C++基础

swh

2023 年 7 月 11 日

目录

Chapter

流程控制语句

1 for

1.1 template 模板

计算 $1+2+3+\cdots+100=$

```
#include <iostream>

using namespace std;

int main(){
        int sum = 0;
        for(int i = 1;i <= 100;i++){
                sum += i;
        }
        cout<<"sum = "<<sum<<endl;
        return 0;
}
```

2 for 循环

```
// 高斯求和公式求和 1+2+3+...+100
#include <iostream>
using namespace std;
int main() {
    int sum = 0;
```

```
    sum = (1+100)*100/2;
    cout << "gauss sum=" << sum << endl;
    return 0;
}
```

3

for 循环求和 1+2+3+...+100

```
int forSum(){
    int sum = 0;
    for (int i = 1; i <= 100; i++) {
        sum += i;
    }
    cout << "for sum=" << sum << endl;
    return 0;
}
```

4 if-else

```
// isPrime 判断是否素数
bool isPrime(int num) {
    if (num < 2)
        return false;
    for (int i = 2; i * i <= num; ++i) {
        if (num % i == 0)
            return false;
    }
    return true;
}

int prime(){
        bool b = isPrime(7);
        if (b == 1){
                cout<<"b是素数 "<<endl;
```

```
        }else{
                cout<<"b不是素数␣"<<endl;
        }
        return 0;
}
```

Chapter

bit 位运算

1 bit 位运算

$1 \ll i$ 等价于 2^i。

```
for(int i = 30;i >= 0; i--){
                int power = 1 << i;
}
```

2 include

2.1 fstream

```
#include <iostream>
#include <fstream>

using namespace std;

int main(){
        ifstream fin("power.in");
        ofstream fout("power.out");
        int n;
        fin>>n;
    fout<<n;
        fin.close();
        fout.close();
        return 0;
}
```

3 if else

3.1 CSP-J-2021-01-candy 分糖果

【题目背景】
红太阳幼儿园的小朋友们开始分糖果啦！
【题目描述】
红太阳幼儿园有 n 个小朋友，你是其中之一。保证 $n \geq 2$。
有一天你在幼儿园的后花园里发现无穷多颗糖果，你打算拿一些糖果回去分给幼儿园的小朋友们。
由于你只是个平平无奇的幼儿园小朋友，所以你的体力有限，至多只能拿 R 块糖回去。
但是拿的太少不够分的，所以你至少要拿 L 块糖回去。保证 $n \leq L \leq R$。
也就是说，如果你拿了 k 块糖，那么你需要保证 $L \leq k \leq R$。
如果你拿了 k 块糖，你将把这 k 块糖放到篮子里，并要求大家按照如下方案分糖果：只要篮子里有不少于 n 块糖果，幼儿园的所有 n 个小朋友（包括你自己）都从篮子中拿走恰好一块糖，直到篮子里的糖数量少于 n 块。此时篮子里剩余的糖果均归你所有——这些糖果是作为你搬糖果的奖励。
作为幼儿园高质量小朋友，你希望让作为你搬糖果的奖励的糖果数量（而不是你最后获得的总糖果数量！）尽可能多；因此你需要写一个程序，依次输入 n, L, R，并输出你最多能获得多少作为你搬糖果的奖励的糖果数量。
【输入格式】
从文件 candy.in 中读入数据。
输入一行，包含三个正整数 n, L, R，分别表示小朋友的个数、糖果数量的下界和上界。
【输出格式】
输出到文件 candy.out 中。
输出一行一个整数，表示你最多能获得的作为你搬糖果的奖励的糖果数量。
【样例 1 输入】
7 16 23
【样例 1 输出】
6
【样例 1 解释】
拿 $k = 20$ 块糖放入篮子里。
篮子里现在糖果数 $20 \geq n = 7$，因此所有小朋友获得一块糖；
篮子里现在糖果数变成 $13 \geq n = 7$，因此所有小朋友获得一块糖；
篮子里现在糖果数变成 $6 < n = 7$，因此这 6 块糖是作为你搬糖果的奖励。容易发现，你获得的作为你搬糖果的奖励的糖果数量不可能超过 6 块（不然，篮子里的糖果数量最后仍然不少

于 n，需要继续每个小朋友拿一块），因此答案是 6。

【样例 2 输入】

10 14 18

【样例 2 输出】

8

【样例 2 解释】

容易发现，当你拿的糖数量 k 满足 $14 = L \le k \le R = 18$ 时，所有小朋友获得一块糖后，剩下的 k − 10 块糖总是作为你搬糖果的奖励的糖果数量，因此拿 $k = 18$ 块是最优解，答案是 8。

【数据范围】

测试点	$n \le$	$R \le$	$R - L \le$
1	2	5	5
2	5	10	10
3	10^3	10^3	10^3
4	10^5	10^5	10^5
5	10^3	10^9	0
6			10^3
7	10^5		10^5
8	10^9		10^9
9			
10			

对于所有数据，保证 $2 \le n \le L \le R \le 10^9$

```
#include <iostream>
#include <fstream>

using namespace std;

int main(){
        ifstream fin("candy.in");
        ofstream fout("candy.out");
        int n,l,r;
        int k;
        fin>>n>>l>>r;
        if(l<n*(r/n)&&n*(r/n)<r){
                k = n * (r/n) -1-n*(r/n-1);
        }else if(n*(r/n)<l){
                k = r - n;
```

```
        }
        fout<<k;
        return 0;
}
```

4 & Reference 引用

```
#include <iostream>
#include <vector>

using namespace std;

void modifyVector(vector<int>& vec){
        vec.push_back(4);
}

int main(){
        vector<int> nums = {1,2,3};
        modifyVector(nums);

        for(int num : nums){
                cout<<num<<" ";
        }
        cout<<endl;
        return 0;
}
```

Chapter

程序基本概念

1 数据类型

使用编程语言进行编程时，需要用到各种变量来存储各种信息。变量保留的是它所存储的值的内存位置。这意味着，当您创建一个变量时，就会在内存中保留一些空间。
您可能需要存储各种数据类型（比如字符型、宽字符型、整型、浮点型、双浮点型、布尔型等）的信息，操作系统会根据变量的数据类型，来分配内存和决定在保留内存中存储什么。

1.1 基本的内置类型

表 .3.1: 几种基本的 C++ 数据类型

类型	关键字
布尔型	bool
字符型	char
整型	int
浮点型	float
双浮点型	double
无类型	void
宽字符型	wchar_t

一些基本类型可以使用一个或多个类型修饰符进行修饰：
signed
unsigned
short
long

1.2 自定义数据类型

1.2.1 struct 结构体

结构体是一种自定义数据类型，可以用来存储不同类型的数据成员。结构体可以包含多个成员变量，每个成员变量可以有不同的数据类型。结构体的定义使用关键字 struct，并通过花括号括起来列出成员变量。

```
struct Person {
    std::string name;
    int age;
    double height;
};
```

上述示例定义了一个名为 Person 的结构体，包含三个成员变量：name（字符串类型）、age（整数类型）和 height（浮点数类型）。

1.2.2 class 类

类是一种更高级的自定义数据类型，它允许定义数据成员和成员函数，并将它们封装在一个单独的实体中。类可以用于实现面向对象编程的概念，如封装、继承和多态。
类的定义使用关键字 class，并通过花括号括起来列出成员变量和成员函数。

```
class Circle {
private:
    double radius;

public:
    Circle(double r) {
        radius = r;
    }

    double getArea() {
        return 3.14159 * radius * radius;
    }
};
```

上述示例定义了一个名为 Circle 的类，包含一个私有成员变量 radius 和两个公有成员函数：构造函数和计算面积的函数 getArea()。
类提供了更强大的封装能力，可以通过访问修饰符（如 private、public 和 protected）来控制成员的访问权限。此外，类还支持继承和多态等面向对象编程的特性。
通过结构体和类，可以根据具体的需求定义自己的数据类型，并在程序中使用它们来组织和处理数据。自定义数据类型可以提高代码的可读性、可维护性和重用性。

2　变量类型

3　变量作用域

一般来说有三个地方可以定义变量：
在函数或一个代码块内部声明的变量，称为局部变量。
在函数参数的定义中声明的变量，称为形式参数。
在所有函数外部声明的变量，称为全局变量。
作用域是程序的一个区域，变量的作用域可以分为以下几种：
局部作用域：在函数内部声明的变量具有局部作用域，它们只能在函数内部访问。局部变量在函数每次被调用时被创建，在函数执行完后被销毁。
全局作用域：在所有函数和代码块之外声明的变量具有全局作用域，它们可以被程序中的任何函数访问。全局变量在程序开始时被创建，在程序结束时被销毁。
块作用域：在代码块内部声明的变量具有块作用域，它们只能在代码块内部访问。块作用域变量在代码块每次被执行时被创建，在代码块执行完后被销毁。
类作用域：在类内部声明的变量具有类作用域，它们可以被类的所有成员函数访问。类作用域变量的生命周期与类的生命周期相同。

4　局部变量

在函数或一个代码块内部声明的变量，称为局部变量。它们只能被函数内部或者代码块内部的语句使用。下面的实例使用了局部变量：

```
int sum(){
        int a,b,sum; // 局部变量声明
        a = 1,b = 2; // 实际初始化
        sum = a + b;
        cout<<"sum␣=␣"<<sum<<endl;
        return 0;
```

```
}
```

5 全局变量

局部变量和全局变量的名称可以相同，但是在函数内，局部变量的值会覆盖全局变量的值。下面是一个实例：

```
// globalVariable 全局变量
int i = 3;
int globalVariable(){
        int i = 5;
        cout<<"i␣=␣"<<i<<endl;
        return 0;
}
```

6 块作用域

块作用域指的是在代码块内部声明的变量：

```
// blockScope块作用域
int blockScope(){
        int i = 1;
        {
                int i = 2; // 块作用域变量
                cout<<"i␣=␣"<<i<<endl;
        }
        cout<<"i␣=␣"<<i<<endl;
        return 0;
}
```

7 代码介绍

```
#include <iostream>
using namespace std;
int main() {
        cout<<"hello"<<endl;
```

```
    return 0;
}
```

C++头文件<iostream>
using namespace std; 告诉编译器使用 std 命名空间。
int main() 是主函数，程序从这里开始执行。
cout«"Hello World"; 会在屏幕上显示消息 "Hello World"。
下一行 return 0; 终止 main( ) 函数，并向调用进程返回值 0。
在 C++ 中，分号是语句结束符。也就是说，每个语句必须以分号结束。它表明一个逻辑实体的结束。

8　标识符

9　关键字

Chapter

基本数据类型

Chapter

基本运算

Chapter

数学库常用函数

Chapter

结构化程序设计

Chapter

数组

Chapter

字符串的处理

Chapter

函数与递归

Chapter

结构体与联合体

Chapter

指针类型

Chapter

文件及基本读写

1 fstream

2 ifstream

3 ifstream

Chapter

std

在 C++中，”std” 和”STL” 都是与标准库（Standard Library）相关的概念，但它们具有不同的含义和范围。

1. STL（Standard Template Library）：STL 是 C++标准库中的一部分，它是一组模板类和函数的集合，提供了丰富的数据结构和算法实现。STL 包括容器（如 vector、list、map 等）、迭代器、算法（如排序、搜索、转换等）、函数对象等。STL 的设计理念是基于泛型编程，它通过模板技术使算法和数据结构能够独立于特定类型工作，提供了高度可复用和可扩展的组件。

2. std（Standard Namespace）：std 是 C++标准库中的命名空间（namespace），其中包含了大量的类、函数和常量。std 命名空间用于将 C++标准库中的所有标识符（如容器、算法、输入输出等）进行组织和隔离，以避免命名冲突。使用 std 命名空间，我们可以通过前缀”std::” 访问标准库中的各种成员。

总结起来，STL 是 C++标准库的一个子集，包含了模板类和函数，提供了通用的数据结构和算法。而”std” 是 C++标准库的命名空间，用于组织和隔离标准库中的各个成员。STL 是 std 命名空间的一部分，我们可以使用”std::” 前缀来访问 STL 中的各种组件。

例如，使用 STL 中的 vector 容器和 sort 算法，我们可以这样引用：

```
#include <vector>    // 包含STL中的vector容器
#include <algorithm>  // 包含STL中的sort算法

int main() {
    std::vector<int> nums = {4, 2, 6, 1, 3};  // 使用STL中的vector容器
    std::sort(nums.begin(), nums.end());//使用STL中的sort算法对nums进行排序

    return 0;
}
```

在上述示例中，我们使用了 STL 中的 vector 容器和 sort 算法，通过 std 命名空间访问这些

组件。

Chapter

stl

Chapter

类

Chapter

面向对象

1 算术运算符

```
// 算术运算符
int arithmeticOperator(){
        int a = 5;
        int b = 3;
        int c;
        cout<<"a = "<<a<<endl;
        cout<<"b = "<<b<<endl;
        c = a + b;
        cout<<"c = a + b = "<<c<<endl;
        c = a - b;
        cout<<"c = a - b = "<<c<<endl;
        c = a * b;
        cout<<"c = a * b = "<<c<<endl;
        c = a / b;
        cout<<"c = a / b = "<<c<<endl;
        c = a % b;
        cout<<"c = a % b = "<<c<<endl;
        int d = 7;
        cout<<"d = "<<d<<endl;
        c = d++;
        cout<<"c = d++ = "<<c<<endl;
        c = d--;
        cout<<"c = d-- = "<<c<<endl;
        return 0;
}
```

2　关系运算符

```
// 关系运算符
int relationalOperator(){
        int a = 5;
        int b = 3;
        cout<<"a␣=␣"<<a<<endl;
        cout<<"b␣=␣"<<b<<endl;
        int c;
        if (a == b){
                cout<<"a␣等于␣b"<<endl;
        }else{
                cout<<"a␣不等于␣b"<<endl;
        }
        if(a < b){
                cout<<"a␣小于␣b"<<endl;
        }else{
                cout<<"a␣不小于␣b"<<endl;
        }
        if(a > b){
                cout<<"a␣大于␣b"<<endl;
        }else{
                cout<<"a␣不大于␣b"<<endl;
        }
        return 0;
}
```

3　逻辑运算符

逻辑运算符在C++中用于解决以下问题：
1. 条件判断：逻辑运算符允许程序员在条件语句中对多个条件进行组合判断。通过使用逻辑与运算符（&&）和逻辑或运算符（||），可以根据多个条件的组合结果来确定程序的执行路径。
2. 循环控制：逻辑运算符在循环语句中起到关键作用，例如在while循环或do-while循环中，使用逻辑运算符可以设置多个条件来控制循环的执行和终止条件。
3. 布尔逻辑操作：逻辑运算符允许对布尔值进行操作，将多个布尔值进行组合，从而得到新的布尔值。这对于程序中的条件逻辑判断非常有用。

通过使用逻辑运算符，程序员可以根据条件的组合结果来进行复杂的判断和控制，从而实现程序的逻辑流程控制和条件判断。这样可以使程序更加灵活和可控，并能够处理多种不同的情况。

C++中的逻辑运算符用于对条件表达式进行逻辑运算，通常返回布尔值（true 或 false）。以下是 C++中常用的逻辑运算符：

1. 逻辑与运算符（&&）：当且仅当两个操作数都为 true 时，结果为 true。否则，结果为 false。

```
bool a = true;
bool b = false;
bool result = a && b; // 结果为false
```

2. 逻辑或运算符（||）：当至少有一个操作数为 true 时，结果为 true。只有当两个操作数都为 false 时，结果为 false。

```
bool a = true;
bool b = false;
bool result = a || b; // 结果为true
```

3. 逻辑非运算符（!）：对操作数进行取反操作，如果操作数为 true，则结果为 false；如果操作数为 false，则结果为 true。

```
bool a = true;
bool result = !a; // 结果为false
```

逻辑运算符通常与条件语句（例如 if 语句和 while 循环）一起使用，用于控制程序的执行流程和判断条件的满足情况。

C++ 中的逻辑运算符包括逻辑与（&&）、逻辑或（||）、逻辑非（!）三种。它们的作用是对逻辑表达式进行求值，以判断表达式的真假。

当使用逻辑与（&&）时，只有当两个操作数都为真（非零）时，整个表达式才为真，否则为假。因此，如果一个操作数为真，另一个操作数为假，整个表达式的结果就是假。

同样的道理，当使用逻辑或（||）时，只有当两个操作数都为假（零）时，整个表达式才为假，否则为真。如果一个操作数为假，另一个操作数为真，整个表达式的结果也是真。

逻辑非（!）则是将操作数的真假值取反。如果操作数为真，取反后就是假；如果操作数为假，取反后就是真。

因此，当使用逻辑运算符时，需要注意操作数的真假值，以便正确地求出整个表达式的值。

```
// 逻辑运算符
int logicalOperator(){
```

```
    int a = 3,b = 5,c;
    cout<<"a␣=␣"<<a<<endl;
    cout<<"b␣=␣"<<b<<endl;
    if (a&&b){
        cout<<"a&&b条件为␣true"<<endl;
    }
    if (a || b){
        cout<<"a||b条件为␣true"<<endl;
    }
    // 改变a和b的值
    a = 0;
    b = 5;
    if (a && b){
        cout<<"a&&b条件为␣true"<<endl;
    }else{
        cout<<"a&&b条件为␣false"<<endl;
    }
    if (!(a&&b)){
        cout<<"!(a&&b)条件为␣true"<<endl;
    }
    return 0;
}
```

Chapter

注释

1 单行多行注释

C++ 支持单行注释和多行注释。注释中的所有字符会被 C++ 编译器忽略。
// - 一般用于单行注释。
/* ... */ - 一般用于多行注释。

Chapter

常量

1 #define 预处理器

使用 #define 预处理器定义常量

```
// #define 预处理器定义常量
#define LENGTH 3
#define WIDTH 2

int areaDefine(){
        int area;
        area = LENGTH * WIDTH;
        cout<<"area␣=␣"<<area<<endl;
        return 0;
}
```

2 const 关键字

```
// 使用 const 前缀声明指定类型的常量
int constConstant(){
        const int LENGTH_ = 3;
        const int WIDTH_ = 2;
        int area;
        area = LENGTH_ * WIDTH_;
        cout<<"area␣=␣"<<area<<endl;
        return 0;
}
```

Chapter

修饰符类型

```
int modifier(){
        short int i; // 有符号短整数
        short unsigned int j;
        j = 50000;
        i = j;
        cout<<"j␣=␣"<<j<<endl;
        cout<<"i␣=␣"<<i<<endl;
        return 0;
}
```

Chapter

函数

1　CSP-J（普及组）2022 年 T1 乘方 (pow)

Chapter

数组

Chapter

STL

Chapter

SLT

1 vector 向量

1.1 stl_vector.h

C:\Program Files\CodeBlocks\MinGW\lib\gcc\x86_64-w64-mingw32\8.1.0\include\c++\bits\stl_vector.h

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    // 创建一个整数向量
    vector<int> numbers;

    // 向向量中添加元素
    numbers.push_back(1);
    numbers.push_back(2);
    numbers.push_back(3);

    // 访问和修改向量中的元素
    cout << "First element: " << numbers[0] << endl;
    cout << "Second element: " << numbers[1] << endl;

    numbers[2] = 4;
```

```
    // 遍历向量中的元素
    cout << "All elements: ";
    for (int i = 0; i < numbers.size(); ++i) {
        cout << numbers[i] << " ";
    }
    cout << endl;

    // 使用迭代器遍历向量中的元素
    cout << "All elements (using iterator): ";
    for (vector<int>::iterator it = numbers.begin(); it != numbers.end(); +
        cout << *it << " ";
    }
    cout << endl;

    // 删除向量中的最后一个元素
    numbers.pop_back();

    // 检查向量是否为空
    if (numbers.empty()) {
        cout << "Vector is empty" << endl;
    } else {
        cout << "Vector is not empty" << endl;
    }

    // 清空向量
    numbers.clear();

    // 检查向量是否为空
    if (numbers.empty()) {
        cout << "Vector is empty" << endl;
    } else {
        cout << "Vector is not empty" << endl;
    }

    return 0;
}
```

Chapter

class 类

在 C++中，‘class‘关键字用于定义一个类。类是一种用户自定义的数据类型，用于封装数据和操作。类可以包含成员变量（属性）和成员函数（方法），用于描述对象的状态和行为。

以下是一个简单的 C++类的示例：

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

class Rectangle {
private:
    int length;
    int width;

public:
    // 构造函数
    Rectangle(int l, int w) : length(l), width(w) {}

    // 成员函数
    int getArea() {
        return length * width;
    }

    // 成员函数
    int getPerimeter() {
        return 2 * (length + width);
    }
};
```

```
int main() {
    // 创建一个 Rectangle 对象
    Rectangle rect(5, 3);

    // 调用对象的成员函数
    int area = rect.getArea();
    int perimeter = rect.getPerimeter();

    // 打印结果
    cout << "Area: " << area << endl;
    cout << "Perimeter: " << perimeter << endl;

    return 0;
}
```

在上述示例中，定义了一个名为 'Rectangle'的类，它具有私有成员变量 'length'和 'width'，以及公有成员函数 'getArea()'和 'getPerimeter()'。构造函数用于初始化对象的数据成员。在 'main()'函数中，创建了一个 'Rectangle'对象 'rect'，并使用对象的成员函数 'getArea()'和 'getPerimeter()'计算矩形的面积和周长。最后，使用 'std::cout'打印结果。类提供了一种组织和封装相关数据和行为的方式，使代码更加模块化和可维护。通过类的实例化，可以创建多个对象，每个对象都有自己的数据和方法。除了成员变量和成员函数，类还可以包含访问修饰符（如 'public'、'private'、'protected'）和其他特性（如继承、多态）等。

1 构造函数

在 C++中，构造函数是一种特殊的成员函数，用于在创建对象时进行初始化操作。构造函数的名称与类的名称相同，并且没有返回类型（包括 'void'）。它可以有参数，也可以没有参数。构造函数在以下几种情况下会被自动调用：

1. 在创建对象时，使用 'new'运算符动态分配内存时。
2. 在声明对象时，使用类的默认构造函数进行初始化。
3. 在将一个对象作为参数传递给函数时，调用拷贝构造函数进行复制。

以下是一个简单的示例，展示了如何定义和使用构造函数：

```
#include <iostream>

class MyClass {
private:
```

```
    int value;

public:
    // 默认构造函数
    MyClass() {
        value = 0;
        std::cout << "Default␣constructor␣called" << std::endl;
    }

    // 带参数的构造函数
    MyClass(int val) {
        value = val;
        std::cout << "Parameterized␣constructor␣called" << std::endl;
    }

    // 成员函数
    int getValue() {
        return value;
    }
};

int main() {
    // 使用默认构造函数创建对象
    MyClass obj1;
    std::cout << "Value:␣" << obj1.getValue() << std::endl;

    // 使用带参数的构造函数创建对象
    MyClass obj2(10);
    std::cout << "Value:␣" << obj2.getValue() << std::endl;

    return 0;
}
```

在上述示例中，定义了一个名为‘MyClass‘的类，其中包含一个私有成员变量‘value‘和三个构造函数。默认构造函数用于初始化‘value‘为0，带参数的构造函数用于将传入的值赋给‘value‘。在‘main()‘函数中，首先使用默认构造函数创建了一个‘MyClass‘对象‘obj1‘，并通过‘getValue()‘方法获取对象的值并打印。然后，使用带参数的构造函数创建了另一个对象‘obj2‘，同样获取并

打印了对象的值。
构造函数在对象创建时自动调用，用于进行必要的初始化工作。你可以根据需要定义不同的构造函数，以支持不同的初始化方式。
希望这个示例对你有帮助！如果你还有其他问题，请随时提问。

2 拷贝构造函数

在 C++中，拷贝构造函数（Copy Constructor）是一种特殊的构造函数，用于创建一个对象的副本。拷贝构造函数通常以传入对象的引用作为参数，并使用该对象的数据来初始化新对象。
拷贝构造函数在以下情况下会被自动调用：
1. 在将一个对象作为参数传递给函数时，进行参数的复制。
2. 在使用一个对象初始化另一个对象时，进行对象的复制。
3. 在函数返回一个对象时，进行对象的复制。
以下是一个简单的示例，展示了如何定义和使用拷贝构造函数：

```
#include <iostream>

class MyClass {
private:
    int value;

public:
    // 默认构造函数
    MyClass() {
        value = 0;
        std::cout << "Default␣constructor␣called" << std::endl;
    }

    // 带参数的构造函数
    MyClass(int val) {
        value = val;
        std::cout << "Parameterized␣constructor␣called" << std::endl;
    }

    // 拷贝构造函数
    MyClass(const MyClass& other) {
```

```
        value = other.value;
        std::cout << "Copy constructor called" << std::endl;
    }

    // 成员函数
    int getValue() {
        return value;
    }
};

void printObject(const MyClass& obj) {
    std::cout << "Object value: " << obj.getValue() << std::endl;
}

int main() {
    // 使用默认构造函数创建对象
    MyClass obj1;
    std::cout << "Value: " << obj1.getValue() << std::endl;

    // 使用带参数的构造函数创建对象
    MyClass obj2(10);
    std::cout << "Value: " << obj2.getValue() << std::endl;

    // 使用拷贝构造函数创建对象的副本
    MyClass obj3 = obj2;
    std::cout << "Value: " << obj3.getValue() << std::endl;

    // 作为函数参数传递对象
    printObject(obj3);

    return 0;
}
```

在上述示例中，‘MyClass‘类定义了默认构造函数、带参数的构造函数和拷贝构造函数。拷贝构造函数以传入对象的引用作为参数，并将传入对象的值复制给新对象的成员变量。
在 ‘main()‘函数中，首先使用默认构造函数创建了一个 ‘MyClass‘对象 ‘obj1‘，然后使用带参数的构造函数创建了另一个对象 ‘obj2‘。接下来，使用拷贝构造函数将 ‘obj2‘复制到新对象

‘obj3‘。最后，通过调用 ‘printObject()‘函数将 ‘obj3‘作为参数传递给函数。
拷贝构造函数在对象的复制过程中起到重要作用，确保新对象与原始对象具有相同的值。如果没有显式定义拷贝构造函数，编译器会自动生成一个默认的拷贝构造函数。
需要注意的是，拷贝构造函数的参数通常是 ‘const‘引用，以防止在拷