被修改了，所有原有的函数调用都不需要改动，因为它们会自动应用 `addNewline` 参数的默认值 `false`。新的代码可以利用这个新的功能，根据需要传递 `true` 或 `false`。

## 面向对象编程(类)

### 类的介绍

#### 关于内存分配

结构体和类的内存空间大小取决于其成员变量的大小之和。对于一个类，`sizeof()` 可以用来计算其所占用的内存空间大小。每个对象都有自己的存储空间，因此一个对象的成员变量的改变不会影响到另一个对象。

#### 关于对象间的运算

和结构变量一样，对象之间可以使用 `=` 进行赋值操作，但通常情况下不能直接使用 `==` 等比较运算符进行比较，除非相应的运算符被重载。

#### 关于成员变量的访问和成员函数的调用

- 对于对象自身或者对象的引用,使用 `对象名/引用名.成员名` 进行访问
- 对于对象的指针,使用 `指针->成员名` 进行访问

# 访问权限

在C++中，类对象的访问权限由类的成员访问修饰符（`public`、`private`、`protected`）来控制。这些访问修饰符决定了类的成员（包括成员变量和成员函数）对外部代码的可见性和可访问性。

## 1. 公有成员（`public`）：

- 公有成员在类的外部 and 内部均可以访问，没有访问限制。
- 外部代码可以通过类对象直接访问公有成员，也可以通过类的公有成员函数来访问。
- 公有成员常用于定义类的接口，提供对外部代码的访问点。

## 2. 私有成员（`private`）：

- 私有成员只能在所属类的成员函数中直接访问，类外部无法直接访问(这种机制叫做**隐藏**)。
- 便于修改成员变量的类型等属性 ==> 只需要修改成员函数
- 外部代码无法直接访问私有成员，需要通过类的公有成员函数提供的接口来间接访问。
- 私有成员通常用于隐藏类的实现细节，确保数据的**封装**和安全性。

## 3. 受保护成员（`protected`）：

- 受保护成员可以在所属类及其派生类中访问，类外部无法直接访问。
- 外部代码无法直接访问受保护成员，但派生类可以访问这些成员。
- 受保护成员通常用于在类的继承层次结构中共享数据或行为。

访问权限控制是面向对象编程中的重要概念，可以帮助实现信息隐藏、封装和继承等特性，提高代码的安全性和可维护性。通过合理地设计类的成员访问权限，可以有效管理类的接口和内部实现，确保类的正确使用并降低外部代码对类内部的依赖。上述关键字在出现次数和先后次序上都有限制。

如果某个成员前面没有上述关键字,则被缺省地认为是私有成员

用Struct定义类的唯一区别:默认访问权限是公有

# 构造函数(Constructor)

## 通用构造函数

**通用构造函数**是一种特殊的成员函数，它在类的对象被创建时自动调用，用以初始化对象。

### 1. 自动初始化行为：

- **全局变量**：未经初始化的全局基本类型变量会被自动设置为0。
- **局部变量**：未经初始化的局部基本类型变量会有一个随机的初始值。

### 2. 默认构造函数：

- 如果一个类中没有显式地定义任何构造函数，编译器会自动生成一个默认的非参数构造函数，它不会执行任何操作。
- 如果类中定义了至少一个构造函数，则编译器不会自动生成默认的非参数构造函数。

### 3. 构造函数特性：

- 构造函数的名称与类名相同。
- 构造函数不能有返回类型，即使是 `void` 也不可以。
- 构造函数可以有参数，支持重载（即一个类可以有多个构造函数，只要它们的参数列表不同）。

### 4. 初始化作用：

- 构造函数的主要作用是初始化对象，比如为成员变量赋予初始值。
- 使用构造函数可以避免编写单独的初始化函数，并确保在使用对象之前它已被适当初始化。

#### 5. 构造函数的访问性:

- 构造函数通常应该是公有的（`public`），以便可以从类外部创建对象。
- 私有构造函数（`private`）会限制对象的创建，例如，它们常用于实现单例模式。

#### 6. 对象的初始化:

- 对象一旦通过构造函数被创建，并完成初始化后，就不能再通过调用任何构造函数来重新初始化。
- 正确编写构造函数是一个好习惯，因为它确保了对象在创建时总会拥有合理的初始值。

#### 7. 构造函数的重载:

- 可以通过函数重载设计不同参数列表的构造函数

```
#include <iostream>

class CSample {
 int x;
public:
 CSample() {
 std::cout << "Constructor1 called" << std::endl;
 }
 CSample(int n) {
 x = n;
 std::cout << "Constructor2 called" << std::endl;
 }

 ~CSample() {
 std::cout << "Destructor called" << std::endl;
 }
};

int main() {
 CSample array1[2];
 std::cout << "step1" << std::endl;

 CSample array2[2] = {4, 5};
 std::cout << "step2" << std::endl;
 // C++允许使用初始化列表来初始化类对象数组。在这种情况下，编译器会根据列表中的值来调用合适的构造函数来初始化数组元素。

 CSample array3[2] = {2};
 std::cout << "step3" << std::endl;
 // 有参数则传递, 没有则调用无参数构造函数

 CSample* array4 = new CSample[2];
 delete[] array4;

 return 0;
}
```

```
#include <iostream>
```

```

class Test {
public:
 Test(int n) {
 std::cout << 1 << std::endl;
 }

 Test(int n, int m) {
 std::cout << 2 << std::endl;
 }

 Test() {
 std::cout << 3 << std::endl;
 }
};

int main() {
 Test array1[3] = {1, Test(1, 2)};

 Test array2[3] = {1, (1, 2)};

 return 0;
}

```

逗号操作符表达式（Comma Operator Expression）是C++中的一种操作符，表示为逗号`,`。逗号操作符可以用来连接多个表达式，并返回最后一个表达式的值。在逗号操作符表达式中，逗号左边的表达式会被执行，但其结果会被忽略；而逗号右边的表达式会被计算，并返回其值作为整个逗号操作符表达式的值。

逗号操作符主要在以下两种情况下使用：

1. 在 `for` 循环中的迭代部分，可以使用逗号操作符连接多个表达式，例如：

```

for (int i = 0, j = 10; i < 5; i++, j--) {
 // do something
}

```

2. 在某些复杂表达式中，可以使用逗号操作符连接多个表达式，例如：

```

int x = (a = 5, b = 10, a + b);

```

需要注意的是，逗号操作符的优先级较低，因此在复杂表达式中使用时，可能需要加上括号来明确运算顺序。此外，由于逗号操作符会使代码变得难以理解，因此在一般情况下建议尽量避免过度使用逗号操作符表达式，以提高代码的可读性和可维护性。

## 复制构造函数(拷贝构造函数)

- 只有一个参数,即对同类对象的引用

- 类名::类名(类名&){}
- 类名::类名(const 类名&){} // 能以常量对象作为参数

- 如果没有定义复制构造函数,那么编译器生成默认复制构造函数
- 如果定义的自己的复制构造函数,则默认的复制构造函数不存在



- 不允许有形如 `x::x(x)` 的构造函数

## 类型转换构造函数

- 类型转换构造函数是C++中一个特殊类型的构造函数，它允许对象从一种类型隐式地转换成另一种类型。这类构造函数符合以下特征：
  - 它们只有一个参数，这使得编译器可以使用单个值来构造新对象。
  - 它们不是复制构造函数，复制构造函数通常有一个相同类类型的参数（通常是引用）。

类型转换构造函数允许在需要特定类型对象时，从提供的单个值隐式构造该类型的对象。这种转换通常是隐式的，除非构造函数前被明确地标记为 `explicit`。

### 示例

假设你有一个 `MyString` 类，它可以从 `const char*` 类型隐式转换：

```
class MyString {
public:
 // 类型转换构造函数
 MyString(const char* s) {
 // 在这里分配内存并复制字符串
 }
 // ...
};

void printString(const MyString& str) {
 // 输出字符串
}

int main() {
 // 虽然该函数的参数是MyString类型的变量,但是由于类型转换构造函数的存在,系统会通过已给的字符串作为参数自动创建一个MyString的临时对象
 printString("This is a string");

 return 0;
}
```

在上面的例子中，当 `printString` 被调用并传入一个 `const char*` 类型的字符串字面量时，`MyString` 的类型转换构造函数会被隐式调用，构造一个 `MyString` 类型的临时对象。

### 防止隐式转换

`explicit` 关键字用于防止构造函数定义的隐式类型转换。如果你不希望在上述 `MyString` 类中的构造函数进行隐式转换，你可以这样声明：

```
class MyString {
public:
 // 显式类型转换构造函数
 explicit MyString(const char* s) {
 // 在这里分配内存并复制字符串
 }
 // ...
};
```

```

void printString(const MyString& str) {
 // 输出字符串
}

int main() {
 // 由于构造函数是 explicit 的, 下面的代码将不合法:
 // printString("This is a string"); // 错误!

 // 必须显式地构造 MyString 对象
 printString(MyString("This is a string"));

 return 0;
}

```

使用 `explicit` 关键字后, 上述代码中的隐式转换将不再被允许。这样做可以提供更多的类型安全, 并防止可能导致错误的隐式转换。

## 析构函数(Destructor)

析构函数是一个特殊的成员函数, 它的名字与类名相同, 在前面加上一个波浪号 (`~`)。它没有返回值, 也不接受参数。

- **数量:** 一个类最多只能有一个析构函数。
- **自动调用:** 析构函数在对象消亡时自动被调用。
- **作用:** 定义析构函数的目的是为了让对象在消亡前做一些善后工作, 例如释放分配的空间。
- **缺省析构函数:** 如果定义类时没有写析构函数, 则编译器会自动生成一个缺省 (默认) 析构函数。
- **缺省析构函数的行为:** 缺省析构函数什么也不做。
- **自定义析构函数:** 如果定义了析构函数, 则编译器不会生成缺省析构函数。
- **析构函数和对象数组:** 当对象数组的生命周期结束时, 对象数组的每个元素的析构函数都会被调用。例如, 如果使用 `new` 关键字创建了一个对象数组, 那么在使用 `delete[]` 释放时, 会调用每个对象的析构函数。

```

Ctest* pTest = new Ctest[3]; // 分配一个对象数组
delete[] pTest; // 调用每个对象的析构函数并释放数组

```

如果使用 `delete` 而不是 `delete[]`, 那么只会调用数组第一个对象的析构函数。

- **delete 运算符:** `delete` 运算符会导致析构函数的调用。

```

Ctest* pTest = new Ctest; // 分配一个单一对象
delete pTest; // 调用对象的析构函数并释放对象

```

- **析构函数在特定场景下的调用:** 当一个对象作为函数的返回值返回时, 析构函数会在复制返回值之后被调用。

下面是一个示例代码, 其中包含一个简单的类 `CMyclass`, 它有一个析构函数, 该析构函数在对象生命周期结束时输出一条消息:

```

#include <iostream>
using namespace std;

```

```

class CMyclass {
public:
 CMyclass() {
 // Constructor
 }
 ~CMyclass() {
 cout << "Destructor called" << endl;
 }
};

CMyclass fun(CMyclass sobj) {
 return sobj;
}

int main() {
 CMyclass obj;
 obj = fun(obj);
 return 0;
}

```

在上面的代码中，当 `fun` 函数被调用时，参数 `sobj` 是通过拷贝构造函数创建的。当 `fun` 返回时，返回值也是通过拷贝构造函数创建的。每当一个临时对象超出其作用域，它的析构函数将被调用。因此，在 `main` 函数中，你会看到析构函数被调用多次：一次是函数返回时创建的临时对象，一次是 `main` 函数结束时 `obj` 对象的析构函数。

## 各种构造函数,析构函数的调用时机

在C++中，构造函数和析构函数是类的特殊成员函数，它们在对象的生命周期中的特定时刻被调用。以下是它们各自的调用时机的详细总结：

### 通用构造函数（普通构造函数）

通用构造函数是指除复制构造函数和类型转换构造函数以外的所有构造函数。它们可以有多个参数，也可以没有参数（默认构造函数）。

调用时机：

- 当创建一个类的新对象时，如果没有提供初始值，则调用默认构造函数（无参数的构造函数）。
  - 如果只提供了含有参数的构造函数,则初始化对象必须调用构造函数

```

class A {
 int v;

public:
 A(int n) { v = n; }
};
// 此时编译器不再给出默认无参构造函数,如果想要使用无参数的方式构造对象,则需要显式的将
// 无参构造函数定义出来
int main() {
 A a2(1);
 return 0;
}
A a1, a2; a1 = a2;

```

- 当创建对象并提供一个或多个初始化参数时，调用相应的构造函数。
- 当通过**直接初始化**（使用圆括号）或**列表初始化**（使用花括号）方式创建对象时。
- 如果一个类成员是对象，并且在其类的构造函数中没有显式初始化，那么该成员对象的默认构造函数会被调用。

## 复制构造函数

复制构造函数用于创建一个类的新对象作为现有对象的副本。

调用时机：

- 当一个对象以值传递的方式传递给函数时。
- 当一个对象以值传递的方式从函数返回时。
- 当使用另一个同类型的对象**初始化新对象**时（例如 `Type obj1 = obj2;` 或 `Type obj1(obj2);`）。
- 当编译器生成临时对象时，比如在表达式中转换或调整对象类型。（这种情况通常发生在将一个子类对象赋值给基类对象或引用时。）

在C++中，参数可以通过值传递、指针传递或引用传递。当使用引用（尤其是常量引用）传递参数时，可以在不牺牲性能的情况下增加代码的效率和安全性。以下是使用引用和常量引用的具体实现及其优化效果的解释：

### 使用引用传递参数

当你使用引用传递参数时，实际上传递的是参数的别名，而不是它的副本。这意味着函数体内对参数所做的更改将影响原始数据。

具体实现：

```
void increment(int& value) {
 value += 1;
}

int main() {
 int i = 0;
 increment(i);
 // i 的值现在是1，因为它通过引用传递给 increment 函数
}
```

### 使用常量引用传递参数

常量引用是一种特殊类型的引用，它不允许通过引用修改数据。这在你需要保证传入的参数在函数内部不会被改变时非常有用，同时又想避免复制成本，尤其是对于大型对象。

具体实现：

```
void print(const std::string& str) {
 std::cout << str;
 // str 不能被修改，因为它通过常量引用传递的
}

int main() {
 std::string hello = "Hello, World!";
 print(hello);
 // hello 可以安全地传递给 print 函数，且不会被修改
}
```

### 优化效果

1. **避免不必要的复制**：当通过值传递大型对象时，会发生复制，这可能涉及分配内存和复制数据，这可能是一个昂贵的操作。通过引用传递避免了这种复制，因此可以提高性能。
2. **节省内存**：引用传递不需要额外的内存来存储参数的副本，这对于内存使用是一个优化。
3. **常量引用保护数据**：当使用常量引用时，你可以保护原始数据不被修改，这提供了更好的安全性，尤其是在使用库或大型代码基时，这样的保护尤为重要。
4. **函数接口清晰**：使用常量引用参数还可以使函数的意图更加明确，即该函数不会修改传入的参数。
5. **支持临时对象**：常量引用还可以接受临时对象作为参数，而非常量引用则不行。
6. **提高代码可读性**：通过明确区分哪些参数可以被修改（引用）和哪些不应被修改（常量引用），代码的可读性和可维护性得到了提升。

总的来说，使用引用和常量引用传递参数能够提高程序的性能和可维护性，尤其是在处理大型对象时。然而，使用引用时还应该注意确保引用的生命周期和作用域，避免悬垂引用等问题。

## 类型转换构造函数

类型转换构造函数是一个只有一个参数的构造函数，或者所有参数都有默认值，它允许隐式转换或显式转换成类类型。

### 调用时机：

- 当将一个对象初始化为不同类型时，如果存在一个可以接受该类型参数的构造函数，则会调用该构造函数。
- 在需要将一个类型隐式转换为类类型以满足表达式或函数调用的语境时。
- 使用 `static_cast` 进行显式类型转换时。

## 析构函数

析构函数用于在对象生命周期结束时进行清理工作，如释放资源、内存等。

### 调用时机：

- 当对象的生命周期结束时（例如，一个局部对象的函数返回时或包含它的作用域结束时）。
- 当通过 `delete` 操作符删除一个动态分配的对象时。
- 当一个临时对象的生命期结束时（通常是在创建它的全表达式结束之后）。
- 对于一个对象数组，当通过 `delete[]` 操作符删除数组时，每个元素的析构函数都会被调用。
- 当一个包含对象成员类对象被销毁时，它的成员对象的析构函数也会被调用。

通常，析构函数是不可重载的，每个类只能有一个析构函数。编译器会在适当的时间自动调用析构函数，不需要手动调用。

理解这些构造函数和析构函数的调用时机对于编写效率高、资源管理恰当的C++程序至关重要。它们的自动调用特性有助于避免资源泄露和其他常见的编程错误。

## 赋值运算符函数

赋值运算符函数(assignment operator function)是一个特殊的成员函数,用于定义当使用赋值运算符`=`对类的对象进行赋值时的行为。它在类中的作用是实现对象之间的赋值操作,将一个对象的值赋给另一个对象。

以下是关于赋值运算符函数的详细介绍:

### 1. 作用:

- 赋值运算符函数定义了当使用赋值运算符`=`对类的对象进行赋值时的行为。
- 它允许你自定义赋值操作的逻辑,例如深拷贝、资源管理等。
- 如果没有显式定义赋值运算符函数,编译器会提供一个默认的赋值运算符函数,该函数会逐个赋值类的每个成员变量。

### 2. 调用时机:

- 当使用赋值运算符`=` 将一个对象赋值给另一个对象时,会调用赋值运算符函数。
- 例如, `a1 = a2;` 会调用 `a1` 的赋值运算符函数,将 `a2` 的值赋给 `a1`。
- 赋值运算符函数通常在已经创建了对象之后调用,用于修改对象的值。

### 3. 语法:

- 赋值运算符函数的函数名为 `operator=`。
- 它接受一个常量引用参数,表示要赋值的源对象。
- 它返回一个对当前对象的引用,以支持连续赋值。(意味着它有一个隐式的参数(在这里是`this`指针)用于指向等号左边的元素,即被赋值对象)
- 一般形式为: `Type& operator=(const Type& other);`
- 支持函数重载

### 4. 和复制构造函数的区别:

- 复制构造函数用于在创建对象时,使用另一个同类型的对象来初始化新对象。
- 赋值运算符函数用于在对象已经创建后,将一个对象的值赋给另一个对象。
- 复制构造函数在对象创建时调用,而赋值运算符函数在对象创建后的赋值操作时调用。
- 复制构造函数有一个常量引用参数,而赋值运算符函数通常有一个常量引用参数和一个 `this` 指针。

下面是一个示例,展示了赋值运算符函数的定义和使用:

```
class MyClass {
private:
 int value;

public:
 MyClass(int val) : value(val) {}

 // 赋值运算符函数重载 1: 接受 MyClass 对象
```

```

MyClass& operator=(const MyClass& other) {
 if (this != &other) {
 value = other.value;
 }
 return *this;
}

// 赋值运算符函数重载 2: 接受整数值
MyClass& operator=(int val) {
 value = val;
 return *this;
}
};

int main() {
 MyClass obj1(10);
 MyClass obj2(20);

 obj1 = obj2; // 调用重载 1, 接受 MyClass 对象
 obj1 = 30; // 调用重载 2, 接受整数值

 return 0;
}

```

## 成员函数重载和参数默认

- 任何有定义的表达式都可以成为函数参数的默认值
  - 包括运算表达式,函数等等
- 成员函数也可以重载
- 成员函数/构造函数也可以带默认值参数
- 使用缺省函数要注意避免有函数重载时的二义性
  - 例如函数既有默认参数,又有无参数的函数重载,此时调用函数时不指定参数,则会报错,编译器不知道调用哪一个函数

## this指针

### c语言中类的实现

使用struct结构体和含有类指针(this)作为参数的函数

在c++,这一指针被作为成员函数的隐式参数(不被写出来),保证了成员函数对于自身成员变量和其他成员函数的访问权限

### c++中的this

作用:指向成员函数所作用的对象

非静态成员函数可以直接使用 **this** 来代表指向该函数作用的对象指针

在C++中, this指针是一个隐式参数, 指向当前对象的指针。它可以用来访问当前对象的成员变量和成员函数。以下是this指针的主要使用时机:

#### 1. 消除同名歧义:

当成员变量与局部变量或函数参数同名时, 可以使用this指针来区分它们。

```
class Person {
private:
 std::string name;
public:
 void setName(const std::string& name) {
 this->name = name; // 使用this指针区分同名的成员变量和参数
 }
};
```

## 2. 返回当前对象的引用:

在某些情况下，我们希望函数返回当前对象的引用，以支持**链式调用**。这时可以使用this指针来返回当前对象的引用。

```
class Person {
public:
 Person& setAge(int age) {
 this->age = age;
 return *this; // 返回当前对象的引用，支持链式调用
 }
private:
 int age;
};

Person person;
person.setAge(20).setAge(30); // 链式调用
```

## 3. 实现自赋值检查:

在赋值运算符重载函数中，我们需要检查自赋值(即用自己给自己赋值)的情况，以避免不必要的操作和资源释放。可以使用this指针来比较当前对象与传入的对象是否相同。

```
class MyString {
public:
 MyString& operator=(const MyString& other) {
 if (this != &other) { // 检查自赋值
 // 执行赋值操作
 }
 return *this;
 }
};
```

## 4. 传递当前对象的指针:

有时我们需要将当前对象的指针传递给其他函数或对象，可以直接使用this指针。

```
class Widget {
public:
 void registerObserver(Observer* observer) {
 observer->addWidget(this); // 将当前对象的指针传递给Observer
 }
};
```

## 5. 在const成员函数中访问非const成员变量:



在const成员函数中，this指针的类型为const指针，不能修改成员变量。但是，如果我们需要在const成员函数中访问非const成员变量，可以使用const\_cast来去除this指针的const限定。

```
class MyClass {
public:
 void constFunction() const {
 int* ptr = const_cast<int*>(&this->nonConstMember); // 去除this指针的const
 // 访问非const成员变量
 }
private:
 int nonConstMember;
};
```

需要注意的是，在使用const\_cast去除this指针的const限定时，要确保对非const成员变量的访问是合理且必要的，不会破坏对象的常量性。

## 静态static

- 类
  - 静态成员
    - 普通成员变量人手一份,静态成员变量所有类对象公有
    - sizeof运算符不会计算静态成员变量
    - 普通成员函数必须具体作用于某个对象,静态成员函数不具体作用于某个对象
    - 静态成员不需要通过对象就可以访问
    - 静态成员变量必须在类外初始化，而且只能初始化一次。(实质是全局变量,但是仍要遵循权限访问规则)
    - 静态成员变量和静态成员函数可以通过类名和作用域解析运算符 (::) 访问，也可以通过对象访问。
      - 类名::成员名
      - 对象名.成员名
      - 指针->成员名
      - 引用.成员名
    - 设置静态成员这种机制的目的
      - 将与某些类紧密相关的全局变量和函数写到类里面
      - 看上去像一个整体
      - 易于维护和理解

在静态成员函数中,不能访问非静态的成员(因为不作用于具体对象)

## 常量成员函数

### RECALL:

- 变量
  - 声明不可变变量。

```
const int maxScore = 100;
```

- 指针

- 指向常量数据的指针（指针指向的数据不能被改变,但是可以通过修改数据本身达到修改数据的效果,只是不能通过指针修改）。

```
const int* ptr = &value;
```

- 常量指针（指针自身的值不能被改变，但指向的数据可以改变）。

```
int* const ptr = &value;
```

- 指向常量数据的常量指针（指针自身和指向的数据都不能被改变）。

```
const int* const ptr = &value;
```

- 引用

- 对常量的引用（通过引用不能修改其绑定的变量,但是可以通过修改值本身来达到修改的目的）。

```
const int& ref = value;
```

- 函数

- 函数如果在参数列表后使用 `const`，说明该函数是一个**常量成员函数**，不会修改对象的状态。(可以用来区分函数重载)

```
int getValue() const { /* ... */ }
```

- `const` 可以用于函数参数，表示参数在函数内不能被修改。

```
void myFunction(const int param) { /* ... */ }
```

- 函数返回值前使用 `const`，表示返回的值是常量，不能被修改。

```
const int getNumber() { return 42; }
```

- 类的成员

- 类成员函数声明为 `const`，意味着它们不会修改**对象的成员变量**。

```
class MyClass {
public:
 int getValue() const { /* ... */ }
};
```

- `const` 数据成员，一旦被初始化后不能被修改。

```
class MyClass {
private:
 const int constantValue;
public:
 MyClass(int v) : constantValue(v) };

```

使用 `const` 可以提高代码的安全性, 防止不小心修改不应该改变的数据, 同时可以帮助编译器优化代码。在设计类和函数接口时, 合理使用 `const` 可以使得接口的使用更加清晰, 也更容易理解代码的意图。

- 常量对象

- 具有不可被修改的特质, 只能使用:

- 构造函数
    - 析构函数
    - 常量成员函数(const在参数列表之后) --> 成员函数的const重载
  - (第一次上机 2024/3/10)

常量对象只能调用静态成员函数和常量成员函数

程序填空, 使得程序输出:

```
2
1
1
0

```

```
#include <iostream>
using namespace std;
class A
{
 static int num;
public:
 A(){num+=1;}
 void func()
 {
 cout<< num <<endl;
 }
 // begin
 void func()const
 {
 num--;
 cout<<num<<endl;
 }
 // end
};

int A::num=1;

int main()
{
 A a1;
 const A a2 = a1;
 A & a3 = a1;
 const A & a4 = a1;
}

```

```

 a1.func();
 a2.func();
 a3.func();
 a4.func();

 return 0;
}

```

输入:

无

输出:

2

1

1

0

样例输入:

None

样例输出:

2

1

1

0

#### ■ 静态成员函数

- 常量成员函数

- 常量成员函数不可以修改非静态属性(成员变量)的值,也不能调用同类的非常量成员函数(静态成员函数除外)

在声明和定义常量成员函数都应该使用**const**关键字

```

// 抓住重载的本质 ==> 让编译器去选择合适的重载函数
#include <iostream>
using namespace std;
class CTest {
private:
 int n;

public:
 CTest() { n = 1; }
 int GetValue() const { return n; }
 int GetValue() { return 2 * n; }
};

int main() {
 const CTest objTest1;
 CTest objTest2;
 cout << objTest1.GetValue() << ", " << objTest2.GetValue();
 return 0;
}

```

## mutable 成员变量

可以在const成员函数中修改的成员变量.

```
class CTest {
public:
 bool GetData() const {
 m_n1++;
 return m_b2;
 }

private:
 mutable int m_n1; // mutable关键字 使得该变量可以在常量成员函数中访问和修改(在我看来像是一种补救)
 bool m_b2;
};
```

## 成员对象和封闭类

### 成员对象和封闭类

一个类的成员变量可以是另一个类的对象

有成员对象的类叫做**封闭类(enclosing)**

```
class CTyre {
 int radius;
 int width;

public:
 CTyre(int r, int w) : radius(r), width(w) {}
};

class CEngine {};

class CCar {
 int price;
 CTyre tyre;
 CEngine engine;

public:
 CCar(int p, int tr, int tw);
};

CCar::CCar(int p, int tr, int tw) : price(p), tyre(tr, tw){}; // 成员初始化列表

int main() {
 CCar car(20000, 17, 225);
 return 0;
}
```

如果 CCar 类不定义构造函数, 则: CCar car; // compile error

因为编译器不明白 car.tyre 该如何初始化;

car.engine 的初始化没问题, 用默认构造函数即可

任何生成封闭类对象的语句, 都要让编译器明白

-> 对象中的成员对象, 是如何初始化的

具体的做法就是,通过封闭类的构造函数的初始化列表

定义封闭类的构造函数时,添加初始化列表:

```
类名::构造函数(参数表):成员变量1(参数表),成员变量2(参数表),...
```

类对象的构造函数的初始化列表是一种特殊的语法,用于在构造函数中初始化类的成员变量。它位于构造函数的参数列表之后,冒号之前,由一系列以逗号分隔的成员初始化器组成。

初始化列表的语法如下:

```
class ClassName {
public:
 ClassName(parameters) : member1(value1), member2(value2), ... {
 // 构造函数体
 }
 // ...
private:
 MemberType1 member1;
 MemberType2 member2;
 // ...
};
```

其中, `member1(value1)` 和 `member2(value2)` 是成员初始化器,用于初始化对应的成员变量 `member1` 和 `member2`。 `value1` 和 `value2` 可以是常量、变量、表达式或其他类型的构造函数。

使用初始化列表的优点:

1. 必要性:对于常量成员、引用成员以及没有默认构造函数的类类型成员,必须在初始化列表中进行初始化,因为它们不能被赋值。
2. 效率:通过初始化列表直接初始化成员变量,避免了先调用默认构造函数,再进行赋值的过程,提高了效率。
3. 清晰性:初始化列表明确地指定了成员变量的初始值,使代码更加清晰和易于理解。

示例:

```
class Point {
public:
 Point(int x, int y) : x_(x), y_(y)
 // ...
private:
 int x_;
 int y_;
};

class Rectangle {
public:
 Rectangle(int x, int y, int width, int height)
 : topLeft_(x, y), width_(width), height_(height)
 // ...
private:
 Point topLeft_;
 int width_;
 int height_;
};
```

在上面的示例中, `Point` 类的构造函数使用初始化列表来初始化 `x_` 和 `y_` 成员变量。 `Rectangle` 类的构造函数使用初始化列表来初始化 `topLeft_` 成员对象(通过调用 `Point` 类的构造函数), 以及 `width_` 和 `height_` 成员变量。

使用初始化列表可以确保成员变量在构造函数体执行之前就已经被正确初始化了。这对于确保对象的有效性和一致性非常重要。

总之, 类对象的构造函数的初始化列表提供了一种方便、高效且清晰的方式来初始化类的成员变量。它在某些情况下是必须的, 并且可以提高代码的可读性和性能。

在初始化列表后面的构造函数体中, 您可以编写任何需要在对象构造时执行的附加逻辑或操作。这部分代码会在成员变量初始化完成后执行。如果您没有任何需要在构造函数体中执行的操作, 则可以将其留空或省略。

下面通过一个具体的示例来解释构造函数体的作用:

```
class Rectangle {
public:
 Rectangle(int width, int height) : width_(width), height_(height) {
 std::cout << "Constructor body: Creating a rectangle." << std::endl;
 area_ = calculateArea();
 }

 int getArea() const {
 return area_;
 }

private:
 int width_;
 int height_;
 int area_;

 int calculateArea() const {
 return width_ * height_;
 }
};
```

在这个示例中, `Rectangle` 类有一个构造函数, 它接受宽度(`width`)和高度(`height`)作为参数。构造函数使用初始化列表来初始化 `width_` 和 `height_` 成员变量。

在初始化列表之后, 构造函数体中有两行代码:

1. `std::cout << "Constructor body: Creating a rectangle." << std::endl;`  
这行代码输出一条消息, 表示正在创建一个矩形对象。这只是一个示例, 用于演示在构造函数体中执行一些操作。
2. `area_ = calculateArea();`  
这行代码调用了私有成员函数 `calculateArea()` 来计算矩形的面积, 并将结果赋值给 `area_` 成员变量。这是在构造函数体中执行的一个有意义的操作, 用于初始化对象的状态。

在这个例子中, 构造函数体的作用是在成员变量初始化完成后, 执行一些额外的操作, 如输出消息和计算面积。

下面是使用 `Rectangle` 类的示例:

```
int main() {
 Rectangle rect(5, 3);
 std::cout << "Area: " << rect.getArea() << std::endl;
 return 0;
}
```

输出结果:

```
Constructor body: Creating a rectangle.
Area: 15
```

可以看到,在创建 `Rectangle` 对象时,构造函数体中的代码被执行,输出了一条消息,并计算了矩形的面积。

当然,在许多情况下,您可能不需要在构造函数体中执行任何操作,因为成员变量的初始化可以通过初始化列表完成。这时,您可以简单地将构造函数体留空或省略:

```
Rectangle(int width, int height) : width_(width), height_(height)
```

总之,构造函数体提供了一个地方,让您可以在对象构造时执行一些额外的操作或逻辑。您可以根据需要在构造函数体中编写代码,也可以将其留空或省略,这取决于您的具体需求。

## 函数的调用顺序

- 封闭类对象生成时
  1. 执行所有成员对象的构造函数
  2. 执行封闭类的构造函数
- 对象成员的构造函数调用次序和**对象成员在类中的声明顺序**一致
- 与它们在成员初始化列表中出现的次序无关
- 封闭类的对象消亡时
  1. 执行封闭类的析构函数
  2. 执行成员对象的析构函数
- 次序和构造函数的调用次序相反

```
#include <iostream>
using namespace std;
class CTyre {
 int radius;
 int width;

public:
 CTyre(int r, int w) : radius(r), width(w) {
 cout << "CTyre contructor " << endl;
 }
 ~CTyre() { cout << "CTyre destructor" << endl; }
};

class CEngine {
 int engNum;

public:
```



```

CEngine(int n) {
 engNum = n;
 cout << "CEngine contructor " << endl;
}
~CEngine() { cout << "CEngine destructor" << endl; }
};

class CCar {
 int price;
 CEngine engine;
 CTyre tyre;

public:
 CCar(int p, int tr, int tw, int eNum);
 ~CCar() { cout << "CCar destructor" << endl; }
};

CCar::CCar(int p, int tr, int tw, int eNum)
 : price(p), tyre(tr, tw), engine(eNum) {
 cout << "CCar constructor" << endl;
};

int main() {
 CCar car(20000, 17, 225, 100);
 return 0;
}

```

```

CEngine contructor
CTyre contructor
CCar constructor
CCar destructor
CTyre destructor
CEngine destructor

```

## 封闭类的复制构造函数

封闭类的对象,如果是用默认复制构造函数初始化 -> 它里面包含的成员对象也会使用复制构造函数初始化

```

#include <iostream>
using namespace std;
class A {
public:
 A() { cout << "default" << endl; }
 A(A &a) { cout << "copy" << endl; }
};
class B {
 A a;
};
int main() {
 B b1, b2(b1);
 return 0;
}

```

```

// 说明b2.a是用类A的复制构造函数初始化,而调用复制构造函数时的 就是b1.a
default
copy

```

初始化 **const** 成员和引用成员时,必须在成员初始化列表中进行

```
#include <iostream>
using namespace std;
int f;
class CDemo {
private:
 const int num;
 int &ref;
 int value;

public:
 CDemo(int n) : num(n), ref(f), value(4) {}
};
int main() {
 cout << sizeof(CDemo) << endl;
 return 0;
}
```

12

## 友元

友元函数:

```
class Rectangle {
private:
 int width;
 int height;

public:
 Rectangle(int w, int h) : width(w), height(h)

 friend int calculateArea(const Rectangle& rect); // 声明友元函数(一个非成员函数,也可以在这里定义出来)
};

int calculateArea(const Rectangle& rect) {
 return rect.width * rect.height;
}

int main() {
 Rectangle rect(5, 3);
 int area = calculateArea(rect);
 std::cout << "Area: " << area << std::endl;
 return 0;
}
// 比如这样
#include <iostream>
class Rectangle {
private:
 int width;
 int height;
```

```

public:
Rectangle(int w, int h) : width(w), height(h) {}

friend int calculateArea(const Rectangle &rect) {
 return rect.width * rect.height;
}
};

int main() {
 Rectangle rect(5, 3);
 int area = calculateArea(rect);
 std::cout << "Area: " << area << std::endl;
 return 0;
}

```

在这个例子中, `Rectangle` 类有一个私有成员变量 `width` 和 `height`。我们希望定义一个独立的函数 `calculateArea` 来计算矩形的面积,但由于 `width` 和 `height` 是私有成员,通常情况下外部函数无法直接访问它们。

为了解决这个问题,我们可以将 `calculateArea` 函数声明为 `Rectangle` 类的友元函数。通过在类内部使用 `friend` 关键字声明该函数,我们授予了 `calculateArea` 函数访问 `Rectangle` 类的私有成员的权限。

友元函数虽然在类内部声明,但它不是类的成员函数,因此在定义时不需要使用类名作为前缀。

可以将一个类的成员函数 (包括构造/析构函数) 说明为另一个类的友元

```

class B {
public:
 void function();
};

class A {
 friend void B::function();
};

```

友元类:

```

class Engine {
private:
 int horsepower;

public:
 Engine(int hp) : horsepower(hp)

 friend class Car; // 声明 Car 类为 Engine 类的友元类
};

class Car {
private:
 Engine engine;

public:
 Car(int hp) : engine(hp)

 void displayEngineInfo() {

```

```

 std::cout << "Engine horsepower: " << engine.horsepower << std::endl;
 }
};

int main() {
 Car car(150);
 car.displayEngineInfo();
 return 0;
}

```

在这个例子中, `Engine` 类有一个私有成员变量 `horsepower`, 表示引擎的马力。 `Car` 类有一个 `Engine` 类型的成员变量 `engine`。

我们希望在 `Car` 类中能够直接访问 `Engine` 类的私有成员 `horsepower`, 以便在 `displayEngineInfo` 函数中显示引擎的相关信息。

为了实现这一点, 我们可以将 `Car` 类声明为 `Engine` 类的友元类。通过在 `Engine` 类内部使用 `friend class Car;` 语句, 我们授予了 `Car` 类访问 `Engine` 类的私有成员的权限。

这样, 在 `Car` 类的成员函数 `displayEngineInfo` 中, 我们可以直接访问 `engine.horsepower`, 即使 `horsepower` 是 `Engine` 类的私有成员。

友元的使用方法总结如下:

#### 1. 友元函数:

- 在类内部使用 `friend` 关键字声明友元函数。
- 友元函数可以访问类的私有成员。
- 友元函数在类内部声明, 但在类外部定义, 不属于类的成员函数。

#### 2. 友元类:

- 在一个类内部使用 `friend class ClassName;` 语句声明另一个类为友元类。
- 友元类可以访问当前类的私有成员。
- 友元关系是单向的, 即使类A是类B的友元类, 类B也不能自动访问类A的私有成员。

友元提供了一种突破类的封装性, 允许外部函数或类访问类的私有成员的机制。但是, 在使用友元时需要谨慎, 因为过度使用友元可能会破坏类的封装性和数据隐藏的原则。以下是一些使用友元的注意事项:

- 尽量避免过度使用友元, 只在必要时才使用友元函数或友元类。
- 友元函数和友元类应该是与当前类紧密相关的, 并且有合理的理由需要访问类的私有成员。
- 友元关系不是传递的, 即如果类A是类B的友元类, 类B是类C的友元类, 但类A不会自动成为类C的友元类。
- 友元关系不是对称的, 即如果类A是类B的友元类, 但类B不会自动成为类A的友元类。
- 友元声明只是一种访问权限的控制, 并不会影响类的实际实现。

总之, 友元函数和友元类提供了一种灵活的机制, 允许外部函数或类访问类的私有成员, 突破了类的封装性。但是, 在使用友元时需要谨慎, 应该根据实际需求和设计原则来决定是否使用友元。过度使用友元可能会导致类的封装性被破坏, 使代码的维护和理解变得更加困难。因此, 在使用友元时, 应该权衡利弊, 并遵循良好的编程实践和设计原则。

# 运算符重载

## 自定义数据类型和运算符重载

C++预定义了一组运算符,用来表示对数据的运算,但是只能用于基本的数据类型

- 整型
- 实型
- 字符型
- 布尔型

```
#include <iostream>

int main() {
 // 算术运算符
 int a = 5, b = 2;
 int c;

 c = a + b; // 加法
 c = a - b; // 减法
 c = a * b; // 乘法
 c = a / b; // 除法
 c = a % b; // 模除

 // 递增和递减运算符
 c = a++; // 后缀递增
 c = a--; // 后缀递减
 c = ++a; // 前缀递增
 c = --a; // 前缀递减

 // 关系运算符
 bool result;
 result = a == b; // 等于
 result = a != b; // 不等于
 result = a > b; // 大于
 result = a < b; // 小于
 result = a >= b; // 大于等于
 result = a <= b; // 小于等于

 // 逻辑运算符
 result = (a == 5) && (b == 2); // 逻辑与
 result = (a == 5) || (b == 3); // 逻辑或
 result = !(a == b); // 逻辑非

 // 位运算符
 c = a & b; // 位与
 c = a | b; // 位或
 c = a ^ b; // 位异或
 c = ~a; // 位取反
 c = a << 1; // 左移
 c = a >> 1; // 右移

 // 赋值运算符
 c = a; // 简单赋值
```

```

c += a; // 加后赋值
c -= a; // 减后赋值
c *= a; // 乘后赋值
c /= a; // 除后赋值
c %= a; // 求模后赋值
c <<= 2; // 左移后赋值
c >>= 2; // 右移后赋值
c &= a; // 按位与后赋值
c ^= a; // 按位异或后赋值
c |= a; // 按位或后赋值

// 其他运算符
int* p = &a; // 取地址运算符
c = *p; // 解引用运算符
c = b ? a : b; // 三元条件运算符

// 类型相关运算符
float f = static_cast<float>(a); // 类型转换运算符

// sizeof运算符
size_t size = sizeof(a); // 获取类型或对象所占字节的大小

// typeid运算符（需要包含头文件<typeinfo>）
// const std::type_info& ti = typeid(a); // 获取对象的类型信息

// new 和 delete 运算符
int* ptr = new int; // 动态分配单个整数
delete ptr; // 删除单个整数
ptr = new int[10]; // 动态分配整数数组
delete[] ptr; // 删除整数数组

// 函数调用运算符
// result = func(a, b); // 假设func是一个函数

// 下标运算符
// c = array[2]; // 假设array是一个整数数组

// 类成员访问运算符
// std::string str;
// char ch = str.at(2); // 访问对象成员函数

// 类成员指针访问运算符
// struct MyClass {
// int data;
// void method() // };
// MyClass obj;
// int MyClass::*ptrToData = &MyClass::data;
// void (MyClass::*ptrToMethod)() = &MyClass::method;
// obj.*ptrToData = 10; // 对象成员数据指针访问
// (obj.*ptrToMethod)(); // 对象成员函数指针访问

// 逗号运算符
// a = (b = 3, b + 2); // b被赋值为3, 然后a被赋值为b+2

// 显式类型转换运算符
// f = float(a); // C风格转换

```

```
// f = (float)a; // C++风格转换

return 0;
}
```

- 关于运算顺序

C++中的运算符优先级定义了表达式中运算符的运算顺序。当在表达式中混合使用多个运算符时，优先级高的运算符会先被计算。以下是C++中运算符的优先级顺序，从最高优先级（先计算）到最低优先级（后计算）：

1. 后缀运算符： `()`, `[]`, `->`, `.`, `++`, `--`
2. 一元运算符： `++`, `--`, `+`, `-`, `!`, `~`, `(type)`, `*`, `&`, `sizeof`
3. 类型转换运算符： `static_cast`, `dynamic_cast`, `const_cast`, `reinterpret_cast`
4. 乘除法运算符： `*`, `/`, `%`
5. 加减法运算符： `+`, `-`
6. 位移运算符： `<<`, `>>`
7. 关系运算符： `<`, `<=`, `>`, `>=`
8. 相等运算符： `==`, `!=`, `=`
9. 位AND运算符： `&`
10. 位XOR运算符： `^`
11. 位OR运算符： `|`
12. 逻辑AND运算符： `&&`
13. 逻辑OR运算符： `||`
14. 条件运算符（三元运算符）： `?:`
15. 赋值运算符： `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `^=`, `|=`
16. 逗号运算符： `,`

请注意，在运算符具有相同优先级的情况下，计算顺序通常是从左到右。唯一的例外是赋值运算符和一些一元运算符，它们的计算顺序是从右到左。

为了避免混淆和错误，最好使用圆括号来明确地指定运算的顺序，尤其是在涉及多个运算符的复杂表达式中。

## 自定义数据类型

C++提供了数据抽象的手段 ==> 允许用户自己定义数据类型, 通过调用类的成员函数, 对它的对象进行操作

## 运算符重载

希望：对一些抽象数据类型 (即自定义数据类型), 也能够直接使用C++提供的运算符, 这能使得:

- 程序更简洁
- 代码更容易理解
- 对已有的运算符(C++中预定义的运算符) 的含义
- 使 同一运算符作用于 不同的数据类型时导致不同类型的行为

- 目的:扩展C++中提供的运算符的适用范围, 以用于类所表示的**抽象数据类型**
- 同一个运算符, 对不同类型的操作数, 所发生的行为不同, 实质上是函数重载 (根据参数列表, 函数名称, const声明来判断是否是函数重载)
  - 注意和返回值类型无关, 怎样都可以
  - `(5, 10i) + (4, 8i) = (9, 18i)`
  - `5 + 4 = 9`

```
返回值类型 operator 运算符(形参列表){
 // 函数体
}
```

在程序编译的时候:

- 把运算符的表达式变成运算符函数的调用
- 把运算符的操作数变成运算符函数的操作数
- 运算符被多次重载时, 根据实参的类型决定调用哪个运算符函数
- 运算符既可以被重载为普通函数, 也可以被重载为类的成员函数
  - 重载为普通函数时, 参数个数为运算符目数
  - 重载为普通函数时, 参数个数为运算符目数-1(有一个隐式参数this)

```
#include <iostream>
using namespace std;
class Complex {
public:
 double real, imag;
 Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}
 Complex operator-(const Complex &c);
};
Complex operator+(const Complex &a, const Complex &b) {
 return Complex(a.real + b.real, a.imag + b.imag);
 // 返回一个临时对象
}
Complex Complex::operator-(const Complex &c) {
 return Complex(real - c.real, imag - c.imag);
 // 返回一个临时对象
}
int main() {
 Complex a(4, 4), b(1, 1), c;
 c = a + b; // 等价于c=operator+(a, b);
 cout << c.real << ", " << c.imag << endl;
 cout << (a - b).real << ", " << (a - b).imag << endl;
 // a-b等价于a.operator-(b)
 return 0;
}
```

```
5,5
3,3
```



## 赋值运算符=重载

- 赋值运算符两边的类型可以不匹配
  - 把一个 int 类型变量赋值给一个 Complex 对象
  - 把一个 char \* 类型的字符串赋值给一个字符串对象
  - 需要重载赋值运算符 '='
- 赋值运算符 '=' 只能重载为成员函数

```
#include <cstring>
#include <iostream>

using namespace std;

/*
编写一个 String
包含一个 char * 的成员变量指向动态分配的存储空间
该存储空间用于存放 '\0' 结尾的字符串
*/
class String {
private:
 char *str;

public:
 String() : str(new char[1]) { str[0] = 0; }
 const char *c_str() { return str; }
 String &operator=(const char *s);
 ~String() { delete[] str; }
};

// 深拷贝的赋值运算符重载
String &String::operator=(const char *s) {
 delete[] str;
 str = new char[strlen(s) + 1];
 strcpy(str, s);
 return *this;
}

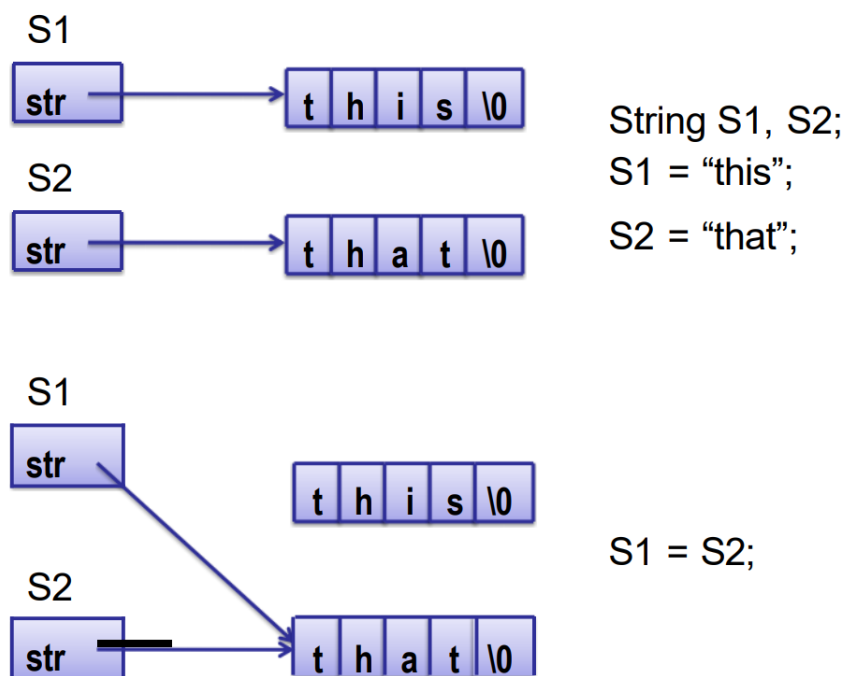
int main() {
 String s;
 s = "Good Luck, "; // 等价于 s.operator = ("Good Luck, ");
 cout << s.c_str() << endl;
 // String s2 = "hello!"; 这句代码如果不注释掉就会出错, 没有定义相关的赋值运算符
 s = "Shenzhou 13!"; // 等价于 s.operator = ("Shenzhou 13!");
 cout << s.c_str() << endl;
 return 0;
}
```

```
GoodLuck,
Shenzhou 13!
```

## 关于重载运算符=的探寻

- 浅拷贝

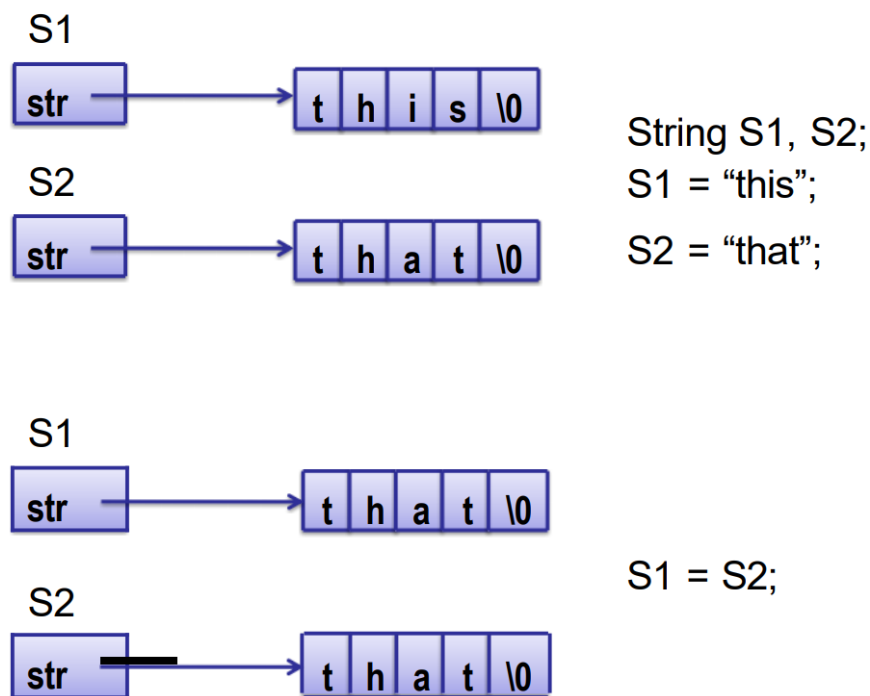
- 浅拷贝相当于使得左值的指针指向了右值的指针指向的位置
- 



- 如不定义自己的赋值运算符, 那么`S1=S2`实际上导致 `S1.str S2.str` 指向同一地方
- 如果S1对象消亡, 析构函数将释放S1.str指向的空间, 则S2消亡时还要释放一次, 不妥
- 另外, 如果执行 `S1 = "other";` 会导致S2.str指向的地方被delete

- 深拷贝

- 深拷贝相当于使得左值指向一片新开辟的内存空间, 该空间中存储的内容是右值存储的内容
- 



- 自赋值

- 如果在赋值运算符中没有对于自赋值的检测, 可能会导致未定义的错误, 同时浪费了资源

- ```
String & operator = (const String & s){
    if( this == &s ) return * this; // 通过这一步进行自赋值检测
    delete [] str; // 如果前一步没有检测,在这里会把原来的内存指向的位置删除,后一步的
    strcpy不能按照预期的方式工作
    str = new char[strlen(s.str)+1];
    strcpy( str, s.str);
    return * this;
}
```
- 赋值运算符的返回值类型
 - 运算符重载时,好的风格 --> 尽量保留运算符的原本的特性(即确有这个运算符所具有的一般性含义)
 - 在上面的例子中,返回值的类型为 `String &`,这使得下面的运算操作成为了可能(既可以使用 `.` 成员操作符,也节省了空间)
 - ```
// 考虑以下
a = b = c;
(a = b) = c; // (a==b)返回的值的类型时对a的引用

a.operator=(b.operator=(c)); // 既完成了对于b的赋值又返回了新的对于b的引用
(a.operator=(b)).operator=(c);
```

## 运算符重载为友元

- 一般情况下,将运算符重载为类的成员函数,是较好的选择
- 重载为成员函数不能满足使用要求,重载为普通函数(成员函数对于参数的顺序有绝对的要求,即this一定位于运算符的左侧)
- 普通函数又不能访问类的私有成员,所以需要将运算符重载为友元函数(针对于运算符的两个操作数倒序的情况)

- ```
class Complex {
public:
    Complex(double r = 0.0, double i = 0.0)
        : real(r), imaginary(i){}; // constructor
    Complex operator+(int r) { return Complex(real + r, imaginary); }
    friend Complex operator+(int r, const Complex &C);

private:
    double real; // real part
    double imaginary; // imaginary part
};
```

- 经过上述的重载,可以满足下面两种运算

- ```
c = 5 + c // 友元函数重载
c = c + 5 // 成员函数重载
```

## 流操作符的重载(插入,输出)

- 重载

在C++中,流操作符(如 `<<` 和 `>>`)通常用于输入输出操作,分别对应于插入操作(向流中写入数据)和提取操作(从流中读取数据)。这些操作符可以通过重载来扩展到用户定义的类型。

以 `<<` 操作符为例，当你想要让自定义类型支持插入到输出流（如 `std::ostream`），你需要定义一个非成员函数重载该操作符。这是因为第一个参数通常是一个流对象，而你不能直接修改标准库中的 `std::ostream` 类来添加成员函数。以下是一个简单的例子，演示如何为自定义类型 `MyClass` 实现 `<<` 流插入操作符：

```
#include <iostream>

class MyClass {
public:
 int value;

 MyClass(int val) : value(val) // 你可以将友元函数放在类内部，这样它可以访问类的私有成员
 friend std::ostream& operator<<(std::ostream& os, const MyClass& obj);
};

// 实现插入操作符
std::ostream& operator<<(std::ostream& os, const MyClass& obj) {
 os << obj.value; // 输出 MyClass 的 value 成员
 return os; // 返回 std::ostream 对象的引用，以支持链式调用
}

int main() {
 MyClass myObject(10);
 std::cout << myObject << std::endl; // 使用流操作符输出 myObject 的值
 return 0;
}
```

流提取操作符 `>>` 的重载方式类似，但通常用于输入流（如 `std::istream`）：

```
#include <iostream>

class MyClass {
public:
 int value;

 MyClass() : value(0) // 友元函数，以便可以访问私有成员
 friend std::istream& operator>>(std::istream& is, MyClass& obj);
};

// 实现提取操作符
std::istream& operator>>(std::istream& is, MyClass& obj) {
 is >> obj.value; // 从输入流中提取值到 MyClass 的 value 成员
 return is; // 返回 std::istream 对象的引用，以支持链式调用
}

int main() {
 MyClass myObject;
 std::cout << "Enter a number: ";
 std::cin >> myObject; // 使用流操作符从用户输入中读取值
 std::cout << "You entered: " << myObject.value << std::endl;
 return 0;
}
```

在上述例子中，`<<` 和 `>>` 操作符被重载为友元函数，这样它们就可以访问类的私有成员。这些操作符的返回类型是流的引用（`std::ostream&` 和 `std::istream&`），这是为了支持链式调用（如 `std::cout << obj1 << obj2`）。

在实现自定义流操作符时，要确保：

1. 对于输出操作符（`<<`），你应该只读取对象的状态，不要修改它。
2. 对于输入操作符（`>>`），你将改变对象的状态，根据从流中读取的数据设置对象的属性。
3. 两个操作符都应该返回流的引用，以便它们可以被链式使用。
4. 如果在输入操作符中遇到错误，应该设置流的错误状态（通过调用 `is.setstate(std::ios::failbit)`）。
5. 通常，流操作符会被重载为非成员函数，而且经常被声明为类的友元，以便它们可以访问类的私有和保护成员。

#### • 运算顺序

当你看到流操作符用于输出时，例如：

```
std::cout << obj1 << " " << obj2 << std::endl;
```

这条语句实际上是多个函数调用的链式组合。每个 `<<` 操作符都是一个函数调用，它接受左边的 `std::ostream` 对象的引用和右边的数据，然后返回 `std::ostream` 对象的引用以供下一个操作符使用。这意味着上面的语句可以展开成以下形式：

```
operator<<(operator<<(operator<<(std::cout, obj1), " "), obj2);
operator<<(std::cout, std::endl); // std::endl 也是一个函数
```

这里我们看到，每个 `operator<<` 函数都是从左到右依次调用的。第一个 `operator<<` 调用将 `std::cout` 和 `obj1` 作为参数，处理它们，然后返回 `std::ostream` 对象的引用。然后这个引用和下一个参数（在这个例子中是一个字符串 `" "`）被传递给下一个 `operator<<` 调用，以此类推。

`std::endl` 是一个特殊的操作符，因为它是一个流操纵符（manipulator），实际上是一个接受 `std::ostream` 引用的函数，执行换行和刷新缓冲区的操作。

所以，假设 `obj1` 和 `obj2` 都是 `MyClass` 类型的对象，其流插入操作符已经被重载，整个流表达式可以被展开为对 `operator<<` 的一系列调用：

```
std::ostream& temp1 = operator<<(std::cout, obj1); // 调用自定义的 operator<<
std::ostream& temp2 = operator<<(temp1, " "); // 调用标准库提供的
operator<<
std::ostream& temp3 = operator<<(temp2, obj2); // 再次调用自定义的
operator<<
operator<<(temp3, std::endl); // 调用处理 std::endl 的
operator<<
```

上述代码中，每一个 `temp` 是一个 `std::ostream` 类型的引用，它代表流的当前状态。这种链式调用的方式使得我们可以连续地将多个值插入到流中。

对于输入流使用 `>>` 操作符，逻辑是类似的，只是方向相反，而且通常涉及到改变右侧参数的状态。如：

```
std::cin >> obj1 >> obj2;
```

可以被看作是：

```
std::istream& temp = operator>>(std::cin, obj1); // 调用自定义的 operator>>
operator>>(temp, obj2); // 再次调用自定义的 operator>>
```

这里，每个 `operator>>` 调用改变了 `obj1` 和 `obj2` 的状态，并返回 `std::istream` 类型的引用以供下一个操作符使用。

- 输入输出操作

**流操作符 `<<` 和 `>>`** 在每一步调用中都会执行实际的输入或输出操作。具体来说，每当这些操作符被调用时，它们会执行与它们重载函数定义中相对应的操作。

对于输出流操作符 `<<`（也就是 `operator<<`）：

1. 对于 `std::ostream`，当你执行 `std::cout << obj1`，假设 `obj1` 是一个原始类型（如 `int`，`double`），`std::cout` 的成员函数将直接被调用，它将 `obj1` 的值格式化为字符序列并将它们发送到标准输出（通常是终端或控制台）。
2. 如果 `obj1` 是一个自定义类型，并且你重载了 `operator<<` 来处理它，那么你的函数将在这一步被调用。你的函数定义了如何将对象的状态转换为字符序列，并将这些字符通过输出流对象发送。

对于输入流操作符 `>>`（也就是 `operator>>`）：

1. 当执行 `std::cin >> obj1` 时，如果 `obj1` 是一个原始类型，`std::cin` 的成员函数会从输入流中提取字符序列，并尝试将它们转换为 `obj1` 的类型，然后将值存储在 `obj1` 中。
2. 如果 `obj1` 是一个自定义类型，你需要为该类型提供一个重载的 `operator>>` 函数。这个函数定义了如何从输入流中读取数据，并根据读取的数据设置 `obj1` 的状态。

在链式调用中，如 `std::cout << obj1 << " " << obj2;`，每个 `operator<<` 的调用相当于一步输出操作。第一个 `operator<<` 将 `obj1` 的表示输出到流中，下一个将字符串 `" "` 输出到流中，接着是 `obj2` 的表示。这些调用是顺序执行的，因此输出操作的顺序与代码中操作符的顺序相同。

同理，对于输入操作 `std::cin >> obj1 >> obj2;`，每个 `operator>>` 的调用相当于一步输入操作。第一个 `operator>>` 从流中读取数据并存储到 `obj1`，接着第二个 `operator>>` 读取流中接下来的数据并存储到 `obj2`。

在每一步中，流对象会维护一个状态，它记录了当前的读或写位置。在输出时，如果流对象处于错误状态（比如因为之前的输出失败），后续的输出操作通常会被忽略。同样地，在输入时，如果流对象因为错误而设置了错误标志（例如类型不匹配或读取失败），后续的输入操作也会因为流的错误状态而可能不执行。

## `operator[]` 的重载

```
#include <iostream>
#include <memory>
using namespace std;
// 考虑定义一个整数数组类
class Array {
public:
```

```

Array(int n = 10) : size(n) { ptr = new int[n]; }
 // 补充定义复制构造函数,完成形如 Array b(a); 方式的初始化,而缺省的复制构造函数不能完成数
 组元素空间的分配
Array(Array &a) {
 ptr = new int[a.size];
 memcpy(ptr, a.ptr, sizeof(int) * a.size);
 size = a.size;
}

~Array() { delete[] ptr; }
int &operator[](int subscript) {
 return ptr[subscript];
 // 这里的ptr虽然只是一个指针,但是当使用[]运算符的时候,会被认为是一个该类型的数组
}
// 使用引用作为返回值是为了可以实现我们常用的赋值操作
const Array &operator=(const Array &a) {
 if (ptr == a.ptr) {
 return *this;
 }
 delete[] ptr;
 ptr = new int[a.size];
 memcpy(ptr, a.ptr, sizeof(int) * a.size);
 /*
 memcpy是内存拷贝函数, 要 #include <memory>
 它将从a.ptr起的sizeof(int) * a.size 个字节拷贝到地址ptr
 */
 size = a.size;
 return *this;
}
// 这里的常引用作为返回值有利有弊, 建议使用引用作为返回值, 这样方便后续的修改, 当然也要考虑的需求

private:
 int size; // size of the array
 int *ptr; // pointer to first element of array
};
// 该类的对象就代表一个数组
// 希望能像普通数组一样使用该类的对象。例如:
int main() {
 // 访问和赋值
 Array a(20);
 a[18] = 62;
 int n = a[18];
 cout << a[18] << ", " << n;
 // 两个类对象之间赋值
 Array c(20), b(30);
 c[18] = 62;
 b[18] = 100;
 b[25] = 200;
 cout << c[18] << ", " << a[25];

 return 0;
}

```

## 重载类型转化运算符

- operator type();
  - 必须为 , 不指定返回类型, 形参为空;
  - 一般不改变被转换对象(即生成临时对象), 因此常定义为const
  - 类型转换自动调用
- `operator TypeName() const;`

```
#include <iostream>
using namespace std;
class Sample {
private:
 int n;

public:
 Sample(int i) {
 n = i;
 cout << "constructor called" << endl;
 }
 Sample operator+(int k) {
 Sample tmp(n + k);
 return tmp;
 }
 operator int() { // 可以指定返回值类型也可以不指定
 cout << "int convertor called" << endl;
 return n;
 }
};

int main() {
 Sample s(5);
 s = s + 4;
 cout << s << endl;
 cout << 3 + s << endl;
 s = 3 + s;

 Sample a(5);
 a = a + 4;
 cout << (int)a << endl;
 cout << 3 + (int)a << endl;
 s = 3 + (int)a;
 return 0;
}
// 还不完备, 没有重载赋值运算符=
```

```
constructor called
constructor called
int convertor called
9
int convertor called
12
int convertor called
constructor called
```



```
constructor called
constructor called
int convertor called
9
int convertor called
12
int convertor called
constructor called
```

## 关于返回值

在C++中，类型转换运算符（也称为转换函数）确实不需要指定返回类型，因为其返回类型隐含在运算符本身。类型转换运算符用于定义类或结构的隐式或显式转换行为。(指定了之后会报错)

类型转换运算符的一般形式如下：

```
operator TypeName() const;
```

这里的 `TypeName` 是你希望转换到的类型。这个运算符必须是类的成员函数，并且不能指定返回类型，因为返回类型已经由运算符的名称隐式给出。

例如，如果你有一个名为 `MyClass` 的类，并希望它能够转换为 `int` 类型，你可以这样写：

```
class MyClass {
public:
 ...
 operator int() const {
 return value; // 假设value是能代表MyClass对象状态的int类型的成员
 }
 ...
};
```

在这个例子中，`operator int()` 是一个类型转换运算符，它允许 `MyClass` 的对象在需要 `int` 类型时自动转换成 `int`。这意味着如果你有一个 `MyClass` 对象，你可以直接在需要 `int` 值的表达式中使用它，而不需要显式地调用转换函数。

需要注意的是，滥用隐式类型转换可能会导致代码难以理解和维护，因为它可能在不明显的地方引入了转换，有时会产生意外的结果。为了提高代码的清晰度，有时候建议使用 `explicit` 关键字来防止隐式转换，这样就需要显式地调用转换运算符。

## 自增自减运算符的重载

- 前置运算符作为一元运算符重载(只有一个隐式参数this)
- 后置运算符作为二元运算符重载(有隐式参数this和一个占位符)

自增 (`++`) 和自减 (`--`) 运算符可以被重载以便它们能被用于用户自定义的类型。这些运算符通常有两种形式：前缀形式和后缀形式。

前缀形式直接对对象进行增加或减少操作，然后返回对象的引用。后缀形式也会进行增加或减少操作，但它返回对象操作前的值，这要求函数创建并返回对象的一个临时副本。

下面是自增和自减运算符重载的一个例子，使用一个简单的 `Integer` 类来演示：

```
class Integer {
public:
```

```

Integer(int value) : value(value) // 前缀自增重载
Integer& operator++() {
 ++value; // 增加对象的值
 return *this; // 返回当前对象的引用
}

// 后缀自增重载
Integer operator++(int) {
 Integer temp = *this; // 保存当前对象的副本
 ++(*this); // 使用前缀自增来增加对象的值
 return temp; // 返回原始对象的副本
}

// 前缀自减重载
Integer& operator--() {
 --value; // 减少对象的值
 return *this; // 返回当前对象的引用
}

// 后缀自减重载
Integer operator--(int) {
 Integer temp = *this; // 保存当前对象的副本
 --(*this); // 使用前缀自减来减少对象的值
 return temp; // 返回原始对象的副本
}

int getValue() const { return value; } // 获取对象的值

private:
 int value;
};

int main() {
 Integer num(10);

 ++num; // 使用重载的前缀自增运算符
 std::cout << num.getValue() << std::endl; // 输出 11

 num++; // 使用重载的后缀自增运算符
 std::cout << num.getValue() << std::endl; // 输出 12

 --num; // 使用重载的前缀自减运算符
 std::cout << num.getValue() << std::endl; // 输出 11

 num--; // 使用重载的后缀自减运算符
 std::cout << num.getValue() << std::endl; // 输出 10

 return 0;
}

```

在上述代码中，我们为 `Integer` 类重载了自增和自减运算符。注意到后缀版本接受一个未使用的 `int` 参数，这是为了区分前缀版本和后缀版本的重载；这个参数是个占位符，用来告诉编译器这是后缀运算符重载。

不需要你在调用时显式传入占位符参数。编译器会根据你使用运算符的方式自动区分你是想要调用前缀版本还是后缀版本，并且在后缀版本中自动传入一个占位值。

当你在对象上使用自增或自减运算符时，如果你在运算符之后没有加任何东西（即 `++object` 或 `--object`），那么编译器会识别为前缀版本。如果你在运算符之后加了一个空的括号对（即 `object++` 或 `object--`），编译器会识别为后缀版本，并自动提供一个整数参数。

这是自动完成的。在后缀形式的成员函数定义中，哑元参数的存在只是为了语法上的区分，编译器不期望你在调用时提供这个参数。

例如：

```
Counter counter(10);
Counter before, after;

before = counter++; // 调用后缀自增，编译器自动传入参数
after = ++counter; // 调用前缀自增
```

在这个例子中，编译器会自动理解 `counter++` 应该调用后缀形式的 `operator++(int)`，而 `++counter` 应该调用前缀形式的 `operator++()`。你不需要（也不可能）显式地为后缀版本提供占位参数。

在C++中，后缀自增和自减运算符的占位符参数的数据类型是固定的，必须是 `int` 类型。这是由C++语言标准规定的，用以区分重载的前缀和后缀版本的自增和自减运算符。

这个 `int` 类型的参数不用于函数的实现，它只是一个语法要求，使得编译器能够区分前缀和后缀形式。因此，你不能使用其他类型作为这个占位符参数。如果你尝试使用其他类型，如 `double` 或 `long`，编译器将不会接受它作为后缀自增或自减的合法重载。

下面是一个正确的后缀自增运算符重载的例子：

```
class MyClass {
public:
 // 其他成员函数...

 // 后缀自增运算符重载
 MyClass operator++(int) {
 MyClass temp = *this;
 // 实际增加操作...
 return temp;
 }
};
```

在这个例子中，虽然有一个占位符参数（通常表示为 `int`），但在实现中我们并不使用它。它只是告诉编译器这个运算符是后缀版本的自增运算符。

前缀版本直接修改对象并返回引用，而后缀版本需要创建一个当前对象状态的副本，然后修改对象的状态并返回副本。这意味着后缀版本的性能可能会稍微差一点，因为它涉及到额外的对象副本的创建。

在使用重载的自增和自减运算符时，需要保证这些操作符对你的类型是有意义的，以免违反直觉。通常，这些操作符用于迭代或者类似数字行为的自定义类型。

## 运算符重载的注意事项

- C++不允许定义新的运算符
- 所有的运算符重载都对于参数的顺序有严格要求
- 重载后运算符的含义应该符合日常习惯
  - `complex_a + complex_b`
  - `word_a > word_b`
  - `date_b = date_a + n`
- 运算符重载不改变运算符的优先级
- 以下运算符不能被重载: `.`, `::`, `?:`, `sizeof`
- 重载运算符 `()`, `[]`, `->` 或者赋值运算符 `=` 时, 运算符重载函数必须声明为类的成员函数
- 重载运算符是为了让它能作用于对象, 因此重载运算符不允许操作数都不是对象
  - 这句话的意思是, 在C++中, 运算符重载是通过在类中创建特定的成员函数或友元函数来实现的, 以便让运算符能够用于类的对象。一个重载的运算符至少需要有一个操作数是用户定义的类型（即对象），这是因为重载的本质是提供了一个类的成员函数或友元函数的新版本。

举个例子, 假设我们有一个 `Complex` 类用于表示复数, 我们可以重载加法运算符 `(+)` 以便它能用于 `Complex` 对象:

```
class Complex {
public:
 Complex(double r, double i) : real(r), imag(i) // 重载 "+" 运算符, 使之能够用于Complex对象之间的加法
 Complex operator+(const Complex& other) const {
 return Complex(real + other.real, imag + other.imag);
 }

private:
 double real, imag;
};

int main() {
 Complex c1(1.0, 2.0);
 Complex c2(2.0, 3.0);
 Complex c3 = c1 + c2; // 使用重载的 "+" 运算符
 return 0;
}
```

在上面的代码中, `operator+` 被重载为 `Complex` 类的成员函数, 它允许我们将两个 `Complex` 对象相加。至少有一个操作数必须是 `Complex` 类型, 这样编译器才能知道应该调用哪个重载版本的运算符。

如果尝试重载一个运算符, 而所有的操作数都是内置类型 (如 `int`, `float` 等), 这是不允许的, 因为这将会改变语言已有的基本行为, 而C++不允许这样做。例如, 你不能为两个整数或两个指针重载加法运算符。

简而言之, 运算符重载的目的是为了让用户定义的类型能够使用运算符, 至少其中一个操作数必须是该类型的对象。这也是一种封装和多态的表现, 使得自定义类型在使用运算符时能够像内置类型一样自然。

- 有一个操作数是枚举类型也可以(不推荐使用)

## 运算符重载的应用

### 作为类传入其他函数

```
#include <algorithm>
#include <cstring>
#include <iostream>
#include <list>

using namespace std;

class TV_Drama {
public:
 char name[100];
 int actor;
 int story;
 int acting_skill;
 TV_Drama(char *_name, int _actor, int _story, int _acting_skill) {
 strcpy(name, _name);
 actor = _actor;
 story = _story;
 acting_skill = _acting_skill;
 }
};

void Printer(const TV_Drama &a) { cout << a.name << ' '; }
bool comparator_1(const TV_Drama &a, const TV_Drama &b) {
 return a.story >= b.story;
}

bool operator<(const TV_Drama &a, const TV_Drama &b) {
 return a.actor >= b.actor;
}

class comparator_2 {
public:
 bool operator()(const TV_Drama &a, const TV_Drama &b) {
 return a.acting_skill >= b.acting_skill;
 }
};

int main() {
 list<TV_Drama> lst;
 int n;

 cin >> n;
 char _name[100];
 int _actor, _story, _acting_skill;
 for (int i = 0; i < n; i++) {
 cin.ignore();
 cin.getline(_name, 100);
 cin >> _actor >> _story >> _acting_skill;
 lst.push_back(TV_Drama(_name, _actor, _story, _acting_skill));
 }

 lst.sort();
 for_each(lst.begin(), lst.end(), Printer);
}
```

```

cout << endl;

lst.sort(comparator_1);
for_each(lst.begin(), lst.end(), Printer);
cout << endl;

lst.sort(comparator_2());
for_each(lst.begin(), lst.end(), Printer);
cout << endl;

return 0;
}

```

在C++中，使用像 `comparator_2` 这样的类来定义排序逻辑是一种实现自定义比较器的方法。这种方法利用了C++的运算符重载和对象的可调性（即函数对象，也称为仿函数 `functor`），允许对象被用作普通函数的方式调用。

函数对象（`Functor`）是一种使用对象来模拟函数的方式，这些对象拥有一个返回值和接受参数的 `operator()` 成员函数。它们可以存储状态（即成员变量），并且可以被传递和复制，而传统的函数指针则不能容易地做到这些。

这里是 `comparator_2` 类的修正版实现，使其成为一个有效的比较器（`functor`）：

```

class comparator_2 {
public:
 bool operator()(const TV_Drama &a, const TV_Drama &b) const {
 return a.acting_skill < b.acting_skill; // 注意这里是小于，以便于正常排序
 }
};

```

当使用 `lst.sort(comparator_2());` 时，你实际上创建了 `comparator_2` 的一个匿名（临时）对象，并将它传递给 `sort` 方法。`sort` 方法会使用这个对象作为比较函数，从而根据 `acting_skill` 成员的值对 `list` 中的 `TV_Drama` 对象进行排序。

使用这种方式的优点包括：

1. **封装性**：比较逻辑被封装在一个对象中，可以携带状态（如果需要）。
2. **可重用性**：比较器可以作为一个独立的对象，在不同的排序调用之间传递和复用。
3. **灵活性**：可以定义多种不同的比较逻辑，每种逻辑对应一个不同的比较器类。

C++标准库中很多其他算法也可以使用类似的方式使用函数对象，比如 `std::sort`，`std::stable_sort`，`std::max_element`，`std::min_element`，`std::set`（在构造函数中传递比较器来排序集合中的元素）等等。此外，函数对象也可以用于创建线程（通过 `std::thread`）或用于定义回调函数。

## 继承和派生

### 基本概念

- **继承**: 在定义一个新的类B时, 如果该类与某个已有的类A相似 (指的是 B 拥有A的全部特点 )
- A—基类 (父类 ), B—基类的派生类(子类 )
- 派生类是通过对基类进行修改和扩充得到的
  - **扩充**: 在派生类中, 可以添加新的成员变量和成员函数

- **修改**: 在派生类中, 可以重新编写从基类继承得到的成员
- 派生类一经定义后, 可以独立使用, 不依赖于基类

```
class 派生类名: 派生方式说明符 基类名{

}
```

派生类拥有基类的全部成员,包括

- private/protected/public 成员变量
- private/protected/public 成员函数(方法)

在派生类的各个成员函数中,不能访问基类中的**private**成员

关于这句话的理解

这句话的意思是,在派生类(子类)中,不能直接访问基类(父类)中被声明为**private**的成员(包括成员变量和成员函数)。这是因为**private**成员具有访问限制,只能在本类中被访问。

下面是一个具体的示例:

```
class Base {
 private:
 int privateVar;

 public:
 int publicVar;

 protected:
 int protectedVar;
};

class Derived : public Base {
 public:
 void accessMembers() {
 // 错误:不能访问基类的private成员
 // privateVar = 10;

 // 正确:可以访问基类的public成员
 publicVar = 20;

 // 正确:可以访问基类的protected成员
 protectedVar = 30;
 }
};
```

在上面的示例中,我们定义了一个基类 `Base`, 其中包含一个 `private` 成员变量 `privateVar`, 一个 `public` 成员变量 `publicVar`, 以及一个 `protected` 成员变量 `protectedVar`。

然后,我们定义了一个派生类 `Derived`, 它继承自 `Base` 类。在 `Derived` 类中,我们尝试访问基类的成员:

1. 尝试访问 `privateVar` 时,编译器会报错,因为 `privateVar` 是基类的 `private` 成员,派生类无法直接访问。
2. 访问 `publicVar` 是正确的,因为 `publicVar` 是基类的 `public` 成员,派生类可以直接访问。

3. 访问 `protectedVar` 也是正确的,因为 `protectedVar` 是基类的 `protected` 成员,派生类可以直接访问。

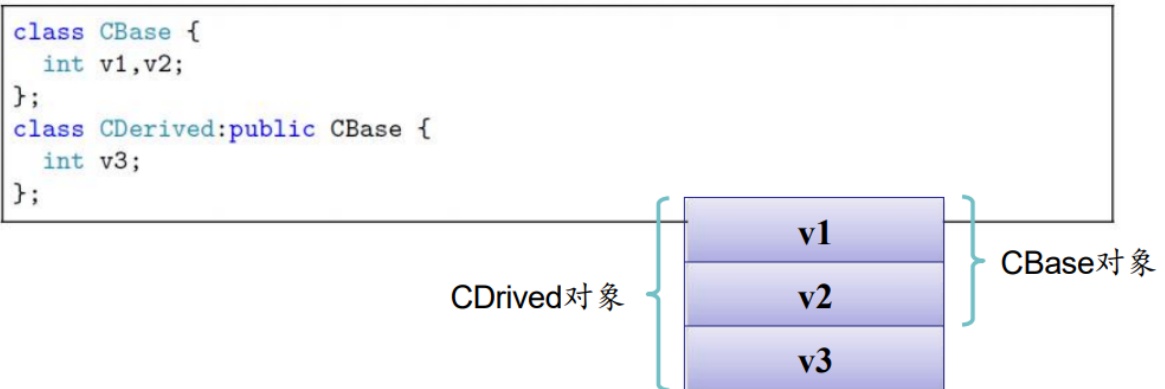
需要注意的是,虽然派生类不能直接访问基类的 `private` 成员,但是可以通过基类提供的 `public` 或 `protected` 接口(如果有)来间接访问 `private` 成员。

总结:派生类不能直接访问基类的 `private` 成员,这是为了保证封装性和数据的安全性。如果基类希望派生类能够访问某些成员,可以将它们声明为 `protected` 或 `public`。

## 派生类的内存空间

派生类对象的大小,等于基类对象的大小+派生类自己的成员变量大小

在派生类对象中,包含着基类对象,而且基类对象存储位置位于派生类对象新增的成员变量之前



## 复合与继承

在C++中, 继承和复合是实现代码重用的两种主要方式。下面将具体介绍它们在C++中的应用和区别。

### 继承

在C++中, 继承用于定义一个类(派生类)从另一个类(基类)继承属性和行为的机制。继承表示的是“是一个”(is-a)关系。

示例:

```
// 基类
class Animal {
public:
 void breathe() {
 std::cout << "呼吸空气" << std::endl;
 }
};

// 派生类
class Fish : public Animal {
public:
 void breathe() {
 std::cout << "通过腮呼吸水中的氧气" << std::endl;
 }
};
```



```
// 使用
int main() {
 Fish myFish;
 myFish.breathe(); // 输出：通过腮呼吸水中的氧气
 return 0;
}
```

在这个例子中，`Fish` 类继承了 `Animal` 类，并重写了 `breathe` 方法来表达鱼类特有的呼吸方式。

## 复合

复合是将一个类的实例（成员）嵌入到另一个类中，表示的是“有一个”（has-a）关系。

示例：

```
// 类Engine
class Engine {
public:
 void start() {
 std::cout << "发动机启动了" << std::endl;
 }
};

// 类Car
class Car {
private:
 Engine engine; // Car类有一个Engine对象

public:
 void startEngine() {
 engine.start(); // 委托给Engine的start方法
 }
};

// 使用
int main() {
 Car myCar;
 myCar.startEngine(); // 输出：发动机启动了
 return 0;
}
```

在这个例子中，`Car` 类通过包含一个 `Engine` 类的对象来使用 `Engine` 的功能，这样 `Car` 类就具有了发动机的功能。

## 继承 vs. 复合

在C++中，继承通常用于以下场景：

- 当你要建立两个类之间的一种类型和子类型关系时。
- 当你需要基类的接口，并打算按照基类的接口实现你的类时。

而复合通常用于以下场景：

- 当你要使用某个类的功能而不是它的接口时。

- 当你需要从多个来源重用代码时, C++不支持多重继承, 或者多重继承可能导致问题时, 复合可以作为一种手段。

在C++编程中, 通常建议尽可能使用复合来实现代码重用, 因为它提供了更高的灵活性和更低的耦合度。然而, 继承也是C++中一个非常重要的概念, 尤其是在定义抽象基类和实现多态时。因此, 应根据具体情况和设计需求来选择使用继承还是复合。

- 使用类的指针来作为复合的方式
  - 避免循环定义的方式:在一个类中使用另一个类的指针,而不是使用对象作为成员变量
- 派生类中可以定义一个和基类成员同名的成员,这叫做覆盖
  - 在派生类中访问这类成员时,缺省的情况是访问派生类中定义的成员
  - 如果要访问在基类中定义的同名成员时,要使用作用域符号 `::`
- 但是,一般基类和派生类不定义同名的成员变量

## 权限说明符: `protected`

- 基类的 `private` 成员:可以被下列函数访问
  - 基类的成员函数
  - 基类的友元函数
- 基类的 `public` 成员:可以被下列函数访问
  - 基类的成员函数
  - 基类的友元函数
  - 派生类的成员函数
  - 派生类的友元函数
  - 其他的函数
- 基类的 `protected` 成员:可以被下列函数访问
  - (可访问范围比 `private` 成员大, 比 `public` 成员小)
  - 基类的成员函数
  - 基类的友元函数
  - 派生类的成员函数可以访问当前对象的基类的保护成员
  - `this` 指针指向的对象
  - 也可以访问其他同类对象的基类的保护成员
  - 派生类的友元函数也可以访问基类的保护成员

```
class Father{
 private: int nPrivate;
 public: int nPublic;
 protected: int nProtected;
};
class Son : public Father
{
 void AccessFather () {
 nPublic = 1; // ok
 nPrivate = 1; // wrong
 nProtected = 1; // OK, 访问从基类继承的protected成员
 Son s;
 }
}
```

```

 s.nProtected = 1; // OK, 访问其他同类对象的基类保护成员
 Father f;
 f.nProtected = 1; // wrong, f不是函数所作用的当前对象
 }
};

int main() {
 Father f;
 Son s;
 f.nPublic = 1; // ok
 s.nPublic = 1; // ok
 f.nProtected = 1; // wrong
 f.nPrivate = 1; // wrong
 s.nProtected = 1; // wrong
 s.nPrivate = 1; // wrong
 return 0;
}

```

- 公有继承(public)
  - 基类的公有成员和保护成员作为派生类的成员时, 它们都**保持原有的状态**, 而基类的私有成员仍然是私有的, 不能 被这个派生类的子类所访问
- 私有继承 (private)
  - 基类的公有成员和保护成员都作为派生类的私有成员, 并且不能被这个派生类的子类所访问
- 保护继承 (protected)
  - 基类的所有公有成员和保护成员都成为派生类的保护成员, 并且只能被它的派生类成员函数或友元访问, 基类的私有成员仍然是私有的

|      | public    | protected | private |
|------|-----------|-----------|---------|
| 公有继承 | public    | protected | 不可见     |
| 私有继承 | private   | private   | 不可见     |
| 保护继承 | protected | protected | 不可见     |

## 生成与消亡

- 在创建派生类时, 需要调用基类的构造函数
  - 初始化派生类对象中从基类继承的成员
  - 在执行派生类的构造函数之前, 总是先执行基类的构造函数
- 调用基类构造函数的两种方式:
  - **显式方式:**在派生类的构造函数中, 为基类构造函数提供参数
    - `derived::derived(arg_derived-list):base(arg_base-list)`
  - **隐式方式:**在派生类的构造函数中, 省略基类构造函数时, 派生类的构造函数则自动调用基类的默认构造函数
- 派生类的析构函数被执行时, 执行完派生类的析构函数后, 自动调用基类的析构函数

```

#include <iostream>
using namespace std;

class Base {
public:
 int n;
 Base(int i) : n(i) {
 cout << "Base " << n << " constructed" << endl;
 }
 ~Base() {
 cout << "Base " << n << " destructed" << endl;
 }
};

class Derived : public Base {
public:
 Derived(int i) : Base(i) {
 cout << "Derived constructed" << endl;
 }
 ~Derived() {
 cout << "Derived destructed" << endl;
 }
};

int main() {
 Derived Obj(3);
 return 0;
}

```

// 构造函数的调用顺序是从基类到派生类，而析构函数的调用顺序是相反的，从派生类到基类。

```

Base 3 constructed
Derived constructed
Derived destructed
Base 3 destructed

```

```

#include <iostream>
using namespace std;

class Bug {
public:
 int nLegs;
 int nColor;
 Bug(int legs, int color) : nLegs(legs), nColor(color) {
 // 构造 Bug
 }
};

class Skill {
public:
 Skill(int n) {
 // 构造 Skill, 这里假设 Skill 类的构造函数没有输出
 }
}

```

```
};

class FlyBug : public Bug {
 int nWings;
 Skill sk1, sk2;
public:
 FlyBug(int legs, int color, int wings);
};

FlyBug::FlyBug(int legs, int color, int wings) :
 Bug(legs, color), sk1(5), sk2(color), nWings(wings) {
 // 构造 FlyBug
}

int main() {
 FlyBug fb(6, 4, 2); // 假设 Bug 有 6 条腿, 颜色为 4, 翅膀数为 2
 return 0;
}
```

请注意, `FlyBug` 构造函数的初始化列表首先调用基类 `Bug` 的构造函数, 然后按照它们在类定义中声明的顺序初始化 `sk1` 和 `sk2`。最后, `nWings` 成员变量被初始化。

- 创建派生类的对象时, 执行一个派生类的构造函数之前:
  - 调用基类的构造函数
  - 初始化派生类对象中从基类继承的成员
  - 调用成员对象类的构造函数(复合)
  - 初始化派生类对象中成员对象
- 派生类析构函数被执行时, 执行完派生类的析构函数后:
  - 调用成员对象类的析构函数
  - 调用基类的析构函数