实验2 类和对象_基础编程2

说明

- **阅读文档**

  请阅读文档，明确各项任务的具体要求、提交方式与截止时间。

- **编程建议**

  随文档提供了部分代码。测试模块请直接从压缩包解压使用，避免手工录入出错。

  设计性实验，请先独立思考和编写。缺失思考和试错过程，会削弱训练效果。

  > 本文档为简化代码呈现，将类的声明和实现都写在`.hpp`文件中。
  >
  > 实际项目中，建议将声明(`.hpp`)与实现(`.cpp`)分离，以减少编译依赖。

- **提交规范**
  - 一次实验的所有任务写在一篇博客中
  - 发布博客后，还需要在作业系统中提交链接
  - 晚交的实验，按课程规则处理（分数折半）

一、实验目的

- 理解类的组合机制（has-a），能熟练用 C++ 定义与使用组合类
- 理解深复制与浅复制的区别
- 灵活运用标准库（array、vector、string、迭代器、算法库等）解决实际问题
- 面向具体问题，运用面向对象思维设计类（自定义/标准库），合理组合并编程解决

二、实验准备

系统浏览/复习以下内容：

- 类的抽象、设计（教材第4-6章）
- 组合类：适配的问题场景、定义和用法（教材第4章）
- 数据共享、保护（教材第5章）
- 标准库`array`、`vector`、`string`、迭代器、算法库等用法（教材第6章、菜鸟教程、 cppreference）

三、实验内容

1. 实验任务1

验证性实验：组合类的定义和使用。实践、理解代码，回答问题。

组合类抽象现实世界***has-a***关系。这个任务模拟GUI窗口和按钮组件。覆盖以下内容：

> - 组合类定义和使用（注意构造函数写法）
> - 数据的共享和保护：`const`引用作形参
> - 标准库`vector`, `string`

代码组织

- button.hpp　　Button类定义
- window.hpp　　Window类定义
- task1.cpp　　测试模块、main

button.hpp

```
#pragma once

#include <iostream>
#include <string>

class Button {
public:
    Button(const std::string &label_);
    const std::string& get_label() const;
    void click();

private:
    std::string label;
};

Button::Button(const std::string &label_): label{label_} {
}
```

```
18
19  inline const std::string& Button::get_label() const {
20      return label;
21  }
22
23  inline void Button::click() {
24      std::cout << "Button '" << label << "' clicked\n";
25  }
```

window.hpp

```
1   #pragma once
2
3   #include <iostream>
4   #include <vector>
5   #include <algorithm>
6   #include "button.hpp"
7
8   // 窗口类
9   class Window{
10  public:
11      Window(const std::string &title_);
12      void display() const;
13      void close();
14      void add_button(const std::string &label);
15      void click_button(const std::string &label);
16
17  private:
18      bool has_button(const std::string &label) const;
19
20  private:
21      std::string title;
22      std::vector<Button> buttons;
23  };
24
25  Window::Window(const std::string &title_): title{title_} {
26      buttons.push_back(Button("close"));
27  }
28
29  inline void Window::display() const {
30      std::string s(40, '*');
31      std::cout << s << std::endl;
32      std::cout << "window : " << title << std::endl;
33      int cnt = 0;
34      for(const auto &button: buttons)
35          std::cout << ++cnt << ". " << button.get_label() << std::endl;
36      std::cout << s << std::endl;
37  }
38
39  inline void Window::close() {
40      std::cout << "close window '" << title << "'" << std::endl;
41      click_button("close");
42  }
```

```cpp

inline bool Window::has_button(const std::string &label) const {
    for(const auto &button: buttons)
        if(button.get_label() == label)
            return true;

    return false;
}

inline void Window::add_button(const std::string &label) {
    if(has_button(label))
        std::cout << "button " << label << " already exists!\n";
    else
        buttons.push_back(Button(label));
}

inline void Window::click_button(const std::string &label) {
    for(auto &button:buttons)
        if(button.get_label() == label) {
            button.click();
            return;
        }

    std::cout << "no button: " << label << std::endl;
}
```

task1.cpp

```cpp
#include "window.hpp"
#include <iostream>

void test(){
    Window w("Demo");
    w.add_button("add");
    w.add_button("remove");
    w.add_button("modify");
    w.add_button("add");
    w.display();
    w.close();
}

int main() {
    std::cout << "用组合类模拟简单GUI:\n";
    test();
}
```

编译、运行程序，结合运行结果，理解代码并回答问题：

> **问题1：** 这个范例中，`Window`和`Button`是组合关系吗?
>
> **问题2：** `bool has_button(const std::string &label) const;`被设计为私有。 思考并回答：
>
> （1）若将其改为公有接口，有何优点或风险?

> （2）设计类时，如何判断一个成员函数应为 public 还是 private？（可从“用户是否需要”、“是否仅为内部实现细节”、“是否易破坏对象状态”等角度分析。）
>
> **问题3：** `Button` 的接口 `const std::string& get_label() const;` 返回 `const std::string&`。对比以下两种接口设计在性能和安全性方面的差异并精炼陈述。
>
> 接口1：`const std::string& get_label() const;`
>
> 接口2：`const std::string get_label() const;`
>
> **问题4：** 把代码中所有 `xx.push_back(Button(xxx))` 改成 `xx.emplace_back(xxx)`，观察程序是否正常运行；查阅资料，回答两种写法的差别。

2. 实验任务2

验证性实验：用标准库模板类vector观察、理解深复制。实践、理解代码，回答问题。

task2.cpp

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

void test1();
void test2();
void output1(const std::vector<int> &v);
void output2(const std::vector<int> &v);
void output3(const std::vector<std::vector<int>>& v);

int main() {
    std::cout << "深复制验证1: 标准库vector<int>\n";
    test1();

    std::cout << "\n深复制验证2: 标准库vector<int>嵌套使用\n";
    test2();
}

void test1() {
    std::vector<int> v1(5, 42);
    const std::vector<int> v2(v1);

    std::cout << "**********拷贝构造后**********\n";
    std::cout << "v1: "; output1(v1);
    std::cout << "v2: "; output1(v2);

    v1.at(0) = -1;

    std::cout << "**********修改v1[0]后**********\n";
    std::cout << "v1: "; output1(v1);
    std::cout << "v2: "; output1(v2);
}

void test2() {
```

```cpp
    std::vector<std::vector<int>> v1{{1, 2, 3}, {4, 5, 6, 7}};
    const std::vector<std::vector<int>> v2(v1);

    std::cout << "**********拷贝构造后**********\n";
    std::cout << "v1: "; output3(v1);
    std::cout << "v2: "; output3(v2);

    v1.at(0).push_back(-1);

    std::cout << "**********修改v1[0]后**********\n";
    std::cout << "v1: \n";  output3(v1);
    std::cout << "v2: \n";  output3(v2);
}

// 使用xx.at()+循环输出vector<int>数据项
void output1(const std::vector<int> &v) {
    if(v.size() == 0) {
        std::cout << '\n';
        return;
    }

    std::cout << v.at(0);
    for(auto i = 1; i < v.size(); ++i)
        std::cout << ", " << v.at(i);
    std::cout << '\n';
}

// 使用迭代器+循环输出vector<int>数据项
void output2(const std::vector<int> &v) {
    if(v.size() == 0) {
        std::cout << '\n';
        return;
    }

    auto it = v.begin();
    std::cout << *it;

    for(it = v.begin()+1; it != v.end(); ++it)
        std::cout << ", " << *it;
    std::cout << '\n';
}

// 使用auto for分行输出vector<vector<int>>数据项
void output3(const std::vector<std::vector<int>>& v) {
    if(v.size() == 0) {
        std::cout << '\n';
        return;
    }

    for(auto &i: v)
        output2(i);
}
```

编译、运行程序，结合运行结果，理解代码并回答问题：

> **问题1**：测试模块1中这两行代码分别完成了什么构造？ `v1`、`v2` 各包含多少个值为 42 的数据项？
>
> ```
> std::vector<int> v1(5, 42);
> const std::vector<int> v2(v1);
> ```
>
> **问题2**：测试模块2中这两行代码执行后，`v1.size()`、`v2.size()`、`v1[0].size()`分别是多少？
>
> ```
> std::vector<std::vector<int>> v1{{1, 2, 3}, {4, 5, 6, 7}};
> const std::vector<std::vector<int>> v2(v1);
> ```
>
> **问题3**：测试模块1中，把`v1.at(0) = -1;`写成`v1[0] = -1;`能否实现同等效果？两种用法有何区别？
>
> **问题4**：测试模块2中执行`v1.at(0).push_back(-1);`后
>
> （1） 用以下两行代码，能否输出-1？为什么？
>
> ```
> std::vector<int> &r = v1.at(0);
> std::cout << r.at(r.size()-1);
> ```
>
> （2）r定义成用`const &`类型接收返回值，在内存使用上有何优势？有何限制？
>
> **问题5**：观察程序运行结果，反向分析、推断：
>
> （1） 标准库模板类`vector`的复制构造函数实现的是深复制还是浅复制？
>
> （2） `vector<T>::at()`接口思考： 当`v`是`vector<int>`时，`v.at(0)`返回值类型是什么？当`v`是`const vector<int>`时，`v.at(0)`返回值类型又是什么？据此推断`at()`是否必须提供**带 const 修饰**的重载版本？

3. 实验任务3

验证性实验：不使用标准库`vector`，自定义简化版`vectorInt`类，深度理解深复制。

> - 实现`vectorInt`类，支持：
>   - 默认构造`vectorInt()`
>   - 指定大小`vectorInt(n)`
>   - 指定大小及初值`vectorInt(n, value)`
>   - 拷贝构造、析构函数
>   - 成员函数：`size()`、`at()`、`assign()`、`begin()`/`end()`（含`const`重载）
> - 编写普通函数：
>   - `output1()`：用`at()`遍历输出
>   - `output2()`：用迭代器遍历输出
> - 通过测试模块观察：
>   - 拷贝构造后修改原对象，副本是否变化
>   - `assign()`后两对象是否完全独立

代码组织：

- vectorInt.hpp　　vectorInt类定义

- task3.cpp　　普通函数、测试模块、main

vectorInt.hpp

```cpp
#pragma once

#include <iostream>

// 动态int数组对象类
class vectorInt{
public:
    vectorInt();
    vectorInt(int n_);
    vectorInt(int n_, int value);
    vectorInt(const vectorInt &vi);
    ~vectorInt();

    int size() const;
    int& at(int index);
    const int& at(int index) const;
    vectorInt& assign(const vectorInt &vi);

    int* begin();
    int* end();
    const int* begin() const;
    const int* end() const;

private:
    int n;     // 当前数据项个数
    int *ptr;  // 数据区
};

vectorInt::vectorInt():n{0}, ptr{nullptr} {
}

vectorInt::vectorInt(int n_): n{n_}, ptr{new int[n]} {
}

vectorInt::vectorInt(int n_, int value): n{n_}, ptr{new int[n_]} {
    for(auto i = 0; i < n; ++i)
        ptr[i] = value;
}

vectorInt::vectorInt(const vectorInt &vi): n{vi.n}, ptr{new int[n]} {
    for(auto i = 0; i < n; ++i)
        ptr[i] = vi.ptr[i];
}

vectorInt::~vectorInt() {
    delete [] ptr;
}
```

```cpp

int vectorInt::size() const {
    return n;
}

const int& vectorInt::at(int index) const {
    if(index < 0 || index >= n) {
        std::cerr << "IndexError: index out of range\n";
        std::exit(1);
    }

    return ptr[index];
}

int& vectorInt::at(int index) {
    if(index < 0 || index >= n) {
        std::cerr << "IndexError: index out of range\n";
        std::exit(1);
    }

    return ptr[index];
}

vectorInt& vectorInt::assign(const vectorInt &vi) {
    if(this == &vi)
        return *this;

    int *ptr_tmp;
    ptr_tmp = new int[vi.n];
    for(int i = 0; i < vi.n; ++i)
        ptr_tmp[i] = vi.ptr[i];

    delete[] ptr;
    n = vi.n;
    ptr = ptr_tmp;
    return *this;
}

int* vectorInt::begin() {
    return ptr;
}

int* vectorInt::end() {
    return ptr+n;
}

const int* vectorInt::begin() const {
    return ptr;
}

const int* vectorInt::end() const {
    return ptr+n;
}
```

task3.cpp

```cpp
#include "vectorInt.hpp"
#include <iostream>

void test1();
void test2();
void output1(const vectorInt &vi);
void output2(const vectorInt &vi);

int main() {
    std::cout << "测试1: \n";
    test1();

    std::cout << "\n测试2: \n";
    test2();
}

void test1() {
    int n;
    std::cout << "Enter n: ";
    std::cin >> n;

    vectorInt x1(n);
    for(auto i = 0; i < n; ++i)
        x1.at(i) = (i+1)*10;
    std::cout << "x1: ";  output1(x1);

    vectorInt x2(n, 42);
    vectorInt x3(x2);
    x2.at(0) = -1;
    std::cout << "x2: ";  output1(x2);
    std::cout << "x3: ";  output1(x3);
}

void test2() {
    const vectorInt  x(5, 42);
    vectorInt y;

    y.assign(x);

    std::cout << "x: ";  output2(x);
    std::cout << "y: ";  output2(y);
}

// 使用xx.at()+循环输出vectorInt对象数据项
void output1(const vectorInt &vi) {
    if(vi.size() == 0) {
        std::cout << '\n';
        return;
    }

    std::cout << vi.at(0);
```

```
    for(auto i = 1; i < vi.size(); ++i)
        std::cout << ", " << vi.at(i);
    std::cout << '\n';
}

// 使用迭代器+循环输出vectorInt对象数据项
void output2(const vectorInt &vi) {
    if(vi.size() == 0) {
        std::cout << '\n';
        return;
    }

    auto it = vi.begin();
    std::cout << *it;

    for(it = vi.begin()+1; it != vi.end(); ++it)
        std::cout << ", " << *it;
    std::cout << '\n';
}
```

编译、运行程序，结合运行结果，理解代码并回答问题：

> **问题1：**当前验证性代码中，`vectorInt`接口`assign`实现是安全版本。如果把`assign`实现改成版本2，逐条指出版本 2存在的安全隐患和缺陷。（提示：对比两个版本，找出差异化代码，加以分析）
>
> ```
> // 版本2
> vectorInt& vectorInt::assign(const vectorInt &vi) {
>  delete[] ptr;
>
>  n = vi.n;
>  ptr = new int[n];
>
>  for(int i = 0; i < n; ++i)
>      ptr[i] = vi.ptr[i];
>
>  return *this;
> }
> ```
>
> **问题2：**当前验证性代码中，重载接口`at`内部代码完全相同。若把非 const 版本改成如下实现，可消除重复并遵循“最小化接口”原则（未来如需更新接口，只更新const接口，另一个会同步）。
>
> ```
> int& vectorInt::at(int index) {
>  return const_cast<int&>(static_cast<const vectorInt*>(this)->at(index));
> }
> ```
>
> 查阅资料，回答：
>
> （1）`static_cast<const vectorInt*>(this)`的作用是什么？转换前后`this`的类型分别是什么？转换目的？
>
> （2）`const_cast<int&>`的作用是什么？转换前后的返回类型分别是什么？转换目的？

问题3： `vectorInt`类封装了`begin()`和`end()`的const/非const接口。

（1）以下代码片段，分析编译器如何选择重载版本，并总结这两种重载分别适配什么使用场景。

```
vectorInt v1(5);
const vectorInt v2(5);

auto it1 = v1.begin();     // 调用哪个版本？
auto it2 = v2.begin();     // 调用哪个版本？
```

（2）拓展思考（选答*）：标准库迭代器本质上是指针的封装。`vectorInt`直接返回原始指针作为迭代器，这种设计让你对迭代器有什么新的理解？

问题4： 以下两个构造函数及`assign`接口实现，都包含内存块的赋值和复制操作。使用算法库`<algorithm>`改成如下写法是否可以？回答这3行更新代码的功能。

```
vectorInt::vectorInt(int n_, int value): n{n_}, ptr{new int[n_]} {
 std::fill_n(ptr, n, value);   // 更新
}

vectorInt::vectorInt(const vectorInt &vi): n{vi.n}, ptr{new int[n]} {
 std::copy_n(vi.ptr, vi.n, ptr); // 更新
}

vectorInt& vectorInt::assign(const vectorInt &vi) {
 if(this == &vi)
     return *this;

 int *ptr_tmp;
 ptr_tmp = new int[vi.n];
 std::copy_n(vi.ptr, vi.n, ptr_tmp); // 更新
 delete[] ptr;

 n = vi.n;
 ptr = ptr_tmp;
 return *this;
}
```

4. 实验任务4

设计性实验：用原始指针实现动态矩阵类`Matrix`，理解资源管理与深复制。

根据给定声明`matrix.hpp`，在`matrix.cpp`中实现类，使其通过`task4.cpp`测试。

要求：

- 连续内存，支持矩阵/方阵构造、深拷贝、元素访问、按行打印；
- 越界或大小不符立即报错并终止；
- 不得修改接口，不得新增公开成员。

matrix.hpp

```cpp
#pragma once

// 类Matrix声明
class Matrix {
public:
    Matrix(int rows_, int cols_, double value = 0); // 构造rows_*cols_矩阵对象, 初值value
    Matrix(int rows_, double value = 0);    // 构造rows_*rows_方阵对象, 初值value
    Matrix(const Matrix &x);    // 深复制
    ~Matrix();

    void set(const double *pvalue, int size);   // 按行复制pvalue指向的数据, 要求size=rows*cols,否则报错退出
    void clear();   // 矩阵对象数据项置0

    const double& at(int i, int j) const;   // 返回矩阵对象索引(i,j)对应的数据项const引用（越界则报错后退出）
    double& at(int i, int j);   // 返回矩阵对象索引(i,j)对应的数据项引用（越界则报错后退出）

    int rows() const;   // 返回矩阵对象行数
    int cols() const;   // 返回矩阵对象列数

    void print() const;   // 按行打印数据

private:
    int n_rows;      // 矩阵对象内元素行数
    int n_cols;       // 矩阵对象内元素列数
    double *ptr;    // 数据区
};
```

task4.cpp

```cpp
#include <iostream>
#include <cstdlib>
#include "matrix.hpp"

void test1();
void test2();
void output(const Matrix &m, int row_index);

int main() {
    std::cout << "测试1: \n";
    test1();

    std::cout << "\n测试2: \n";
    test2();
}

void test1() {
    double x[1000] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    int n, m;
    std::cout << "Enter n and m: ";
```

```
    std::cin >> n >> m;

    Matrix m1(n, m);     // 创建矩阵对象m1, 大小n×m
    m1.set(x, n*m);      // 用一维数组x的值按行为矩阵m1赋值

    Matrix m2(m, n);     // 创建矩阵对象m2, 大小m×n
    m2.set(x, m*n);      // 用一维数组x的值按行为矩阵m1赋值

    Matrix m3(n);        // 创建一个n×n方阵对象
    m3.set(x, n*n);      // 用一维数组x的值按行为矩阵m3赋值

    std::cout << "矩阵对象m1: \n";   m1.print();
    std::cout << "矩阵对象m2: \n";   m2.print();
    std::cout << "矩阵对象m3: \n";   m3.print();
}

void test2() {
    Matrix m1(2, 3, -1);
    const Matrix m2(m1);

    std::cout << "矩阵对象m1: \n";   m1.print();
    std::cout << "矩阵对象m2: \n";   m2.print();

    m1.clear();
    m1.at(0, 0) = 1;

    std::cout << "m1更新后: \n";
    std::cout << "矩阵对象m1第0行 "; output(m1, 0);
    std::cout << "矩阵对象m2第0行: "; output(m2, 0);
}

// 输出矩阵对象row_index行所有元素
void output(const Matrix &m, int row_index) {
    if(row_index < 0 || row_index > m.rows()) {
        std::cerr << "IndexError: row index out of range\n";
        std::exit(1);
    }

    std::cout << m.at(row_index, 0);
    for(int j = 1; j < m.cols(); ++j)
        std::cout << ", " << m.at(row_index, j);
    std::cout << '\n';
}
```

`Matrix`类正确实现后，预期测试效果如下：

```
测试1:
Enter n and m: 2 3
矩阵对象m1:
1, 2, 3
4, 5, 6
矩阵对象m2:
1, 2
3, 4
5, 6
矩阵对象m3:
1, 2
3, 4

测试2:
矩阵对象m1:
-1, -1, -1
-1, -1, -1
矩阵对象m2:
-1, -1, -1
-1, -1, -1
m1更新后:
矩阵对象m1第0行 1, 0, 0
矩阵对象m2第0行: -1, -1, -1
```

5. 实验任务5

设计性实验：综合使用类的组合、自定义类、C++标准库实现简化版通讯录，实现联系人增/删/查/显操作。

联系人`Contact`类已封装，通讯录`ContactBook`类接口和测试代码已提供。阅读、理解代码，补足`ContactBook`内部工具函数，使程序能正确运行并符合预期。

- **任务1**

  实现私有工具`int ContactBook::index(const std::string& name) const;`

  功能：在通讯录中查找指定姓名的联系人 （姓名查找区分大小写）

  返回：如找到则返回该联系人在`contacts`中的索引，否则，返回-1

- **任务2**

  实现私有工具`void ContactBook::sort();`

  功能：把 `contacts` **按姓名字典序升序**排列。

- **约束**

  只在ContactBook.hpp中补足私有工具实现

  禁止修改其他文件，**保持接口与输出格式一致。**

代码正确补足后，预期测试如下：

```
1. add contacts
Bob add successfully.
Alice add successfully.
Linda add successfully.
Alice already exists. fail to add!

2. display contacts
There are 3 contacts.
Alice, 17300886371
Bob, 18199357253
Linda, 18184538072

3. find contacts
Bob, 18199357253
David not found!

4. remove contact
Bob remove successfully.
David not found, fail to remove!
```

contact.hpp

```cpp
#pragma once

#include <iostream>
#include <string>

// 联系人类
class Contact {
public:
    Contact(const std::string &name_, const std::string &phone_);

    const std::string &get_name() const;
    const std::string &get_phone() const;
    void display() const;

private:
   std::string name;    // 必填项
   std::string phone;   // 必填项
};

Contact::Contact(const std::string &name_, const std::string &phone_):name{name_},
phone{phone_} {
}

const std::string& Contact::get_name() const {
    return name;
}

const std::string& Contact::get_phone() const {
    return phone;
}

void Contact::display() const {
```

```
    std::cout << name << ", " << phone;
}
```

contactBook.hpp

```
# pragma  once

#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include "contact.hpp"

// 通讯录类
class ContactBook {
public:
    void add(const std::string &name, const std::string &phone); // 添加联系人
    void remove(const std::string &name); // 移除联系人
    void find(const std::string &name) const; // 查找联系人
    void display() const; // 显示所有联系人
    size_t size() const;

private:
    int index(const std::string &name) const;  // 返回联系人在contacts内索引，如不存在，返回-1
    void sort(); // 按姓名字典序升序排序通讯录

private:
    std::vector<Contact> contacts;
};

void ContactBook::add(const std::string &name, const std::string &phone) {
    if(index(name) == -1) {
        contacts.push_back(Contact(name, phone));
        std::cout << name << " add successfully.\n";
        sort();
        return;
    }

    std::cout << name << " already exists. fail to add!\n";
}

void ContactBook::remove(const std::string &name) {
    int i = index(name);

    if(i == -1) {
        std::cout << name << " not found, fail to remove!\n";
        return;
    }

    contacts.erase(contacts.begin()+i);
    std::cout << name << " remove successfully.\n";
}
```

```cpp

void ContactBook::find(const std::string &name) const {
    int i = index(name);

    if(i == -1) {
        std::cout << name << " not found!\n";
        return;
    }

    contacts[i].display();
    std::cout << '\n';
}

void ContactBook::display() const {
    for(auto &c: contacts) {
        c.display();
        std::cout << '\n';
    }
}

size_t ContactBook::size() const {
    return contacts.size();
}

// 待补足1：int index(const std::string &name) const;实现
// 返回联系人在contacts内索引；如不存在，返回-1



// 待补足2：void ContactBook::sort();实现
// 按姓名字典序升序排序通讯录
```

task5.cpp

```cpp
#include "contactBook.hpp"

void test() {
    ContactBook contactbook;

    std::cout << "1. add contacts\n";
    contactbook.add("Bob", "18199357253");
    contactbook.add("Alice", "17300886371");
    contactbook.add("Linda", "18184538072");
    contactbook.add("Alice", "17300886371");

    std::cout << "\n2. display contacts\n";
    std::cout << "There are " << contactbook.size() << " contacts.\n";
    contactbook.display();

    std::cout << "\n3. find contacts\n";
    contactbook.find("Bob");
    contactbook.find("David");

```

```
    std::cout << "\n4. remove contact\n";
    contactbook.remove("Bob");
    contactbook.remove("David");
}

int main() {
    test();
}
```

四、实验结论

1. 实验任务1

此部分书写内容：

- 分别给出button.hpp, window.hpp, task1.cpp源码，以及，运行测试截图
- 回答问题

> **问题1**：这个范例中，`Window`和`Button`是组合关系吗?
>
> **问题2**：`bool has_button(const std::string &label) const;`被设计为私有。 思考并回答：
>
> （1）若将其改为公有接口，有何优点或风险?
>
> （2）设计类时，如何判断一个成员函数应为 public 还是 private? （可从“用户是否需要”、“是否仅为内部实现细节”、“是否易破坏对象状态”等角度分析。）
>
> **问题3**：`Button`的接口`const std::string& get_label() const;`返回`const std::string&`。简要说明以下两种接口设计在性能和安全性方面的差异。
>
> 接口1：`const std::string& get_label() const;`
>
> 接口2：`const std::string get_label() const;`
>
> **问题4**：把代码中所有`xx.push_back(Button(xxx))`改成`xx.emplace_back(xxx)`，观察程序是否正常运行；查阅资料，回答两种写法的差别。

2. 实验任务2

此部分书写内容：

- 给出task2.cpp源码，以及，运行测试截图
- 回答问题

> **问题1**：测试模块1中这两行代码分别完成了什么构造? `v1`、`v2`各包含多少个值为 42 的数据项?
>
> ```
> std::vector<int> v1(5, 42);
> const std::vector<int> v2(v1);
> ```
>
> **问题2**：测试模块2中这两行代码执行后，`v1.size()`、`v2.size()`、`v1[0].size()`分别是多少?
>
> ```
> std::vector<std::vector<int>> v1{{1, 2, 3}, {4, 5, 6, 7}};
> const std::vector<std::vector<int>> v2(v1);
> ```

> **问题3**：测试模块1中，把`v1.at(0) = -1;`写成`v1[0] = -1;`能否实现同等效果？两种用法有何区别？
>
> **问题4**：测试模块2中执行`v1.at(0).push_back(-1);`后
>
> （1） 用以下两行代码，能否输出-1？为什么？
>
> ```
> std::vector<int> &r = v1.at(0);
> std::cout << r.at(r.size()-1);
> ```
>
> （2）r定义成用`const &`类型接收返回值，在内存使用上有何优势？有何限制？
>
> **问题5**：观察程序运行结果，反向分析、推断：
>
> （1） 标准库模板类`vector`的复制构造函数实现的是深复制还是浅复制？
>
> （2） `vector<T>::at()`接口思考： 当`v`是`vector<int>`时，`v.at(0)`返回值类型是什么？当`v`是`const vector<int>`时，`v.at(0)`返回值类型又是什么？据此推断`at()`是否必须提供**带 const 修饰**的重载版本？

3. 实验任务3

此部分书写内容：

- 分别给出vectorInt.hpp, task3.cpp源码，及，运行测试截图
- 回答问题

> **问题1**：当前验证性代码中，`vectorInt`接口`assign`实现是安全版本。如果把`assign`实现改成版本2，逐条指出版本 2存在的安全隐患和缺陷。（提示：对比两个版本，找出差异化代码，加以分析）
>
> ```
> 1  // 版本2
> 2  vectorInt& vectorInt::assign(const vectorInt &vi) {
> 3   delete[] ptr;
> 4
> 5   n = vi.n;
> 6   ptr = new int[n];
> 7
> 8   for(int i = 0; i < n; ++i)
> 9       ptr[i] = vi.ptr[i];
> 10
> 11  return *this;
> 12 }
> ```
>
> **问题2**：当前验证性代码中，重载接口`at`内部代码完全相同。若把非 const 版本改成如下实现，可消除重复并遵循“最小化接口”原则（未来如需更新接口，只更新const接口，另一个会同步）。
>
> ```
> 1  int& vectorInt::at(int index) {
> 2   return const_cast<int&>(static_cast<const vectorInt*>(this)->at(index));
> 3  }
> ```
>
> 查阅资料，回答：
>
> （1）`static_cast<const vectorInt*>(this)`的作用是什么？转换前后`this`的类型分别是什么？转换目的？

（2）`const_cast<int&>`的作用是什么？转换前后的返回类型分别是什么？转换目的？

问题3：`vectorInt`类封装了`begin()`和`end()`的const/非const接口。

（1）以下代码片段，分析编译器如何选择重载版本，并总结这两种重载分别适配什么使用场景。

```
vectorInt v1(5);
const vectorInt v2(5);

auto it1 = v1.begin();     // 调用哪个版本？
auto it2 = v2.begin();     // 调用哪个版本？
```

（2）拓展思考（选答*）：标准库迭代器本质上是指针的封装。`vectorInt`直接返回原始指针作为迭代器，这种设计让你对迭代器有什么新的理解？

问题4：以下两个构造函数及`assign`接口实现，都包含内存块的赋值/复制操作。使用算法库`<algorithm>`改写是否可以？回答这3行更新代码的功能。

```
vectorInt::vectorInt(int n_, int value): n{n_}, ptr{new int[n_]} {
    std::fill_n(ptr, n, value);   // 更新1
}

vectorInt::vectorInt(const vectorInt &vi): n{vi.n}, ptr{new int[n]} {
    std::copy_n(vi.ptr, vi.n, ptr); // 更新2
}

vectorInt& vectorInt::assign(const vectorInt &vi) {
    if(this == &vi)
        return *this;

    int *ptr_tmp;
    ptr_tmp = new int[vi.n];
    std::copy_n(vi.ptr, vi.n, ptr_tmp); // 更新3
    delete[] ptr;

    n = vi.n;
    ptr = ptr_tmp;
    return *this;
}
```

4. 实验任务4

此部分书写内容：

- 分别给出程序matrix.hpp,matrix.cpp, task4.cpp源码，及，运行测试截图
- 拓展思考
  若把内部存储`double *ptr;`换成`std::vector<double>`并保持原接口：
  - 无需手写复制构造/析构等，自动获得 RAII 与异常安全
  - 仍保持连续内存，性能无损，代码量精简
  - 对比两种实现，体会“零成本抽象”

```cpp
class Matrix {
public:

    // 接口保持不变（略）

private:
    int n_rows;
    int n_cols;
    std::vector<double>  data;  // 替代原始指针
};
```

5. 实验任务5

此部分书写内容：

- 分别给出程序contact.hpp,contactBook.hpp（补足后版本）, task5.cpp源码，及，运行测试截图
- 拓展思考
  这个简易通讯录模拟练习功能有限，要让其接近真实手机通讯录，后续迭代：
  - 修改联系人功能
  - 数据有效性校验
  - 异常处理
  - 数据持久化存储
  - 高阶功能：联系人分组管理、模糊检索， ...

  思考如何基于现有类结构逐步扩展，保持接口兼容与代码可维护性。

五、实验总结(选)

此部分书写内容：（可以包括但不限以下内容）

- 提炼本次实验的核心收获与思考
- 记录新发现、真实感受与待解疑问
- 任意想补充的反馈或分享

六、实验文档提交要求

- **提交形式**

  请将实验文档以**在线技术博客**形式提交，发布后请在**相应实验作业**页面提交链接。

  博客开通与提交方法详见：**「博客园博客开通及发布步骤_2024级_OOP.pdf」**
- **撰写内容**

  必写：「四、实验结论」

  选写：「五、实验总结」

  要求：内容完整，表述简练，逻辑清晰，图文整洁

- **截止时间**

  通常为实验课后一周内，具体**截止时间**以博客园发布的作业通知为准。 （**提交时间**以博客园文章页面显示的日期为准）

  截止后，欢迎通过教学班级博客主页相互评阅。

  代码正确、风格规范、图文清晰、总结到位的博客，欢迎点赞推荐，一起进步！

七、博客园编辑器使用补充说明

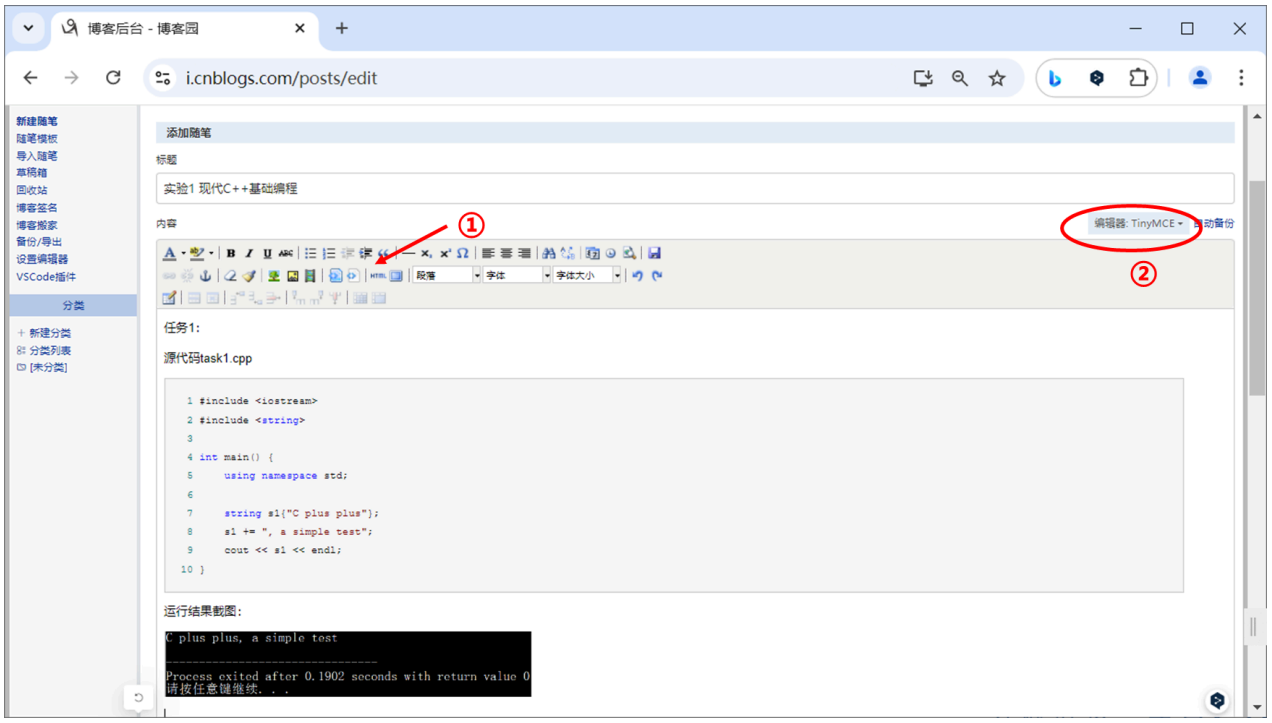
针对博客园平台使用的默认编辑器，插入代码和图片后，文字难以插入编辑的状况：

- 解决方案1

  在插入代码和图片前，自己先编辑好文字，或者，用回车，预留好空白行

- 解决方案2

  在编辑界面，点开页面对应的html源代码，在你想插入文字的地方加上html标签`<p></p>`（图中①位置），然后再回到编辑界面，就会看到空白行。可以在那里编辑。



另，博客园平台设置页面支持更改默认编辑器（图中②位置）。目前，默认编辑器是markdown编辑器。

如果你暂不熟悉markdown基础用法，可切换成TinyMCE编辑器，其界面和使用风格接近office系列。

markdown编辑器使用参考：https://www.runoob.com/markdown/md-tutorial.html