# 程序设计B with 吕春利

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

## 一、面向对象编程

### 1. 类

```
class 类名
{
    public:
        // 公有成员
    private:
        // 私有成员
    protected:
        // 受保护的成员
}; // 注意结尾的分号！
```

私有成员只能在类内部访问。

### 2. 函数重载（多态）

#### 缺省函数

```
void func(int a, int b = 2, int c = 3)
{
    // 函数体
}
```

相当于python中的默认值。注意默认值应当从右往左设置。

#### 运算符重载

```
int operator 运算符(参数列表)
{
    // 函数体
}
```

例如，对复数类`Cmycomplex`进行运算符重载。在`public`成员中定义以下函数：

```
Cmycomplex operator+(Cmycomplex& adder)
{
    // 返回值是一个对象（称作临时变量，用户没办法再修改了）
    return Add(adder);
}
Cmycomplex operator-(Cmycomplex& other)
{
    return Cmycomplex(real - other.real, imag - other.imag);
```

```
}
Cmycomplex operator*(Cmycomplex& other)
{
    double new_real = real * other.real - imag * other.imag;
    double new_imag = real * other.imag + imag * other.real;
    return Cmycomplex(new_real, new_imag);
}
Cmycomplex operator/(Cmycomplex& other)
{
    double new_real = (real * other.real + imag * other.imag) / (other.real * other.real + other.imag *
other.imag);
    double new_imag = (imag * other.real - real * other.imag) / (other.real * other.real + other.imag *
other.imag);
    return Cmycomplex(new_real, new_imag);
}
bool operator==(Cmycomplex other)
{
    return (real - other.real == 0) && (imag - other.imag == 0);
}
bool operator!=(Cmycomplex other)
{
    return (real - other.real != 0) || (imag - other.imag != 0);
}
```

据说，推荐的做法是在`class`定义中只声明函数的原型。在类外实现函数。例如：

```
class Cmycomplex
{
    public:
        // 只进行函数原型的声明，不定义函数体
        Cmycomplex operator+(Cmycomplex& other);
};

// 由于是在类外对类内函数进行编辑，需要限定名字空间
Cmycomplex Cmycomplex::operator+(Cmycomplex& other)
{
    return Cmycomplex(real + other.real, imag + other.imag);
}
```

## 输入输出的重载

- `<<`和`>>`本身是位运算符（左移运算符和右移运算符）。经重载后可以用来输入输出对象。

```
// 类内：定义友元函数
friend istream& operator>>(istream& is, Cmycomplex& x);
friend ostream& operator<<(ostream& os, const Cmycomplex& x);
// 两个函数都不是 const，毕竟都不是类内的成员函数，只是朋友罢了，没有 this 指针，不会修改内容；
// 类外
istream& operator>>(istream& is, Cmycomplex& x)
{
    is >> x.real >> x.imag;
```

```
    return is;
}

ostream& operator<<(ostream& os, const Cmycomplex& x) // 第二个参数必须是引用，否则会创建临时对象出现麻烦
{
    os << "(" << x.real << ((x.imag > 0) ? "+" : "") << x.imag << "i)";
    return os;
}
```

### 双目运算符

## 3. 构造器

### 构造器

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

        int getValue() const
        {
            return val;
        }
    int val;
};

int main()
{
    MyClass obj(10);
    // obj.getValue() = 10
}
```

此处还涉及到了**常成员函数**。在成员函数名称后加`const`，能保证参数在函数体内不被修改。

### 构造器的初始化列表

```
class MyClass {
    public:
        // 使用初始化列表的构造器
        MyClass(int val) : value(val)
        {
            // 构造器函数体
        }

    int value;
};
```

```
    public:
        // 带有初始化列表的构造器，初始化多个成员变量
        Person(const std::string& n, int a, double h, bool s)
            : name(n), age(a), height(h), isStudent(s)
        {
            // 构造器函数体
        }
```

### 构造函数的重载

- 特点
    - 没有返回值
    - 函数名必须与类名保持一致

## 4. 友元函数

- 通过在类内定义友元函数，可以实现在类外访问类的私有成员。

```
class Cmycomplex
{
    private:
        int real, imag;
    public:
        ...
        friend Cmycomplex operator+(int a, const Cmycomplex& b);
};

Cmycomplex operator+(int a, const Cmycomplex& b)
{
    return Cmycomplex(a + b.real, b.imag);
}
```

- 友元函数不是类的成员函数，不具有`this`指针，也不可以继承，传递等。
- 在进行运算符重载时，友元函数可以使运算符重载更加自然。

---

# 二、引用

- 同样的变量名指向了同一个地址空间

```
int main()
{
    int a = 0;
    int& b = a;
    int& c = b;
    // a, b, c 共用同一个地址空间
    // 改变 a, b, c 中的任意一个，三者都会被波及到
    // 相当于是 b 和 c 是 a 的“别名”
}
```

- 使用**引用**实现的交换函数，使得形参和实参共用一个地址空间.

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

int main()
{
    int a = 1, b = 2;
    swap(a, b);
    cout << a << b << endl;
}
```

- C.f. 用**指针**实现的交换函数

```
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main()
{
    int a = 1, b = 2;
    swap(&a, &b); // 注意要取地址！
    cout << a << b << endl;
}
```

# 三、`const`类型修饰符

在C++中，`const`关键字用于定义常量、限制变量或对象的修改，并在多种上下文中提供类型安全性。以下是`const`关键字的详细内容：

---

## 1. 基本用法：定义常量

- 声明变量为常量，不可修改。

```
const int a = 5;
// a = 10; // 错误：常量不可修改
```

---

## 2. 指针与const

- **指向常量的指针**（底层const）：指针指向的值不可变，但指针本身可以指向其他地址。

```
const int* ptr = &a;
// *ptr = 10; // 错误：不能通过ptr修改值
int b = 20;
ptr = &b;     // 正确：指针本身可变
```

- **常量指针**（顶层const）：指针本身不可变，但指向的值可变。

```
int* const ptr = &b;
*ptr = 30;    // 正确：可以修改指向的值
// ptr = &a;  // 错误：指针本身不可变
```

- **指向常量的常量指针**：指针和指向的值均不可变。

```
const int* const ptr = &a;
// *ptr = 10;  // 错误
// ptr = &b;  // 错误
```

---

## 3. 函数参数中的const

- 保护参数不被修改，常用于引用或指针参数。

```
void print(const std::string& str) {
    // str[0] = 'A'; // 错误：str是常量引用
    std::cout << str;
}
```

---

## 4. 成员函数后的const

- 表示该函数不会修改对象状态，可被`const`对象调用。

```
class MyClass {
    int value;
public:
    int getValue() const {
        // value = 10; // 错误：const成员函数不能修改成员变量
        return value;
    }
};

const MyClass obj;
obj.getValue(); // 正确：调用const成员函数
```

- **mutable成员**：允许在`const`成员函数中修改。

```
class MyClass {
    mutable int counter;
public:
    void increment() const { counter++; } // 合法
};
```

---

## 5. 类中的const成员变量

- 必须在构造函数初始化列表中初始化。

```
class MyClass {
    const int a;
public:
    MyClass(int val) : a(val) {} // 正确：初始化列表
    // MyClass(int val) { a = val; } // 错误：不能在函数体内赋值
};
```

---

## 6. const与返回值

- 返回常量值，防止返回值被意外修改。

```
const int getValue() { return 42; }
// getValue() = 10; // 错误：返回值是const
```

---

## 7. constexpr与const的区别

- const：运行时常量或编译时常量。
- constexpr（C++11）：强制编译时常量，用于优化和模板元编程。

```
constexpr int square(int x) { return x * x; }
int arr[square(5)]; // 合法：编译时计算
```

---

## 8. 函数重载与const

- `const`成员函数与非`const`成员函数可重载。

```
class MyClass {
public:
    void func() { /* 修改对象 */ }
    void func() const { /* 只读操作 */ }
};

MyClass obj;
obj.func();         // 调用非const版本
const MyClass cobj;
cobj.func();        // 调用const版本
```

## 9. const_cast运算符

- 去除`const`属性，但需谨慎使用。

```
const int a = 5;
int* p = const_cast<int*>(&a);
*p = 10; // 未定义行为！原对象是const的。
```

## 10. 其他注意事项

- **字符串字面量**：类型为`const char[]`，不可修改。

  ```
  const char* str = "Hello";
  // str[0] = 'h'; // 错误
  ```

- **类型别名中的const**：

  ```
  typedef char* pchar;
  const pchar ptr = nullptr; // 等价于 char* const ptr
  ```

## 总结

- `const`提供编译时的类型安全性，防止意外修改。
- 灵活应用于变量、指针、函数参数、成员函数等场景。
- 结合`constexpr`、`mutable`和`const_cast`等特性，可精确控制代码行为。

理解并合理使用`const`能显著提升代码的健壮性和可维护性。

## 重要：练习中涉及到的`const`关键字的使用错误

### 1. `const`成员函数里修改成员变量

```
int getMysize() const { TimesofGetMysize++; return Mysize; }
```

#### 报错原因

在 C++ 里，`const`成员函数是不能修改对象的成员变量的。`getMysize`被声明成`const`成员函数，然而它却修改了`TimesofGetMysize`，这就违背了`const`成员函数的规则。

#### 知识点讲解

- **`const`成员函数的作用**：`const`成员函数的作用是保证在调用该函数时，对象的状态不会被改变。这在对象是常量对象时尤为重要，因为常量对象只能调用`const`成员函数。
- **`mutable`关键字的使用**：若想在`const`成员函数里修改某个成员变量，可使用`mutable`关键字修饰该成员变量。像下面这样修改`Point`类：

```
class Point {
    // ...
private:
    int x, y;
    mutable int TimesofGetMysize; // 使用 mutable 关键字
    const int Mysize;
};
```

经过这样的修改，`getMysize`函数就能在`const`成员函数里修改`TimesofGetMysize`了。

## 2. 对`const`对象的成员变量进行修改

```
float dist( const Point &p1, const Point &p2) {
    double x = p1.x – p2.x;
    double y = p1.y – p2.y;
    p2.x++; // 错误：试图修改 const 对象的成员
    return static_cast<float>(sqrt(x*x+y*y));
}
```

**报错原因**

`p2`是`const`引用，也就意味着不能修改`p2`的成员变量。但代码里却尝试修改`p2.x`，这会引发编译错误。

**知识点讲解**

- **`const`引用的特性**：当一个对象以`const`引用的形式传递时，在函数内部不能修改该对象的成员变量。这是为了确保在函数调用期间对象的状态不会被改变。
- **解决办法**：要是不需要修改对象的成员变量，那就保持函数参数为`const`引用；若需要修改，就把参数改为非`const`引用。不过要注意，若传入的是常量对象，就只能使用`const`引用。

## 3. 常量对象调用非`const`成员函数

在`main`函数里，若尝试让常量对象调用非`const`成员函数，就会产生编译错误。例如：

```
const Point myp1(1, 1);
myp1.getX(); // 错误：常量对象不能调用非 const 成员函数
```

**报错原因**

常量对象只能调用`const`成员函数，因为非`const`成员函数可能会修改对象的状态，这与常量对象的特性相冲突。

**知识点讲解**

- **常量对象的限制**：常量对象在其生命周期内状态不能被改变，所以只能调用那些不会修改对象状态的`const`成员函数。
- **解决办法**：若某个成员函数不会修改对象的状态，就把它声明为`const`成员函数。像下面这样修改`getX`和`getY`函数：

```
class Point {
public:
    int getX() const { return x; }
    int getY() const { return y; }
    // ...
};
```

### 4. 函数重载和`const`参数

```
friend float dist(const Point &p1,const Point &p2);
friend float dist( Point &p1, const Point &p2);
```

**知识点讲解**

- **函数重载的规则**：函数重载是指在同一个作用域内，有多个同名函数，但它们的参数列表不同（参数个数、类型或者顺序不同）。这里的两个`dist`函数构成了重载，一个接受两个`const`引用参数，另一个接受一个非`const`引用和一个`const`引用参数。
- **调用规则**：在调用`dist`函数时，编译器会依据传入参数的类型来决定调用哪个重载函数。若传入的是常量对象，就会调用接受`const`引用参数的函数；若传入的是非常量对象，编译器会优先选择接受非`const`引用参数的函数。

# `this`指针

## 1. 基本概念

- `this` 指针是一个隐含于每一个非静态成员函数中的特殊指针。它指向调用该成员函数的那个对象。
- 当你调用一个对象的成员函数时，编译器会自动将该对象的地址作为隐藏参数传递给成员函数，这个地址就存储在 `this` 指针中。

## 2. `this` 指针的用途

### 2.1 区分成员变量和局部变量

当成员函数的参数名与成员变量名相同时，使用 `this` 指针可以明确指定访问的是成员变量。

```
// 构造函数
Rectangle(int width, int height) {
    this->width = width;
    this->height = height;
}
```

### 2.2 返回对象本身

在成员函数中，可以使用 `*this` 返回对象本身，从而实现链式调用。

```
class Number {
private:
    int value;
```

```
public:
    Number(int val) : value(val) {}
    Number& add(int num) {
        this->value += num;
        return *this;
    }
};

int main() {
    Number num(5);
    num.add(3).add(2);
}
```

在 `add` 函数中，返回 `*this`，这样就可以连续调用 `add` 函数，实现链式操作。

## 3. `this` 指针的特点

### 3.1 类型

在非 `const` 成员函数中，`this` 指针的类型是 `类名*`；
在 `const` 成员函数中，`this` 指针的类型是 `const 类名*`。

### 3.2 不能被显式赋值

`this` 指针是由编译器自动管理的，不能被显式赋值。

## 4. `this` 指针在静态成员函数中的情况

静态成员函数不依赖于任何对象，它属于整个类，因此静态成员函数中没有 `this` 指针。如果在静态成员函数中尝试使用 `this` 指针，编译器会报错。

```
#include <iostream>
class MyClass {
public:
    static void staticFunction() {
        // 错误：静态成员函数中没有 this 指针
        std::cout << this << std::endl;
    }
};

int main() {
    MyClass::staticFunction();
    return 0;
}
```

# 单目运算符

## 1. 常见的内置单目运算符

### 算术运算符

- **正号** `+`：一般用来表明数值的正性，在实际运用时常常可以省略。例如：`int a = +5;` 这种写法和 `int a = 5;` 是等价的。
- **负号** `-`：把操作数的符号进行取反操作。例如：`int b = -a;` 若 `a` 的值是 5，那么 `b` 的值就为 -5。

### 自增和自减运算符

- **前置自增** `++`：先把变量的值增加 1，然后返回增加之后的值。

```
int num = 5;
int result = ++num;
// num 的值变为 6，result 的值也是 6
```

- **后置自增** `++`：先返回变量当前的值，之后再把变量的值增加 1。

```
int num = 5;
int result = num++;
// result 的值为 5，num 的值变为 6
```

- **前置自减** `--`：先把变量的值减少 1，然后返回减少之后的值。
- **后置自减** `--`：先返回变量当前的值，之后再把变量的值减少 1。

### 逻辑非运算符 `!`

对布尔值进行取反操作。要是操作数为 `true`，那么结果就是 `false`；要是操作数为 `false`，那么结果就是 `true`。

```
bool flag = true;
bool result = !flag;
// result 的值为 false
```

### 按位取反运算符 `~`

对整数的二进制位执行取反操作。

```
int num = 5;
// 二进制表示为 00000000 00000000 00000000 00000101
int result = ~num;
// 二进制表示为 11111111 11111111 11111111 11111010
```

### 地址运算符 `&`

用于获取变量的内存地址。

```
int num = 5;
int* ptr = &num;
// ptr 指向 num 的内存地址
```

### 解引用运算符 `*`

用于访问指针所指向的变量的值。

```
int num = 5;
int* ptr = &num;
int value = *ptr;
// value 的值为 5
```

## 2. 单目运算符的重载

### 前置自增运算符 `++` 的重载

先自增，后返回函数本身。
重载时返回类型通常是对象的引用，这样就能支持链式操作。

```
class Counter {
    private:
        int count;
    public:
        // 构造函数
        Counter(int c = 0) : count(c) {}

        // 前置自增运算符重载
        Counter& operator++() { // 注意返回值类型！
            ++count;
            return *this; // 注意返回值！
        }
};
```

注意重载函数的类型是`Counter&`（引用）。

### 后置运算符 `++` 的重载

`++` 先返回对象的当前状态，然后再对对象进行自增操作。

后置运算符重载时返回的是临时对象，**不是**引用。

为了和前置自增运算符区分开来，后置自增运算符重载函数需要一个额外的 `int` 类型参数（该参数仅用于区分，在函数内部不会使用）。

```
#include <iostream>

class Counter {
```

```
private:
    int count;
public:
    // 构造函数
    Counter(int c = 0) : count(c) {}

    // 前置自增运算符重载
    Counter& operator++() {
        ++count;
        return *this;
    }

    // 后置自增运算符重载
    Counter operator++(int) {
        Counter temp = *this;
        ++(*this);
        return temp;
    }

    // 获取计数器的值
    int getCount() const {
        return count;
    }
};
```

### 代码解释

- `Counter operator++(int)`：`int` 参数仅用于区分前置和后置自增，在函数内部不会使用。函数首先创建一个临时对象 `temp` 来保存对象的当前状态，然后对对象本身进行自增操作，最后返回临时对象 `temp`。
- `main` **函数**：分别演示了前置和后置自增运算符的使用，并输出操作前后对象的值，以验证重载的正确性。

## `++` 运算符重载的注意事项

- **返回类型**：前置自增运算符重载函数返回对象的引用，后置自增运算符重载函数返回对象的副本。
- **区分前置和后置**：后置自增运算符重载函数需要一个额外的 `int` 参数来和前置自增运算符区分。
- `const` **成员函数**：如果运算符重载函数不会修改对象的状态，应该将其声明为 `const` 成员函数。

通过对 `++` 运算符的重载，能够让自定义类类型像内置类型一样使用自增运算符，提升代码的可读性和可维护性。

# 临时变量

## 一、定义与本质

- **临时变量** 是一种 **没有标识符（名称）** 的对象或基本类型变量，由编译器在需要时自动生成，用于临时存储数据。
- 本质上是 **右值（Rvalue）**，只能出现在表达式右侧（如函数返回值、类型转换结果等），不能被直接赋值或取地址（除非用 `const` 引用绑定）。

## 二、核心特点

1. **自动创建与销毁**
   无需程序员手动声明，由编译器根据语法规则自动生成；生命周期结束后自动释放资源（如调用析构函数）。
2. **作用域局限**
   仅存在于当前表达式或函数调用的上下文，无法在外部访问。
3. **右值属性**
   临时变量是右值，不能直接用左值引用（`T&`）绑定，但可以用 **常量左值引用（`const T&`）** 或 **右值引用（`T&&`）** 绑定。

## 三、常见创建场景

### 1. 函数返回非引用类型时

当函数返回一个 **非引用类型的值**（如普通对象、基本类型），编译器会创建临时变量存储返回值，供调用者使用。

```
Point createPoint() {
    return Point(1, 2); // 返回临时 Point 对象
}
// 调用时，返回值为临时变量
Point p = createPoint(); // 临时变量被复制（或移动）到 p
```

### 2. 类型转换时

显式或隐式类型转换会生成临时变量。

- **显式转换**：如 `static_cast<Point>(5)`（将整数转换为 `Point` 对象）。
- **隐式转换**：当函数参数类型不匹配时，编译器自动转换生成临时变量。

  ```
  void func(Point p);
  func(10); // 隐式将整数 10 转换为 Point(10, 0) 临时对象
  ```

### 3. 表达式中的中间结果

复杂表达式的中间计算结果可能生成临时变量。例如：

```
int a = 1 + 2 * 3; // 2*3 生成临时变量 6，再与 1 相加生成临时变量 7，最后赋值给 a
Point p1(1,2), p2(3,4);
float d = dist(p1, p2); // dist 函数返回的结果是临时变量，赋值给 d
```

### 4. 临时对象作为函数实参

直接用构造函数调用作为实参时，会生成临时对象。

```
TestFun(Point(1, 1)); // 直接创建临时 Point 对象作为参数传递给 TestFun
```

### 5. 运算符重载返回对象

当运算符重载函数返回一个对象（而非引用）时，结果为临时变量。

```
Point operator+(const Point& p1, const Point& p2) {
    return Point(p1.x + p2.x, p1.y + p2.y); // 返回临时对象
}
Point p = p1 + p2; // p1+p2 的结果是临时对象，赋值给 p
```

## 四、生命周期与销毁规则

### 1. 基本规则（C++ 标准）

临时变量的生命周期在 **完整表达式（Full Expression）结束时销毁**。完整表达式指的是一个语句、条件表达式、循环表达式等的结束点。
例如：

```
{
    Point p = Point(1, 2); // 临时对象在赋值给 p 后销毁（C++17 后优化为直接构造 p）
    cout << dist(p, Point(3, 4)) << endl; // 临时 Point(3,4) 在表达式结束后销毁
}
```

### 2. 特殊延长规则

- **绑定到 `const` 引用时**：若临时变量被绑定到 `const T&` 或 `volatile T&`，其生命周期延长至引用的作用域结束。

  ```
  const Point& ref = Point(1, 2); // 临时对象的生命周期延长至 ref 作用域结束
  ```

- **作为类成员初始化器时**：临时变量在构造函数完成后销毁。

## 五、作用与意义

1. **支持表达式语法**
   使复杂表达式（如函数调用、类型转换）能够无缝衔接，无需显式声明中间变量。
2. **临时存储中间结果**
   在函数返回值、运算符计算等场景中暂存结果，避免手动管理内存。
3. **隐式类型转换桥梁**
   当参数类型不匹配时，通过临时变量实现隐式转换，增强代码灵活性。

## 六、注意事项与常见错误

### 1. 不能返回局部变量的引用或指针

局部变量在函数结束后销毁，返回其引用或指针会导致 **悬垂引用（Dangling Reference）**。

```
Point& wrongFunc() {
    Point p(1, 2); // 局部变量，函数结束后销毁
    return p; // 错误！返回局部变量的引用，指向已销毁的对象
}
```

### 2. 非 `const` 左值引用不能绑定临时变量

临时变量是右值，只能用 `const T&` 或 `T&&` 绑定，否则编译错误。

```
Point& ref = Point(1, 2); // 错误！非 const 左值引用绑定右值
const Point& constRef = Point(1, 2); // 正确，const 左值引用可绑定右值
Point&& rvalRef = Point(1, 2); // 正确，右值引用可绑定右值
```

### 3. 临时变量的修改无效

临时变量在表达式结束后销毁，修改其值不会影响外部变量。

```
dist(p1, p2); // 若 dist 函数修改了非 const 引用参数（如 p1.x++），则 p1 的值会被改变
dist(const_p1, p2); // 若参数是 const 引用，修改会报错（如之前代码中的错误）
```

### 4. 资源管理问题

若临时变量是包含动态资源（如指针、文件句柄）的对象，需确保其析构函数正确释放资源（通过移动语义或智能指针优化）。

## 总结

临时变量是 C++ 中编译器自动生成的“无名对象”，用于支持表达式计算、类型转换等场景，具有自动创建和销毁、右值属性等特点。合理使用临时变量可以简化代码，但需注意其生命周期和绑定规则（尤其是 `const` 引用和右值引用的使用），避免出现编译错误或运行时问题（如悬垂引用、资源泄漏）。理解临时变量的机制，对掌握 C++ 的拷贝构造、移动语义、性能优化等关键特性至关重要。

# 函数模板

数据类型参数化。

创建通用函数，而不用针对每种数据类型单独编写函数。

## 定义

```
template <typename T>
返回类型 函数名(参数列表) {
    // 函数体
}
```

这里的 `template` 是声明模板的关键字，`typename`（也可以用 `class`）用来指定模板参数，`T` 是模板参数的名称，可代表任意数据类型。返回类型和参数列表里能使用 `T`。

## 使用

使用函数模板时，编译器会依据函数调用时的实参类型来推导模板参数的具体类型，进而生成对应类型的函数实例。

## 示例

下面是一个简单的交换函数模板示例：

```
#include <iostream>

// 定义函数模板
template <typename T>

void swap(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}
```

注意：在每个函数前都需要定义`template`。

## 多模板参数

函数模板也能有多个模板参数，示例如下：

```
#include <iostream>

// 定义带有两个模板参数的函数模板
template <typename T1, typename T2>
void printPair(T1 first, T2 second) {
    std::cout << "First: " << first << ", Second: " << second << std::endl;
}

int main() {
    printPair(10, 3.14);
    printPair("Hello", 20);
    return 0;
}
```

## 结论

编译器并不是把函数模板处理成能够处理任意类型的函数。
编译器从函数模板通过具体类型产生不同的函数。

# 类模板

## 1. 类模板的定义

类模板是一种通用的类，它允许在定义类时使用类型参数。这样，我们就可以用相同的类定义来创建不同类型的对象。

类模板的一般语法形式如下：

```
template <typename T>
class 类名 {
    // 类的成员变量和成员函数定义
    T data;  // 成员变量使用类型参数 T
public:
    类名(T value) : data(value) {}
    // 构造函数，接受类型为 T 的参数
    T getData() const {
        return data;
    }
};
```

在上述代码中，`template <typename T>` 声明了一个模板，`typename T` 表示定义了一个类型参数 `T`。在类 `类名` 中，`data` 成员变量的类型是 `T`，构造函数和 `getData` 成员函数也都与类型 `T` 相关。

## 2. 类模板的实例化

要使用类模板，需要对其进行实例化，即指定具体的类型来替代模板参数。

实例化类模板的语法是：`类名<具体类型> 对象名(构造函数参数);`

例如，对于上述定义的类模板，我们可以这样实例化：

```
类名<int> intObj(10);  // 实例化类模板，将 T 替换为 int 类型
类名<double> doubleObj(3.14);  // 实例化类模板，将 T 替换为 double 类型
```

在这个例子中，`类名<int>` 和 `类名<double>` 分别创建了使用 `int` 类型和 `double` 类型的类模板实例。

## 3. 类模板的成员函数

类模板的成员函数可以在类定义内部实现（如前面的示例），也可以在类定义外部实现。

类外部实现成员函数时，需使用特定的语来表明这是类模板的成员函数。

```
template <typename T>
class MyClass {
    T data;
public:
    MyClass(T value);  // 构造函数声明
    T getData() const;  // 成员函数声明
};

// 在类外实现构造函数
template <typename T>
MyClass<T>::MyClass(T value) : data(value) {}

// 在类外实现成员函数 getData
template <typename T>
T MyClass<T>::getData() const {
    return data;
}
```

1. `template <typename T>` 必须在函数定义的开头再次出现

2. 在类名后面需要使用 `<T>` 来指定模板参数。

## 4. 类模板的默认模板参数

C++ 允许为类模板的类型参数指定默认值。这样，在实例化类模板时，如果不指定类型参数，就会使用默认的类型。

例如：

```
template <typename T = int>  // T 的默认类型为 int
class MyDefaultClass {
    T data;
public:
    MyDefaultClass(T value) : data(value) {}
    T getData() const {
        return data;
    }
};

int main() {
    MyDefaultClass<> obj1(10);  // 使用默认类型 int
    MyDefaultClass<double> obj2(3.14);  // 显式指定类型为 double

    return 0;
}
```

在 `MyDefaultClass` 类模板中，`T` 的默认类型是 `int`。`obj1` 实例化时没有指定类型参数，所以使用默认的 `int` 类型；`obj2` 则显式指定了类型为 `double`。

## 5. 类模板的友元

类模板也可以定义友元函数或友元类。友元函数或友元类可以访问类模板的私有成员。

例如，定义一个友元函数：

```
template <typename T>
class MyFriendClass {
    T data;
public:
    MyFriendClass(T value) : data(value) {}
    friend void printData(const MyFriendClass<T>& obj);  // 声明友元函数
};

template <typename T>
void printData(const MyFriendClass<T>& obj) {
    std::cout << "Data: " << obj.data << std::endl;
}

int main() {
    MyFriendClass<int> obj(10);
    printData(obj);

    return 0;
}
```

在这个例子中，`printData` 函数是 `MyFriendClass` 类模板的友元函数，它可以访问 `MyFriendClass` 对象的私有成员 `data`。

类模板是 C++ 中实现代码复用和泛型编程的重要工具，通过使用类模板，可以编写出更加通用、灵活和可维护的代码。

# 动态数组

普通数组初始化必须定义大小，很不方便！

C++ 规定，下标运算符[ ]必须以成员函数的形式进行重载。该重载函数在类中的声明格式如下：
`返回值类型 & operator[ ] (参数);`

或者：
`const 返回值类型 & operator[ ] (参数) const;`

使用第一种声明方式，[ ]不仅可以访问元素，还可以修改元素。使用第二种声明方式，[ ]只能访问而不能修改元素。在实际开发中，**我们应该同时提供以上两种形式**，这样做是为了适应 const 对象，因为**通过 const 对象只能调用 const 成员函数**，如果不提供第二种形式，那么将无法访问 const 对象的任何元素。

```
template <typename T>
class DynamicArray {
    private:
        T* array;
        unsigned int mallocSize;
    public:
        // Constructor
        DynamicArray(unsigned int length, const T& content);

        // Copy Constructor 拷贝构造器
        //（浅拷贝：编译器自己会生成，但是不能用！原因见下）
        //（深拷贝）
        DynamicArray(const DynamicArray& AnotherDynamicArray);

        // Destructor
        ~DynamicArray();

        unsigned int capacity() const;

        T& operator[](unsigned int i);
        const T& operator[](unsigned int i) const;

        // 赋值重载（编译器自带“浅赋值”，但还是会发生问题）
        DynamicArray& operator=(const DynamicArray<T> & anotherDA) ;

};

template <typename T>
DynamicArray<T>::DynamicArray(unsigned int length, const T& content) {
    mallocSize = length;
    array = new T[mallocSize];
    // new 关键字：申请空间 + 返回指针 + 调用 T 类型的构造函数
    // Literally 很“新”的知识点233
    for (int i = 0; i < this->mallocSize; i++) {
        array[i] = content;
```

```
    }
}

// template <typename T>
// DynamicArray<T>::DynamicArray(const DynamicArray& AnotherDynamicArray) {
//     // 这是浅拷贝：编译器其实会默认生成，我们其实不需要写这个函数
//     // 数据成员指向同一块儿内存，导致最后析构函数会释放同一块内存两次！！程序编译没问题，但运行会出错！！
//     // 对于那些不需要申请空间的类来说，自带的浅拷贝函数就足够了。不涉及指向同一块儿内存的问题
//     this->array = AnotherDynamicArray.array;
//     this->mallocSize = AnotherDynamicArray.mallocSize;
// }


template <typename T>
DynamicArray<T>::DynamicArray(const DynamicArray& AnotherDynamicArray) {
    // 这是深拷贝
    this->mallocSize = AnotherDynamicArray.mallocSize;
    this->array = new T[this->mallocSize];
    for (int i = 0; i < this->mallocSize; i++) {
        this->array[i] = AnotherDynamicArray.array[i];
    }
}


template <typename T>
unsigned int DynamicArray<T>::capacity() const {
    return this->mallocSize;
}

template <typename T>
T& DynamicArray<T>::operator[](unsigned int i) {
    return this->array[i];
}

template <typename T>
const T& DynamicArray<T>::operator[](unsigned int i) const {
    // 和上个函数不同，这个有 const 修饰
    // 就是为了处理赋值的情况，例 int_darray[2] = 3;
    return this->array[i];
}

template <typename T>
DynamicArray& DynamicArray<T>::operator=(const DynamicArray& anotherDA) {
    // 和拷贝函数一样，自带“浅拷贝”，但对于需要 new 资源的情况会因为重复释放内存而报错
    if (this == &x) {
        // 避免出现 a = a 的赋值情况
        return *this;
    }

    delete[] this->array;
    this->mallocSize = x.mallocSize;
    this->array = new T[this->mallocSize];
    for (int i = 0; i < x.mallocSize; i++) {
        array[i] = x.array[i];
    }
```

```
}

template <typename T>
DynamicArray<T>::~DynamicArray() {
    delete[] array;
    // delete / delete[] 关键字：归还空间 + 调用 T 类型的析构函数
    // delete: 删除普通的元素
    // delete[] : 删除数组
}
```

“有借有还，再借不难。”此处，我们引入析构函数。析构函数在函数去世之前调用。

# 把引用想象成“别名”，实际是同一个东西。原先是const，给他取别名也要是const。原先是男生，取别名依然需要是男生。

## 1. 非常量引用（`T&`）不能绑定到常量对象

非常量引用（T&）只能被绑定非常量对象。
常量引用（const T&）可以被绑定到任何对象（常量或非常量）

在C++中，**非常量引用**（如`DynamicArray<T>&`）只能绑定到**非常量对象**，而不能绑定到**常量对象**或**临时对象**。

**示例1：错误的引用绑定**

```
const int x = 10;
int& ref = x;  // 错误！非常量引用不能绑定到常量对象
```

**示例2：你的赋值运算符签名**

```
DynamicArray<T>& operator=(DynamicArray<T>& anotherDA);
```

- 这里的`anotherDA`是**非常量引用**，因此只能接受**非常量对象**。
- 如果传入**常量对象**（如`const DynamicArray<T> obj`），编译器会报错，因为**常量对象不能隐式转换为非常量引用**。

## 2. 为什么需要`const`参数？

当你使用`DynamicArray<DynamicArray<int>> b(10, a);`时，编译器会：

1. 尝试用`a`（类型为`DynamicArray<int>`）初始化`b`中的每个元素（类型为`DynamicArray<int>`）。
2. 这会调用`DynamicArray`的赋值运算符`operator=`。
3. 但由于`b`的元素在初始化时可能被视为**临时常量对象**，传入的`anotherDA`实际上是**常量引用**，因此无法匹配当前的`operator=`签名。

**修正后的签名**：

```
DynamicArray<T>& operator=(const DynamicArray<T>& anotherDA);  // 接受常量引用
```

## 3. 对比两种签名的区别

| 签名 | 接受的参数类型 | 能否修改 anotherDA |
| --- | --- | --- |
| `operator=(X&)` | 只能是非常量对象（如 `X obj`） | 能 |
| `operator=(const X&)` | 常量对象（如 `const X obj`）和非常量对象 | 不能（因为是 `const`） |

## 4. 为什么C++有这样的规则？

这是为了保证**类型安全**：

- 如果允许非常量引用绑定到常量对象，就可以通过引用修改原本不可变的对象，违反了 `const` 的语义。

```
const int x = 10;
int& ref = x;  // 假设允许
ref = 20;      // 试图修改常量对象，矛盾！
```

- 因此，C++禁止这种隐式转换，强制你使用 `const` 引用（如 `const int& ref = x;`）来保证不修改原对象。

## 5. 其他可能的场景

**场景1：临时对象无法绑定到非常量引用**

```
DynamicArray<int> func() { return DynamicArray<int>(); }
b[0] = func();  // 错误！func()返回的临时对象无法绑定到DynamicArray<int>&
```

**场景2：常量对象调用成员函数**

```
const DynamicArray<int> a(10);
b[0] = a;  // 错误！a是常量对象，无法绑定到DynamicArray<int>&
```

## 总结

你的赋值运算符必须接受 `const DynamicArray<T>&` 参数，以支持以下情况：

1. 赋值右侧是**常量对象**。
2. 赋值右侧是**临时对象**（如函数返回值）。
3. 保持与标准库容器（如 `std::vector`）一致的接口规范。

修正后的代码通过 `const` 引用解决了这个问题，同时也遵循了C++的最佳实践。

# C++中的继承：面向对象编程的核心机制

## 1. 基本语法与概念

继承允许派生类（子类）复用基类（父类）的属性和方法，形成"is-a"关系。

```
class Base {
    // 基类成员（属性/方法）
};

class Derived : [访问限定符] Base {
    // 派生类新增成员
};
```

## 2. 继承方式与访问权限

继承方式决定基类成员在派生类中的可访问性：

| 继承方式 | 基类 `public` 成员 | 基类 `protected` 成员 | 基类 `private` 成员 |
| --- | --- | --- | --- |
| `public` 继承 | 保持 `public` | 保持 `protected` | 不可访问 |
| `protected` 继承 | 变为 `protected` | 变为 `protected` | 不可访问 |
| `private` 继承 | 变为 `private` | 变为 `private` | 不可访问 |

**示例：**

```
class Base {
public:
    int publicVar;
protected:
    int protectedVar;
private:
    int privateVar;
};

class PublicDerived : public Base {
    // publicVar 可通过对象访问，protectedVar 仅限派生类内部访问
};
```

## 3. 构造函数与析构函数的执行顺序

- **构造顺序**：先调用基类构造函数，再调用派生类构造函数。
- **析构顺序**：先调用派生类析构函数，再调用基类析构函数。

```
class Base {
public:
    Base() { cout << "Base构造" << endl; }
    ~Base() { cout << "Base析构" << endl; }
};

class Derived : public Base {
```

```
public:
    Derived() { cout << "Derived构造" << endl; }
    ~Derived() { cout << "Derived析构" << endl; }
};

// 输出：
// Base构造 → Derived构造 → Derived析构 → Base析构
```

### 4. 函数重写与多态

- **虚函数（`virtual`）**：基类中用 `virtual` 声明的函数可在派生类中重写。
  - 父类要写成虚函数；
  - 子类的函数名要与父类相同；
  - 函数参数类型不能是对象本身，应当是对象的引用。
  - 虚函数相当于一个函数指针，和普通函数所占的内存不同
- **多态实现**：通过基类指针/引用调用虚函数时，动态绑定到派生类的重写版本。

```
class Shape {
public:
    virtual void draw() { cout << "绘制图形" << endl; }
};

class Circle : public Shape {
public:
    void draw() override { cout << "绘制圆形" << endl; }  // C++11 用 `override` 显式标记重写
};

Shape* shape = new Circle();
shape->draw();  // 输出："绘制圆形"（动态绑定）
```

### 5. 多重继承与菱形继承问题

- **多重继承语法**：

  ```
  class A {}; class B {};
  class C : public A, public B {};  // C 继承自 A 和 B
  ```

- **菱形继承（钻石问题）**：

  ```
  class A {};
  class B : public A {};
  class C : public A {};
  class D : public B, public C {};  // D 继承自 B 和 C，导致 A 的成员重复
  ```

  **解决方案：虚拟继承（`virtual`）**

  ```
  class A {};
  class B : virtual public A {};  // 声明虚拟基类
  class C : virtual public A {};
  class D : public B, public C {};  // D 中仅保留一份 A 的成员
  ```

### 6. 继承与组合的选择

- **继承（is-a）**：适用于严格的类型派生关系（如 `Dog` 是 `Animal`）。
- **组合（has-a）**：适用于"包含"关系（如 `Car` 包含 `Engine`），推荐优先使用组合以降低耦合。

### 完整示例代码

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

class Animal {
protected:
    string name;
public:
    Animal(string n) : name(n) { cout << "Animal构造: " << name << endl; }
    virtual void speak() { cout << name << "发出声音" << endl; }
    virtual ~Animal() { cout << "Animal析构: " << name << endl; }
};

class Dog : public Animal {
public:
    Dog(string n) : Animal(n) { cout << "Dog构造: " << name << endl; }
    void speak() override { cout << name << "汪汪叫" << endl; }
    ~Dog() { cout << "Dog析构: " << name << endl; }
};

int main() {
    Animal* pet = new Dog("Buddy");
    pet->speak();  // 多态调用：输出"Buddy汪汪叫"
    delete pet;    // 正确释放资源（派生类析构先执行）
    return 0;
}
```

### 总结：继承的设计原则

1. 基类析构函数必须为 `virtual`，避免内存泄漏。
2. 优先使用 `public` 继承，谨慎使用 `protected` 和 `private`。
3. 多重继承需警惕菱形问题，通过虚拟继承或组合优化设计。
4. 继承层次不宜过深（建议不超过3层），保持代码清晰性。
5. 包含成纯虚函数的类叫做抽象类，不能直接生成对象

```
// 纯虚函数示例
virtual void PowerOn() = 0;
```

## 设计模式(Design Patterns)

### 定义：设计程序时的基本模式，一些经典的范例，一些好的设计方案

### 基本原则

- 单一职责原则（Single Responsibility Principle）：一个对象应该对其他对象有最少的了解。
- 开闭原则（Open/Closed Principle）：对扩展开放，对修改关闭。
- 里氏替换原则（Liskov Substitution Principle）：子类可以替换父类而不影响程序的正确性。
- 接口隔离原则（Interface Segregation Principle）：使用多个专门的接口，而不是一个大而全的接口。
- 依赖倒置原则（Dependency Inversion Principle）：依赖抽象而非具体实现。

# 见 shape.cpp 学习如何对输出进行多态化（friend 和 virtual 不能同时出现

## STL

### 容器

可容纳各种数据类型的通用数据结构，是类模板。如`vector`，`list`，`queue`，`deque`等

1. 顺序容器：`vector`，`list`
2. 关联容器：`set`，`multiset`，`map`，`multimap`
3. 容器适配器： `stack`，`queue`

### 迭代器

相当于是"泛型指针"，可以用于依次存储容器中的元素。是面向对象版本指针。
用于指向顺序容器和关联容器中的元素。`const`迭代器不可修改容器内的元素。
`容器名称::iterator 变量名`

### 函数对象

对`()`操作符的重载。函数对象本质上还是一个对象，只不过看上去像一个函数，也称Functor。

```
class Adder {
public:
    int operator()(int a, int b) const {
        return a + b;
    }
};

int main() {
    Adder adder;
    int result = adder(3, 4);  // 像调用函数一样使用对象
    std::cout << "Result: " << result << std::endl;  // 输出: 7
    return 0;
}
```

## 算法

独立于数据类型存在。通过迭代器与容器构建联系。
例如：插入，删除，查找，排序。分为修改容器的算法和不修改容器的算法。