程设救急

1. 数位翻转函数

我们需要实现一个函数，将一个整数的数字进行翻转。为了高效地实现这一功能，我们可以直接操作数字，避免转换成字符串再翻转。

实现代码：

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

int reverseNumber(int num) {
    int reversed = 0;
    while (num != 0) {
        int digit = num % 10;  // 获取当前数字的最后一位
        reversed = reversed * 10 + digit;  // 翻转数字
        num /= 10;  // 去掉数字的最后一位
    }
    return reversed;
}

int main() {
    int num = 123;
    cout << "Reversed number: " << reverseNumber(num) << endl;  // 输出：321
    return 0;
}
```

解释：

- 我们通过不断取出数字的最后一位并将其加入结果的末尾，最终完成翻转。
- 由于没有使用字符串类型转换，效率较高。

---

2. 判定一个整数是否是质数的函数

判断一个数是否是质数的标准方法是通过除以从 `2` 到该数的平方根。如果一个数能被其中的某个数整除，那么它就不是质数。

实现代码：

```
#include <iostream>
#include <cmath>  // 用于sqrt函数
```

```cpp
using namespace std;

bool isPrime(int num) {
    if (num <= 1) return false;  // 1 和负数不是质数
    if (num == 2) return true;   // 2 是质数
    if (num % 2 == 0) return false;  // 排除偶数

    int limit = sqrt(num);  // 只检查到平方根
    for (int i = 3; i <= limit; i += 2) {  // 只检查奇数
        if (num % i == 0) return false;
    }
    return true;
}

int main() {
    int num = 29;
    cout << "Is prime: " << isPrime(num) << endl;  // 输出: 1 (是质数)
    return 0;
}
```

解释：

- 我们通过 `sqrt(num)` 限定检查范围，优化了效率。
- 对于偶数（除了 2）直接排除，减少不必要的计算。
- 使用了 `i += 2` 只检查奇数，从而减少了检查次数。

---

3. 高精度乘法运算

为了实现高精度乘法，我们不能直接使用C++的整数类型。我们需要用字符表示每一位数字，然后手动进行数字的乘法和进位处理。输入是两个字符，输出是一个字符。

实现代码：

```cpp
#include <iostream>
#include <vector>
#include <algorithm>  // 用于reverse函数
using namespace std;

string multiplyHighPrecision(string num1, string num2) {
    // 处理特殊情况
    if (num1 == "0" || num2 == "0") return "0";

    int m = num1.size(), n = num2.size();
    vector<int> result(m + n, 0);
```

```
    // 逐位相乘，并进行加和
    for (int i = m - 1; i >= 0; --i) {
        for (int j = n - 1; j >= 0; --j) {
            int mul = (num1[i] - '0') * (num2[j] - '0');
            int sum = mul + result[i + j + 1];  // 计算当前位的总和
            result[i + j + 1] = sum % 10;  // 当前位的数字
            result[i + j] += sum / 10;  // 进位
        }
    }

    // 去掉前导零
    string resultStr = "";
    for (int num : result) {
        if (!(resultStr.empty() && num == 0)) {  // 跳过前导零
            resultStr += to_string(num);
        }
    }

    return resultStr;
}

int main() {
    string num1 = "12345", num2 = "6789";
    cout << "High Precision Multiply: " << multiplyHighPrecision(num1, num2)
<< endl;
    return 0;
}
```

解释：

- 通过模拟手动乘法进行逐位计算。
- 每一位的结果存储在 `result` 数组中，最后生成结果字符串。
- 结果字符串在生成后去除了前导零。

模板设计思路：

- **排序规则**：排序时首先按 `A` 排序，如果 `A` 相同，则按 `B` 排序，如果 `B` 也相同，则按 `C` 排序。
- **升序或降序**：可以通过传入不同的比较函数来控制升序或降序。
- **并列输出**：如果所有排序标准都相同，则会并列输出。

我们可以设计一个通用的比较函数模板，在比较时灵活地根据需要返回升序或降序的结果。

示例代码：

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

```cpp
#include <algorithm>
using namespace std;

// 自定义结构体，表示每一项
struct Data {
    int A, B, C;

    Data(int a, int b, int c) : A(a), B(b), C(c) {}
};

// 比较函数模板，用于多重标准排序
template <typename T>
bool compare(const T& x, const T& y, bool A_asc, bool B_asc, bool C_asc) {
    // 按 A 排序
    if (x.A != y.A) {
        return A_asc ? x.A < y.A : x.A > y.A;
    }
    // 如果 A 相同，则按 B 排序
    if (x.B != y.B) {
        return B_asc ? x.B < y.B : x.B > y.B;
    }
    // 如果 B 也相同，则按 C 排序
    return C_asc ? x.C < y.C : x.C > y.C;
}

int main() {
    // 初始化数据
    vector<Data> data = {
        {5, 2, 3},
        {5, 3, 1},
        {4, 5, 6},
        {5, 3, 2},
        {4, 5, 2},
    };

    // 设置排序规则：按A升序，B降序，C升序
    bool A_asc = true;
    bool B_asc = false;
    bool C_asc = true;

    // 使用自定义的比较函数进行排序
    sort(data.begin(), data.end(), [&](const Data& x, const Data& y) {
        return compare(x, y, A_asc, B_asc, C_asc);
    });

    // 输出排序结果
    for (const auto& d : data) {
        cout << "(" << d.A << ", " << d.B << ", " << d.C << ")" << endl;
    }
```

```
    return 0;
}
```

说明：

1. `Data` **结构体**：表示排序的数据项，其中包含 `A`，`B`，`C` 三个字段。
2. `compare` **函数模板**：这是一个自定义的比较函数，根据传入的排序标志 `A_asc`，`B_asc`，`C_asc` 来决定是否进行升序或降序排序。
   - 如果 `A` 不相同，直接按 `A` 排序；
   - 如果 `A` 相同，按 `B` 排序；
   - 如果 `A` 和 `B` 都相同，按 `C` 排序。
3. **排序规则的控制**：通过 `A_asc`，`B_asc`，`C_asc` 控制升序（`true`）或降序（`false`）：
   - `A_asc`：控制 `A` 是升序还是降序；
   - `B_asc`：控制 `B` 是升序还是降序；
   - `C_asc`：控制 `C` 是升序还是降序。
4. `sort` **调用**：我们使用了 C++ 标准库中的 `sort`，并传入自定义的比较函数。在比较过程中，通过 `compare` 函数来决定排序的方式。

C++标准库的 `<algorithm>` 头文件提供了许多非常有用的算法函数，可以大大简化程序设计中的常见任务。下面列出了一些常用的 `<algorithm>` 函数，并给出它们的主要参数列表和简明的代码解释。

1. `std::sort`

- **功能**：对容器中的元素进行排序。
- **参数**：

```
void sort(RandomAccessIterator first, RandomAccessIterator last);
void sort(RandomAccessIterator first, RandomAccessIterator last,
Compare comp);
```

  - `first, last`：容器中待排序的区间。
  - `comp`：自定义的比较函数或函数对象。
- **示例**：

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

int main() {
    std::vector<int> vec = {5, 3, 8, 1, 2};
    std::sort(vec.begin(), vec.end());  // 默认升序排序
    for (int num : vec) {
        std::cout << num << " ";
```

```
    }
    return 0;
}
```

输出：

```
1 2 3 5 8
```

---

2. `std::reverse`

- **功能**：反转容器中的元素顺序。
- **参数**：

```
void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

  - `first, last`：容器中待反转的区间。
- **示例**：

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

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    std::reverse(vec.begin(), vec.end());  // 反转顺序
    for (int num : vec) {
        std::cout << num << " ";
    }
    return 0;
}
```

输出：

```
5 4 3 2 1
```

---

3. `std::find`

- **功能**：查找容器中是否包含指定元素。

- **参数**：

```
iterator find(InputIterator first, InputIterator last, const T& value);
```

  - `first, last`：容器的起始和结束迭代器。
  - `value`：要查找的元素。
- **示例**：

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

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    auto it = std::find(vec.begin(), vec.end(), 3);  // 查找元素3
    if (it != vec.end()) {
        std::cout << "Found: " << *it << std::endl;
    } else {
        std::cout << "Not Found" << std::endl;
    }
    return 0;
}
```

  输出：

```
Found: 3
```

---

4. `std::count`

- **功能**：统计容器中某个元素出现的次数。
- **参数**：

```
size_t count(InputIterator first, InputIterator last, const T& value);
```

  - `first, last`：容器的起始和结束迭代器。
  - `value`：要统计的元素。
- **示例**：

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

```cpp
int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5, 3};
    int countOf3 = std::count(vec.begin(), vec.end(), 3);  // 统计3的出现次数
    std::cout << "Count of 3: " << countOf3 << std::endl;
    return 0;
}
```

输出：

```
Count of 3: 2
```

---

5. `std::max`

- **功能**：返回两个元素中的较大者。
- **参数**：

```cpp
const T& max(const T& a, const T& b);
const T& max(const T& a, const T& b, Compare comp);
```

  - `a, b`：比较的两个元素。
  - `comp`：自定义的比较函数或函数对象。
- **示例**：

```cpp
#include <algorithm>
#include <iostream>

int main() {
    int a = 5, b = 10;
    int result = std::max(a, b);  // 获取较大的值
    std::cout << "Max: " << result << std::endl;
    return 0;
}
```

输出：

```
Max: 10
```

---

6. `std::accumulate`

- **功能**：对容器中的元素进行累加或执行其他操作。
- **参数**：

```
T accumulate(InputIterator first, InputIterator last, T init);
T accumulate(InputIterator first, InputIterator last, T init, BinaryOp
op);
```

  - `first, last`：容器的起始和结束迭代器。
  - `init`：初始值。
  - `op`：二元操作函数，默认为加法。
- **示例**：

```
#include <algorithm>
#include <vector>
#include <iostream>
#include <numeric>  // 包含accumulate

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    int sum = std::accumulate(vec.begin(), vec.end(), 0);  // 求和
    std::cout << "Sum: " << sum << std::endl;
    return 0;
}
```

  输出：

```
Sum: 15
```

---

7. `std::unique`

- **功能**：去除容器中相邻的重复元素。
- **参数**：

```
OutputIterator unique(InputIterator first, InputIterator last);
OutputIterator unique(InputIterator first, InputIterator last,
BinaryPredicate pred);
```

  - `first, last`：容器的起始和结束迭代器。
  - `pred`：自定义的比较函数，决定是否认为元素相同。

- **示例**：

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

int main() {
    std::vector<int> vec = {1, 1, 2, 3, 3, 4, 5};
    auto it = std::unique(vec.begin(), vec.end());  // 去重
    vec.erase(it, vec.end());  // 删除重复元素后的元素
    for (int num : vec) {
        std::cout << num << " ";
    }
    return 0;
}
```

输出：

```
1 2 3 4 5
```

---

8. `std::lower_bound`

- **功能**：返回一个迭代器，指向容器中第一个不小于指定值的元素。
- **参数**：

```
iterator lower_bound(BidirectionalIterator first, BidirectionalIterator
last, const T& value);
iterator lower_bound(BidirectionalIterator first, BidirectionalIterator
last, const T& value, Compare comp);
```

  - `first, last`：容器的起始和结束迭代器。
  - `value`：查找的值。
  - `comp`：自定义的比较函数。
- **示例**：

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

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    auto it = std::lower_bound(vec.begin(), vec.end(), 3);  // 查找>=3的
```

```
    第一个元素
    std::cout << "Lower bound: " << *it << std::endl;
    return 0;
}
```

输出：

```
Lower bound: 3
```

---

9. `std::shuffle`

- **功能**：对容器中的元素进行随机排列。
- **参数**：

```
void shuffle(RandomAccessIterator first, RandomAccessIterator last,
RandomNumberGenerator&& g);
```

  - `first, last`：容器的起始和结束迭代器。
  - `g`：随机数生成器，用于产生随机数。
- **示例**：

```
#include <algorithm>
#include <vector>
#include <iostream>
#include <random>

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    std::random_device rd;
    std::mt199
```

37 g(rd()); std::shuffle(vec.begin(), vec.end(), g); // 打乱顺序 for (int num : vec) { std::cout << num << " "; } return 0; }

在 C++ 中，运算符重载允许你为自定义类型定义标准运算符的行为。通过重载运算符，你可以使自定义类的对象支持常规的运算符操作（如 `+`, `>>`, `<<`, `[]` 等）。下面是几个常见的运算符重载示范。

1. 重载 `+` 运算符

我们可以重载 `+` 运算符，使其支持两个自定义类型的对象相加。

示例：重载 `+` 运算符（向量加法）

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

class Vector {
public:
    int x, y;

    Vector(int x, int y) : x(x), y(y) {}

    // 重载 + 运算符
    Vector operator+(const Vector& other) {
        return Vector(x + other.x, y + other.y);
    }

    void display() const {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};

int main() {
    Vector v1(1, 2), v2(3, 4);
    Vector v3 = v1 + v2;  // 使用重载的 + 运算符
    v3.display();  // 输出 (4, 6)
    return 0;
}
```

2. 重载 `>>` 运算符（输入流）

`>>` 运算符通常用于输入流，我们可以重载它，使其能够从输入流中读取数据并赋值给类的成员。

示例：重载 `>>` 运算符（输入一个 `Point` 对象）

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

```cpp
class Point {
public:
    int x, y;

    Point(int x = 0, int y = 0) : x(x), y(y) {}

    // 重载 >> 运算符
    friend istream& operator>>(istream& in, Point& p) {
        in >> p.x >> p.y;
        return in;
    }

    void display() const {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};

int main() {
    Point p;
    cout << "Enter x and y: ";
    cin >> p;  // 使用重载的 >> 运算符
    p.display();  // 输出输入的点，例如 (3, 4)
    return 0;
}
```

3. 重载 `<<` 运算符（输出流）

`<<` 运算符用于输出流，我们可以重载它，使其能够输出类的内容。

示例：重载 `<<` 运算符（输出一个 `Point` 对象）

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

class Point {
public:
    int x, y;

    Point(int x = 0, int y = 0) : x(x), y(y) {}

    // 重载 << 运算符
    friend ostream& operator<<(ostream& out, const Point& p) {
        out << "(" << p.x << ", " << p.y << ")";
        return out;
    }
};

int main() {
```

```
    Point p(3, 4);
    cout << "Point: " << p << endl;  // 使用重载的 << 运算符
    return 0;
}
```

输出：

```
Point: (3, 4)
```

4. 重载 `[]` 运算符（下标操作）

`[]` 运算符通常用于数组或容器类的元素访问，我们可以重载它，使其支持自定义对象的下标访问。

示例：重载 `[]` 运算符（模拟一个简单的数组类）

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

class Array {
private:
    int arr[5];

public:
    Array() {
        for (int i = 0; i < 5; ++i) {
            arr[i] = i + 1;
        }
    }

    // 重载 [] 运算符
    int& operator[](int index) {
        if (index >= 0 && index < 5) {
            return arr[index];
        } else {
            throw out_of_range("Index out of range");
        }
    }
};

int main() {
    Array a;
    cout << "a[2] = " << a[2] << endl;  // 输出 3
    a[2] = 10;  // 修改数组的值
    cout << "a[2] = " << a[2] << endl;  // 输出 10
    return 0;
}
```

输出：

```
a[2] = 3
a[2] = 10
```

5. 重载 `=` 运算符（赋值操作）

当我们创建一个类的对象时，如果没有定义赋值运算符，编译器会提供默认的赋值操作。但如果类中有指针或动态分配的内存，我们通常需要重载 `=` 运算符，以便正确地复制对象。

示例：重载 `=` 运算符（深拷贝）

```
#include <iostream>
#include <cstring>  // 为了使用 strcpy
using namespace std;

class String {
private:
    char* str;

public:
    String(const char* s = "") {
        str = new char[strlen(s) + 1];
        strcpy(str, s);
    }

    // 重载 = 运算符（深拷贝）
    String& operator=(const String& other) {
        if (this != &other) {  // 防止自赋值
            delete[] str;  // 释放现有的内存
            str = new char[strlen(other.str) + 1];
            strcpy(str, other.str);
        }
        return *this;
    }

    void display() const {
        cout << str << endl;
    }

    ~String() {
        delete[] str;
    }
};

int main() {
    String s1("Hello");
    String s2;
```

```
    s2 = s1;  // 使用重载的 = 运算符
    s2.display();  // 输出 "Hello"
    return 0;
}
```

总结：

1. `+` **运算符**：通过重载 `operator+` 让两个对象能够相加。
2. `>>` **运算符**：通过重载 `operator>>` 让输入流可以读取对象的数据。
3. `<<` **运算符**：通过重载 `operator<<` 让输出流可以输出对象的数据。
4. `[]` **运算符**：通过重载 `operator[]` 让对象像数组一样支持下标访问。
5. `=` **运算符**：通过重载 `operator=` 使得类支持赋值操作，处理深拷贝的情形。
   这些重载运算符使得自定义类能够像内置类型一样进行自然的运算符操作，提升了代码的可读性和使用便捷性。

C++ 中与字符串操作相关的类和库主要包括以下几个重要的类：`std::string`（标准字符串类）、`std::stringstream`（字符串流类）、`std::ostringstream`（输出字符串流）、`std::istringstream`（输入字符串流）、以及 `std::wstring`（宽字符字符串类）等。这些类用于处理字符串数据，提供了丰富的操作方法。接下来，我将简要介绍这些类及其主要方法，并给出代码实例。

1. `std::string` 类

`std::string` 是 C++ 标准库中提供的用于处理字符串的类，提供了许多操作函数，可以方便地进行字符串的拼接、查找、替换等操作。

主要成员函数：

- **构造函数**：

```
std::string s;  // 空字符串
std::string s = "Hello";  // 使用字符常量初始化
std::string s(5, 'a');  // 字符串长度为5，内容为 'aaaaa'
```

- **大小和修改**：

```
s.length();  // 返回字符串的长度
s.empty();   // 判断字符串是否为空
s.clear();   // 清空字符串
s.append(" World");  // 拼接字符串
s += "!!";   // 追加内容
```

- **查找与替换**：

```
s.find("llo");  // 返回子字符串的位置
s.replace(0, 5, "Hi");  // 替换从位置 0 开始的5个字符为 "Hi"
```

- **访问字符**：

```
s[0];  // 访问第一个字符
s.at(1);  // 安全地访问第二个字符，越界会抛出异常
```

示例代码：

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

int main() {
    string str = "Hello";
    str.append(" World");
    cout << str << endl;  // 输出 "Hello World"

    if (str.find("World") != string::npos) {
        cout << "Found 'World' in string" << endl;
    }

    str.replace(0, 5, "Hi");
    cout << str << endl;  // 输出 "Hi World"
    return 0;
}
```

---

2. `std::stringstream` 类

`std::stringstream` 是一个字符串流类，允许你将字符串用作流进行输入输出操作。它可以看作是 `std::cin` 和 `std::cout` 的字符串版。

主要成员函数：

- **构造函数**：

```
std::stringstream ss;  // 空的字符串流
std::stringstream ss("123");  // 用初始字符串初始化
```

- **读取和写入**：

```
ss << "Hello";  // 将字符串 "Hello" 写入流
int x;
ss >> x;  // 从流中读取整数
```

- **转换成字符串**：

```
string str = ss.str();  // 获取流中的所有内容为字符串
ss.str("New string");  // 设置流的内容
```

示例代码：

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

int main() {
    stringstream ss;
    int num = 42;
    ss << "The number is " << num;
    string result = ss.str();
    cout << result << endl;  // 输出 "The number is 42"

    stringstream ss2("123 456");
    int a, b;
    ss2 >> a >> b;
    cout << a << " " << b << endl;  // 输出 123 456
    return 0;
}
```

---

3. `std::ostringstream` 类

`std::ostringstream` 是一个输出字符串流类，它用于将数据输出到字符串中。它继承自 `std::ostream`，可以使用流的操作符将数据写入字符串。

主要成员函数：

- **构造函数**：

```
std::ostringstream oss;  // 创建一个空的输出字符串流
```

- **输出操作**：

```
oss << "Number: " << 42;  // 向字符串流写入数据
```

- **获取字符串**：

```
string str = oss.str();  // 获取流中存储的字符串
```

示例代码：

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

int main() {
    ostringstream oss;
    oss << "Hello" << " World " << 2024;
    string result = oss.str();
    cout << result << endl;  // 输出 "Hello World 2024"
    return 0;
}
```

---

4. `std::istringstream` 类

`std::istringstream` 是一个输入字符串流类，用于将字符串转换为数据。它继承自 `std::istream`，通过流操作符读取字符串中的内容。

主要成员函数：

- **构造函数**：

```
std::istringstream iss("123 456");  // 从字符串初始化
```

- **读取数据**：

```
int x;
iss >> x;  // 从流中读取数据
```

- **重置流内容**：

```
iss.str("New string");  // 重置流的内容
```

示例代码：

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

int main() {
    istringstream iss("123 456");
    int a, b;
    iss >> a >> b;
    cout << a << " " << b << endl;  // 输出 123 456
    return 0;
}
```

---

5. `std::wstring` 类

`std::wstring` 是 C++ 中的宽字符字符串类，用于处理宽字符（`wchar_t`）。它与 `std::string` 类似，只不过它处理的是宽字符数据。

主要成员函数：

- **构造函数**：

```
std::wstring ws = L"Hello";  // 使用宽字符常量初始化
```

- **查找和替换**：

```
ws.find(L"Hello");
ws.replace(0, 5, L"Hi");
```

- **访问字符**：

```
ws[0];  // 访问第一个字符
```

示例代码：

```
#include <iostream>
#include <string>
```

```cpp
using namespace std;

int main() {
    wstring ws = L"Hello World";
    wcout << ws << endl;  // 输出宽字符字符串

    ws.replace(0, 5, L"Hi");
    wcout << ws << endl;  // 输出 "Hi World"
    return 0;
}
```

总结

1. `std::string`：用于处理普通字符串，提供了丰富的字符串操作方法。
2. `std::stringstream`：允许将字符串作为输入输出流进行处理，方便地进行字符串的转换。
3. `std::ostringstream`：用于将数据输出到字符串中，类似于 `std::cout`。
4. `std::istringstream`：用于从字符串中读取数据，类似于 `std::cin`。
5. `std::wstring`：用于处理宽字符字符串，支持更广泛的字符集。

通过这些类，C++ 提供了强大的字符串操作功能，可以方便地进行字符串的拼接、读取、转换等任务。
在 C++ 中，继承是面向对象编程的一个重要特性。通过继承，可以让一个类（子类）继承另一个类（父类）的属性和行为，从而实现代码的复用和扩展。以下是几种常见的继承模板，并介绍在操作时需要注意的事项。

1. 单继承

单继承是最基本的继承方式，一个子类继承自一个父类。

示例代码：

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

// 父类
class Animal {
public:
    void eat() {
        cout << "Animal is eating" << endl;
    }
};

// 子类
```

```
class Dog : public Animal {
public:
    void bark() {
        cout << "Dog is barking" << endl;
    }
};

int main() {
    Dog d;
    d.eat();  // 继承自 Animal 类
    d.bark(); // Dog 类自己定义的
    return 0;
}
```

注意事项：

- `Dog` 类通过 `public` 继承自 `Animal` 类，可以直接访问 `Animal` 类的公共成员。
- 在子类中，父类的成员函数会自动成为可用的，除非它们被子类重载。

2. 多重继承

多重继承允许一个类从多个父类继承成员。虽然 C++ 支持多重继承，但要小心处理类的构造函数、析构函数以及名字冲突等问题。

示例代码：

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

// 父类 1
class Animal {
public:
    void eat() {
        cout << "Animal is eating" << endl;
    }
};

// 父类 2
class Mammal {
public:
    void breathe() {
        cout << "Mammal is breathing" << endl;
    }
};

// 子类
class Dog : public Animal, public Mammal {
public:
```

```
    void bark() {
        cout << "Dog is barking" << endl;
    }
};

int main() {
    Dog d;
    d.eat();     // 来自 Animal 类
    d.breathe(); // 来自 Mammal 类
    d.bark();    // Dog 类自己定义的
    return 0;
}
```

注意事项：

- 子类 `Dog` 可以继承多个父类的成员。
- 如果多个父类中有同名函数，调用时需要明确指定要使用哪个父类的成员。

3. 虚继承（解决菱形继承问题）

虚继承用于解决菱形继承问题，在多重继承中，两个子类继承自同一个父类的情况下，子类会有多个父类实例，虚继承确保父类只被实例化一次。

示例代码：

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

// 父类
class Animal {
public:
    Animal() {
        cout << "Animal constructor" << endl;
    }
    void eat() {
        cout << "Animal is eating" << endl;
    }
};

// 子类 1
class Mammal : virtual public Animal {
public:
    Mammal() {
        cout << "Mammal constructor" << endl;
    }
};

// 子类 2
```

```
class Bird : virtual public Animal {
public:
    Bird() {
        cout << "Bird constructor" << endl;
    }
};

// 最终子类
class Bat : public Mammal, public Bird {
public:
    Bat() {
        cout << "Bat constructor" << endl;
    }
};

int main() {
    Bat b;
    b.eat();  // 调用来自 Animal 类的 eat()
    return 0;
}
```

注意事项：

- 在虚继承中，父类的构造函数只会被调用一次。
- `Mammal` 和 `Bird` 都虚继承自 `Animal`，因此 `Bat` 只会有一个 `Animal` 的实例。

4. 私有继承

私有继承使得父类的公有成员在子类中变为私有成员。通常用于实现 "实现" 而不是 "是一个" 的关系。

示例代码：

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

// 父类
class Animal {
public:
    void eat() {
        cout << "Animal is eating" << endl;
    }
};

// 子类
class Dog : private Animal {
public:
    void bark() {
```

```
        cout << "Dog is barking" << endl;
    }
    void eatDog() {
        eat();  // 通过子类的成员函数调用父类的 eat
    }
};

int main() {
    Dog d;
    d.bark();
    d.eatDog();  // 正确，调用 eatDog()，间接调用父类的 eat()

    // d.eat();  // 错误，不能直接访问 eat()，因为继承为私有
    return 0;
}
```

注意事项：

- 私有继承意味着父类的公有成员在子类中变为私有，无法直接通过对象访问父类的成员，只能通过子类中的公有成员间接访问。

5. 保护继承

保护继承将父类的公有和保护成员都作为子类的保护成员。这种继承方式介于公有继承和私有继承之间。

示例代码：

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

// 父类
class Animal {
public:
    void eat() {
        cout << "Animal is eating" << endl;
    }
protected:
    void sleep() {
        cout << "Animal is sleeping" << endl;
    }
};

// 子类
class Dog : protected Animal {
public:
    void bark() {
        cout << "Dog is barking" << endl;
```

```
    }
    void dogSleep() {
        sleep();  // 可以访问父类的 protected 成员
    }
};

int main() {
    Dog d;
    d.bark();
    d.dogSleep();  // 可以访问 protected 成员 sleep()

    // d.eat();  // 错误，无法直接访问 eat()，因为继承为保护
    return 0;
}
```

注意事项：

- 保护继承下，父类的公有成员会变为子类的保护成员，子类内部可以访问父类的公有和保护成员，但外部无法访问。

---

6. 多态与虚函数

多态是继承的重要特性，允许通过基类指针或引用调用派生类中的重写方法。为了启用多态，基类的函数需要声明为虚函数（`virtual`）。

示例代码：

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

// 基类
class Animal {
public:
    virtual void sound() {  // 声明为虚函数
        cout << "Animal makes a sound" << endl;
    }
    virtual ~Animal() {}  // 虚析构函数
};

// 派生类
class Dog : public Animal {
public:
    void sound() override {  // 重写基类的虚函数
        cout << "Dog barks" << endl;
    }
};
```

```cpp
int main() {
    Animal* animal = new Dog();
    animal->sound();  // 调用派生类的 sound()，实现动态绑定
    delete animal;    // 确保删除派生类对象时，调用正确的析构函数
    return 0;
}
```

注意事项：

- 使用虚函数时，确保基类的析构函数也声明为虚函数，以便正确地销毁派生类对象。
- `override` 关键字可以确保你确实重载了基类的虚函数，编译器会进行检查。

总结：

- **单继承**：继承自一个父类，子类可以访问父类的公有成员。
- **多重继承**：继承自多个父类，可能会遇到名字冲突和构造函数问题，需要小心。
- **虚继承**：用于解决菱形继承问题，保证父类只实例化一次。
- **私有继承**：父类的公有成员在子类中变为私有，通常用于实现继承而非"is-a"关系。
- **保护继承**：父类的公有成员变为子类的保护成员，适用于继承时不需要对外公开父类的接口。
- **多态和虚函数**：通过虚函数实现多态，使得基类指针或引用可以调用派生类的重写方法。

继承是 C++ 的一个强大特性，但要注意避免一些潜在的问题，如菱形继承、构造函数调用、虚函数的使用等。