以项目的方式学QT开发

P1 QT介绍

1.1 QT简介

Qt 是一个跨平台的应用程序和用户界面框架，用于开发图形用户界面（GUI）应用程序以及命令行工具。它最初由挪威的 Trolltech （奇趣科技）公司开发，现在由 Qt Company 维护，2020年12月8日发布QT6。Qt 使用 C++ 语言编写，支持多种编程语言通过绑定进行使用。

对于许多开发者和小型企业来说，Qt 的开源版提供了一个强大且灵活的开发框架，而对于需要额外支持和专有功能的大型企业或具有特定需求的项目，商业版则提供了所需的服务和资源。

- Qt 商业版

  商业版提供专有许可，需要购买许可证来使用。这适用于希望在不共享源代码的情况下开发商业软件的公司和开发人员

- QT免费开源版

  开源版根据 GNU Lesser General Public License (LGPL) 和 GNU General Public License (GPL) 发布。这意味着用户可以免费使用 Qt，但必须遵守特定的开源许可条款

- QT主要历史版本

版本	发布年份	关键特性
Qt 1.x	1996	初始发布，专注于 X11 平台
Qt 2.x	1999	引入了对 Microsoft Windows 的支持
Qt 3.x	2001	添加了许多新功能，包括网络和 XML 支持
Qt 4.x	2005	重大改进，增强了跨平台支持和图形视图框架
Qt 5.x	2012	专注于现代硬件的性能，引入了 QML 和 Qt Quick 用于开发流畅的动画和触摸界面
Qt 6.x	2020	进一步增强了性能和功能，针对未来的软件开发趋势进行了优化，包括对 3D 图形的支持

学习者学习QT5和QT6都是可以的，无论选择哪个版本，Qt的基本概念和理念在各个版本之间是相通的，因此你可以相对轻松地转换到其他版本。本次我们基于QT5学习

- 成熟和稳定性

  Qt 5已经存在了一段时间，经过了多个版本的迭代和改进。它在很多项目中被广泛使用，证明了其成熟性和稳定性。这对于在大型项目或生产环境中使用Qt的开发者来说是一个优势。

- 丰富的文档和社区支持

  Qt 5有大量的文档和社区支持。你可以轻松找到各种教程、示例和解决方案，这对于初学者来说是非常宝贵的

- 广泛的应用领域

  Qt 5有大量的文档和社区支持。你可以轻松找到各种教程、示例和解决方案，这对于初学者来说是非常宝贵的。

1.2 QT安装

1.2.1 Windows QT安装

- 下载windowsQT安装包

本教程使用的QT版本是：https://download.qt.io/archive/qt/5.12/5.12.9/ 本教程的安装包放在百度网盘供大家获取。

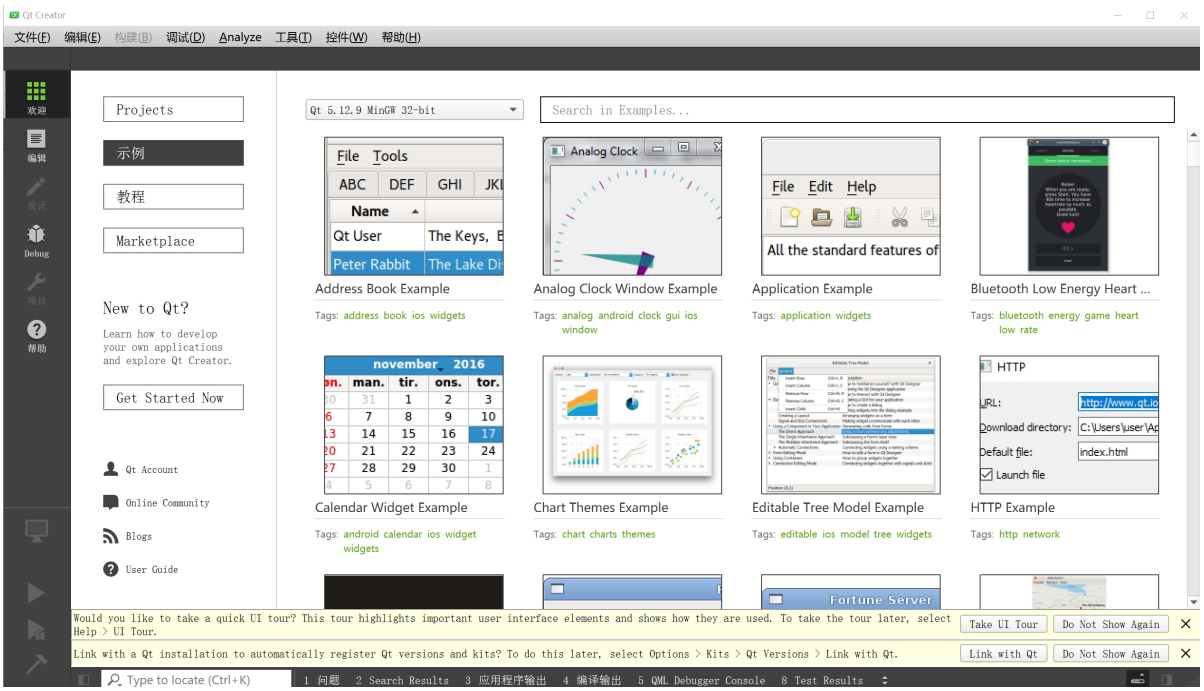
Name	Last modified	Size	Metadata
Parent Directory		-	
submodules/	16-Jun-2020 07:04	-	
single/	16-Jun-2020 07:04	-	
qt-opensource-windows-x86-5.12.9.exe	16-Jun-2020 18:07	3.7G	Details
qt-opensource-mac-x64-5.12.9.dmg	16-Jun-2020 14:49	2.7G	Details
qt-opensource-linux-x64-5.12.9.run	16-Jun-2020 14:48	1.3G	Details
md5sums.txt	16-Jun-2020 18:14	207	Details

- QT安装

如果没有梯子，大家登录QT官网可能会失败，这里可以**不需要QT账号**，直接**离线**安装，所以要断开网络。

选择windows底下的编译工具，QT源代码，QT的绘图模块及QT的虚拟键盘

安装完成后打开



1.2.2 QT Creator 使用基本介绍

- 课程录屏展示-创建并运行第一个QT项目
- 课程录屏展示-创建并运行第一个C++项目
- 课程录屏展示-QT Creator的界面介绍
- QT Creator常用的快捷键介绍

功能	快捷键	中文说明
撤销	CTRL + Z	撤销最近的操作
重做	CTRL + Y	重做最近的撤销操作
复制	CTRL + C	复制选中内容
粘贴	CTRL + V	粘贴内容
复制行向下	CTRL + ALT + DOWN	将当前行复制到下一行
复制行向上	CTRL + ALT + UP	将当前行复制到上一行
运行	CTRL + R	运行当前项目
返回编辑模式	ESCAPE	返回到编辑状态
切换当前文件	CTRL + TAB	在打开的文件间切换
切换声明和定义	F2	在代码的声明与定义间切换
切换头文件和源文件	F4	在头文件和源文件间切换
开始调试	F5	启动调试
停止调试	SHIFT + F5	停止当前的调试
构建当前项目	CTRL + B	构建当前打开的项目

功能	快捷键	中文说明
构建所有项目	CTRL + SHIFT + B	构建所有项目
新建文件或项目	CTRL + N	创建新文件或项目
打开文件或项目	CTRL + O	打开现有文件或项目
保存当前文件	CTRL + S	保存当前编辑的文件
保存所有文件	CTRL + SHIFT + S	保存所有打开的文件
关闭当前文件	CTRL + W	关闭当前文件
关闭所有文件	CTRL + SHIFT + W	关闭所有打开的文件
退出QT Creator	CTRL + Q	退出QT Creator
位置后退	ALT+Left	光标位置回退

P2 C++基础

C和C++之间的关系是紧密且复杂的。C++最初是作为C语言的一个扩展开发的，目的是在不放弃C的强大功能和效率的同时，增加对象导向编程、泛型编程和其他一些特性。下面是C和C++之间主要的关系和区别：

1. **兼容性**：C++在很大程度上是与C兼容的。这意味着许多C程序可以在C++编译器中编译并运行，尽管可能需要一些小的修改。
2. **面向对象编程（OOP）**：C++引入了面向对象编程。它允许使用类和对象，而C是一个过程性语言，不支持这些概念，或者说支持的不好，麻烦。
3. **模板**：C++支持模板，这是一种允许程序员编写与数据类型无关的代码的功能。C没有这个功能。
4. **标准库**：C++有一个更丰富的标准库，包括STL（标准模板库），这为数据结构和算法提供了广泛的支持。而C的标准库相对较小。
5. **类型检查**：C++比C提供更严格的类型检查。这意味着某些在C中可行但可能导致错误的代码在C++中可能无法编译。
6. **异常处理**：C++支持异常处理，这是一种处理程序运行时错误的机制。C没有内置的异常处理机制。
7. **命名空间**：C++引入了命名空间，这有助于防止名称冲突。C没有这个概念。

2.1 命名空间

2.1.1 命名空间作用

创建自己的命名空间是 C++ 中组织代码的一种好方法，特别是在开发大型项目或库时。命名空间可以帮助你避免名称冲突，并且清晰地组织代码。

`std` 是 C++ 标准库的命名空间。它是一个定义在 C++ 标准库中的所有类、函数和变量的命名空间。

我们新建一个QTCreator的C++工程，默认生成的代码

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello World!" << endl;
    return 0;
}
```

在 C++ 中，如果你想使用标准库中的任何类、函数或对象，你通常有两种选择：

1. **使用 `std::` 前缀**：这是最常见的方式，它明确指定了你正在使用的是位于 `std` 命名空间中的元素。

```cpp
std::cout << "Hello, world!" << std::endl;
```

2. **使用 `using namespace std;`**：这允许你在不显式指定 `std::` 的情况下使用 `std` 命名空间中的所有元素。

```cpp
using namespace std;
cout << "Hello, world!" << endl;
```

std包含的内容

`std` 命名空间包含了许多类、函数和对象，例如：

- 输入输出库（如 `std::cout`, `std::cin`, `std::endl`）
- 容器类（如 `std::vector`, `std::map`, `std::set`）
- 字符串类（`std::string`）
- 异常类（`std::exception` 和相关子类）
- 算法（如 `std::sort`, `std::find`）
- 实用工具（如 `std::pair`, `std::tuple`）
- 其他许多功能

使用建议

- 对于小型代码或示例代码，使用 `using namespace std;` 通常是安全的。
- 对于大型项目或库，建议显式地使用 `std::` 前缀，以避免潜在的名称冲突，并提高代码的可读性和可维护性。

`std` 命名空间是 C++ 编程的基础部分，理解和正确使用它对于编写健壮和高效的 C++ 代码至关重要。

2.1.2 自定义命名空间

定义命名空间

假设我们要创建一个命名空间来包含与圆形相关的功能。我们可以命名这个命名空间为 `Cir`：

```
```

```
#ifndef CIR_H
#define CIR_H

namespace Cir {
    const double PI = 3.141592653;
    double areaOfCircle(double radius){
        return PI*radius*radius;
    }
    double lenthOfCircle(double radius){
        return 2*PI*radius;
    }
}

#endif // CIR_H
```

在这个头文件中，我们定义了一个名为 `Cir` 的命名空间，其中包含了计算圆的面积和周长的函数，以及圆周率常量 `PI`。

使用命名空间

在另一个文件中，我们可以使用这个命名空间中定义的函数和常量：

```
#include "cir.h"
#include <stdio.h>

int main()
{
    double  radius = 5;
    printf("半径为%f的圆，周长为%f,面积为%f\n",
           radius,Cir::lenthOfCircle(radius), Cir::areaOfCircle(radius));
    return 0;
}
```

在 `main.cpp` 中，我们首先包含了定义 `Cir` 命名空间的头文件。然后，我们可以使用 `Cir::` 前缀来访问该命名空间中的函数和常量。

通过使用自定义命名空间，你可以有效地组织你的代码，并减少不同库之间的名称冲突。这在大型项目和团队协作中尤其重要。

```
#include "cir.h"
#include <stdio.h>

using namespace Cir;
int main()
{
    double  radius = 5;
    printf("半径为%f的圆，周长为%f,面积为%f\n",
           radius,lenthOfCircle(radius), areaOfCircle(radius));
    return 0;
}
```

2.2 从C语言快速入门

2.2.1 输入输出

C++ 中的输入和输出（I/O）主要是通过标准库中的输入输出流来实现的。最常用的是 iostream 库，它提供了用于输入和输出的基本流类，包括 `cin`、`cout`、`cerr` 和 `clog`。

标准输出流（`cout`）

- `cout` 代表标准输出流，通常用于向屏幕输出数据。
- 使用操作符 `<<`（插入操作符）向 `cout` 发送数据。
- 例如，`std::cout << "Hello, world!" << std::endl;` 会在屏幕上打印 "Hello, world!" 并换行。

标准输入流（`cin`）

- `cin` 代表标准输入流，用于从键盘接收数据。
- 使用操作符 `>>`（提取操作符）从 `cin` 提取数据。
- 例如，`int x; std::cin >> x;` 会从用户那里读取一个整数并存储在变量 `x` 中。

标准错误流（`cerr`）和标准日志流（`clog`）

- `cerr` 用于输出错误消息。与 `cout` 不同，`cerr` 不是缓冲的，这意味着它会立即输出。
- `clog` 类似于 `cerr`，但它是缓冲的。它通常用于记录错误和日志信息。

示例代码

下面是一个展示如何使用这些基本流的简单示例：

```
#include <iostream>

int main() {
    // 使用 cout 输出
    std::cout << "Enter a number: ";

    // 使用 cin 输入
    int num;
    std::cin >> num;
    // 输出结果
    std::cout << "You entered: " << num << std::endl;
    std::clog << "Logging: user entered a number." << std::endl;

    return 0;
}
```

2.2.2 基本变量类型

C++ 基本数据类型整理成表格。以下是一个表格，展示了不同的基本数据类型及其一般用途和大小范围：

和C语言类似。

数据类型	描述	大小（通常情况下）	用途
`int`	整型	至少 16 位	存储整数
`short int`	短整型	至少 16 位	存储较小的整数
`long int`	长整型	至少 32 位	存储较大的整数
`long long int`	更长的整型	至少 64 位	存储非常大的整数
`unsigned int`	无符号整型	同 `int`	存储非负整数
`float`	单精度浮点类型	32 位	存储小数，精度约为 6-7 位小数
`double`	双精度浮点类型	64 位	存储小数，精度约为 15-16 位小数
`long double`	扩展精度浮点类型	80 位或更多	存储小数，提供比 `double` 更高的精度
`char`	字符型	8 位	存储单个字符或小整数
`unsigned char`	无符号字符型	8 位	存储较大的字符或作为字节使用
`signed char`	有符号字符型	8 位	明确作为带符号的字符或小整数使用
`bool`	布尔型	通常为 8 位	存储真值 `true` 或假值 `false` C语言C99以上支持
wchar_t	宽字符类型	通过为16位或32位	存储中文或者unicode

宽字符的用法

```
#include <iostream>
#include <locale>
#include <wchar.h>

int main() {
    // 设置本地化以支持宽字符
    std::setlocale(LC_ALL, "");

    // 使用 wchar_t 类型定义一个宽字符串
    wchar_t wstr[] = L"你好，世界！";

    // 在 C++ 中打印宽字符串
    std::wcout << wstr << std::endl;

    return 0;
}
```

在 C++ 中，`<climits>`（或在 C 中是 `<limits.h>`）是一个标准头文件，提供了关于整型限制的信息。这个头文件中定义了各种整型数据类型的属性，如最大值、最小值等。使用这些信息可以帮助你了解在特定编译器和平台上各种数据类型的大小和范围。

如何使用 `<climits>`

要使用 `<climits>` 中定义的常量，你首先需要包含这个头文件：

```
#include <climits>
```

然后，你可以使用它提供的各种常量，例如：

- `INT_MAX`：`int` 类型的最大值。
- `INT_MIN`：`int` 类型的最小值。
- `UINT_MAX`：`unsigned int` 类型的最大值。
- `LONG_MAX`：`long int` 类型的最大值。
- `LONG_MIN`：`long int` 类型的最小值。
- `LLONG_MAX`：`long long int` 类型的最大值。
- `LLONG_MIN`：`long long int` 类型的最小值。

示例代码

下面是一个简单的示例，展示了如何使用 `<climits>` 中的值：

```
#include <iostream>
#include <climits>

int main() {
    std::cout << "The range of int is from " << INT_MIN << " to " << INT_MAX <<
std::endl;
    std::cout << "The maximum value of unsigned int is " << UINT_MAX <<
std::endl;
    std::cout << "The range of long long is from " << LLONG_MIN << " to " <<
LLONG_MAX << std::endl;

    return 0;
}
```

这个程序会输出 `int`、`unsigned int` 和 `long long int` 类型的最大值和最小值。

注意事项

- `<climits>` 提供的是编译时确定的常量，这意味着这些值在编译时就已经固定，根据编译器和平台的不同而可能有所不同。
- 使用这些限制值可以帮助你编写更可移植和安全的代码，特别是在处理可能超出数据类型范围的操作时。

2.2.3 流程控制

在 C++ 中，流程控制语句用于根据不同条件控制程序的执行流程。它们是编程中的基本构建块，允许程序根据条件执行不同的代码段，重复执行某些操作，或者根据特定情况跳过某些代码段。下面是 C++ 中最常见的流程控制语句：

条件语句

1. `if` **语句**：基于条件的基本控制结构。如果条件为真，则执行代码块。

```
if (condition) {
    // 条件为真时执行的代码
}
```

`else` **语句**：与 `if` 语句配合使用，当 `if` 的条件为假时执行。

```
if (condition) {
    // 条件为真时执行的代码
} else {
    // 条件为假时执行的代码
}
```

`else if` **语句**：用于测试多个条件。

```
if (condition1) {
    // 第一个条件为真时执行的代码
} else if (condition2) {
    // 第二个条件为真时执行的代码
} else {
    // 所有条件为假时执行的代码
}
```

`switch` **语句**：基于变量的值选择执行不同代码块的方法。

```
switch (expression) {
    case value1:
        // expression 等于 value1 时执行的代码
        break;
    case value2:
        // expression 等于 value2 时执行的代码
        break;
    default:
        // 没有匹配的 case 时执行的代码
}
```

循环语句

for 循环：当知道循环应该执行的次数时使用。

```
for (initialization; condition; increment) {
    // 循环体
}
```

`while` **循环**：当条件为真时，重复执行代码块。

```
while (condition) {
    // 循环体
}
```

`do-while` **循环**：至少执行一次循环体，然后再检查条件。

```
do {
    // 循环体
} while (condition);
```

跳转语句

1. `break` **语句**：用于立即跳出最近的 `switch` 或循环（`for`、`while`、`do-while`）。
2. `continue` **语句**：跳过循环的当前迭代，并继续下一次迭代。
3. `goto` **语句**：直接跳转到程序中的另一个点。使用 `goto` 通常不推荐，因为它可以使代码难以阅读和维护。

流程控制语句是编程中非常重要的部分，允许开发者编写可以根据不同情况改变行为的灵活且强大的程序。在使用这些语句时，应该确保逻辑清晰，以便代码易于理解和维护。

2.2.4 函数

在 C++ 中，函数是一段执行特定任务的代码块，它可以带有参数，并且可能返回一个值。函数的使用使得代码更加模块化和可重用，有助于降低代码的复杂性，并提高可维护性。

函数的基本结构

C++ 函数的基本结构包括返回类型、函数名、参数列表和函数体：

```
返回类型 函数名(参数列表) {
    // 函数体
    // 返回语句（如果有返回值的话）
}
```

示例

以下是一个 C++ 函数的简单示例：

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

// 函数声明
int add(int x, int y);

int main() {
    int result = add(5, 3);
    cout << "Result: " << result << endl;
    return 0;
}

// 函数定义
int add(int x, int y) {
```

```
    return x + y;
}
```

在这个示例中，`add` 函数接收两个整数参数，并返回它们的和。

函数的组成部分

1. **返回类型**：指定函数返回的数据类型。如果函数不返回任何值，则使用 `void`。
2. **函数名**：函数的标识符，用于调用函数。
3. **参数列表**：括号内的变量列表，用于从函数的调用者那里接收值。如果函数不接收任何参数，则此列表为空。
4. **函数体**：大括号 `{}` 内的一系列语句，定义了函数的执行操作。

2.2.5 内联函数

内联函数（Inline Function）是C++中一种特殊的函数，其定义直接在每个调用点展开。这意味着编译器会尝试将函数调用替换为函数本身的代码，这样可以减少函数调用的开销，尤其是在小型函数中。

特点

1. **减少函数调用开销**：内联函数通常用于优化小型、频繁调用的函数，因为它避免了函数调用的常规开销（如参数传递、栈操作等）。
2. **编译器决策**：即使函数被声明为内联，编译器也可能决定不进行内联，特别是对于复杂或递归函数。
3. **适用于小型函数**：通常只有简单的、执行时间短的函数适合做内联。
4. **定义在每个使用点**：内联函数的定义（而非仅仅是声明）必须对每个使用它的文件都可见，通常意味着将内联函数定义在头文件中。

使用方法

通过在函数声明前添加关键字 `inline` 来指示编译器该函数适合内联：

```
inline int max(int x, int y) {

    return x > y ? x : y;
}
```

示例

```
#include <iostream>

inline int add(int a, int b) {
    return a + b;
}

int main() {
    int result = add(5, 3); // 编译器可能会将此替换为：int result = 5 + 3;
    std::cout << "Result: " << result << std::endl;
    return 0;
}
```

在这个示例中，函数 `add` 被定义为内联函数。当它被调用时，编译器可能会将函数调用替换为函数体内的代码。

注意事项

- **过度使用的风险**：不应滥用内联函数，因为这可能会增加最终程序的大小（代码膨胀）。对于大型函数或递归函数，内联可能导致性能下降。
- **编译器的决定**：最终是否将函数内联是由编译器决定的，即使函数被标记为 `inline`。
- **适用场景**：最适合内联的是小型函数和在性能要求高的代码中频繁调用的函数。

内联函数是一种用于优化程序性能的工具，但需要合理使用，以确保代码的可维护性和性能的平衡。

2.2.6 Lambda 表达式

Lambda 表达式是 C++11 引入的一种**匿名函数**的方式，它允许你在需要函数的地方内联地定义函数，而无需单独命名函数

Lambda 表达式的基本语法如下：

```
[capture clause](parameters) -> return_type {
    // 函数体
    // 可以使用捕获列表中的变量
    return expression; // 可选的返回语句
}
```

Lambda 表达式由以下部分组成：

- **捕获列表（Capture clause）**：用于捕获外部变量，在 Lambda 表达式中可以访问这些变量。捕获列表可以为空，也可以包含变量列表 `[var1, var2, ...]`。
- **参数列表（Parameters）**：与普通函数的参数列表类似，可以为空或包含参数列表 `(param1, param2, ...)`。
- **返回类型（Return type）**：Lambda 表达式可以自动推断返回类型auto，也可以显式指定返回类型 `-> return_type`。如果函数体只有一条返回语句，可以省略返回类型。
- **函数体（Body）**：Lambda 表达式的函数体，包含需要执行的代码。

Lambda 表达式最简单的案例是在需要一个小型函数或临时函数时直接使用它。以下是一个非常简单的例子，其中使用 Lambda 表达式来定义一个加法操作，并立即使用它来计算两个数的和。

示例：使用 Lambda 表达式进行加法

```
#include <iostream>

int main() {
    // 定义一个简单的 Lambda 表达式进行加法
    auto add = [](int a, int b) {
        return a + b;
    };

    // 使用 Lambda 表达式计算两个数的和
    int sum = add(10, 20);

    std::cout << "Sum is: " << sum << std::endl;

    return 0;
```

```
}
```

在这个例子中：

- 我们定义了一个名为 `add` 的 Lambda 表达式，它接受两个整数参数，并返回它们的和。
- 然后，我们使用这个 Lambda 表达式来计算两个数字（10 和 20）的和，并将结果存储在变量 `sum` 中。
- 最后，我们打印出这个和。

这个例子展示了 Lambda 表达式的基本用法：作为一种简洁而快速的方式来定义小型函数。

我们可以写一个例子，其中使用一个函数来找出两个数中的较大数，这个函数将接受一个 lambda 函数作为回调来比较这两个数。Lambda 函数将直接在函数调用时定义，完全是匿名的。

先回忆以下回调函数

```
#include <iostream>

// 函数，接受两个整数和一个比较的 lambda 函数
bool myCompare(int a, int b){
    return a > b;
}

int getMax(int a, int b, bool(*compare)(int, int)) {
    if (compare(a, b)) {
        return a;
    } else {
        return b;
    }
}

int main() {
    int x = 10;
    int y = 20;

    // 回调函数
    int max = getMax(x, y, myCompare);

    std::cout << "The larger number is: " << max << std::endl;

    return 0;
}
```

示例：使用匿名 Lambda 函数来返回两个数中的较大数

```
#include <iostream>

// 函数，接受两个整数和一个比较的 lambda 函数
int getMax(int a, int b, bool(*compare)(int, int)) {
    if (compare(a, b)) {
        return a;
    } else {
        return b;
    }
```

```
}

int main() {
    int x = 10;
    int y = 20;

    // 直接在函数调用中定义匿名 lambda 函数
    int max = getMax(x, y, [](int a, int b) -> bool {
        return a > b;
    });

    std::cout << "The larger number is: " << max << std::endl;

    return 0;
}
```

在这个例子中：

- `getMax` 函数接受两个整数 `a` 和 `b`，以及一个比较函数 `compare`。这个比较函数是一个指向函数的指针，它接受两个整数并返回一个布尔值。
- 在 `main` 函数中，我们调用 `getMax`，并直接在调用点定义了一个匿名的 lambda 函数。这个 lambda 函数接受两个整数并返回一个表示第一个整数是否大于第二个整数的布尔值。
- 这个 lambda 函数在 `getMax` 中被用作比较两个数的逻辑。根据 lambda 函数的返回值，`getMax` 返回较大的数。

这个例子展示了如何直接在函数调用中使用匿名 lambda 函数，使代码更加简洁和直接。这种方法在需要临时函数逻辑的场合非常有用，尤其是在比较、条件检查或小型回调中。

在 Lambda 表达式中，参数捕获是指 Lambda 表达式从其定义的上下文中捕获变量的能力。这使得 Lambda 可以使用并操作在其外部定义的变量。捕获可以按值（拷贝）或按引用进行。

让我们通过一个简单的示例来展示带参数捕获的 Lambda 表达式。

示例：使用带参数捕获的 Lambda 表达式

```
#include <iostream>

int main() {
    int x = 10;
    int y = 20;

    // 捕获 x 和 y 以便在 Lambda 内部使用
    // 这里的捕获列表 [x, y] 表示 x 和 y 被按值捕获
    auto sum = [x, y]() {
       // x++;
       // y++; 按值捕获，关注的是值本身，无法修改
        return x + y;
    };

    std::cout << "Sum is: " << sum() << std::endl;
    std::cout << "x is now: " << x << ", y is now: " << y << std::endl;

    // 捕获所有外部变量按值捕获（拷贝）
    int z = 30;
    auto multiply = [=]() {
```

```
        // x++;
        // y++; 按值捕获，关注的是值本身，无法修改
         return x * y * z;
    };
    count << x << "," << y << endl;
    std::cout << "Product is: " << multiply() << std::endl;
    std::cout << "x is now: " << x << ", y is now: " << y << std::endl;

    // 捕获所有外部变量按引用捕获
    auto modifyAndSum = [&]() {
        x = 15; // 修改 x 的实际值
        y = 25; // 修改 y 的实际值，引用捕获可以修改
        return x + y;
    };

    std::cout << "Modified Sum is: " << modifyAndSum() << std::endl;
    std::cout << "x is now: " << x << ", y is now: " << y << std::endl;

    return 0;
}
```

在这个例子中：

- 第一个 Lambda 表达式 `sum` 按值捕获了 `x` 和 `y`（即它们的副本）。这意味着 `sum` 内的 `x` 和 `y` 是在 Lambda 定义时的值的拷贝。
- 第二个 Lambda 表达式 `multiply` 使用 `[=]` 捕获列表，这表示它按值捕获所有外部变量。
- 第三个 Lambda 表达式 `modifyAndSum` 使用 `[&]` 捕获列表，这表示它按引用捕获所有外部变量。因此，它可以修改 `x` 和 `y` 的原始值。

这个示例展示了如何使用不同类型的捕获列表（按值和按引用）来控制 Lambda 表达式对外部变量的访问和修改。按值捕获是安全的，但不允许修改原始变量，而按引用捕获允许修改原始变量，但需要注意引用的有效性和生命周期问题。

以下是一个表格，概述了 Lambda 函数和内联函数在 C++ 中的相似之处和区别：

特性	Lambda 函数	内联函数
定义	一种匿名函数，通常用于定义在需要它们的地方。	一种常规函数，通过 `inline` 关键字定义。
用途	提供一种快捷方式来定义临时的、小型的函数。	用于优化小型函数，减少函数调用的开销。
语法	使用 `[capture](params) { body }` 的形式定义。	使用常规函数定义语法，但在前面加上 `inline` 关键字。
生命周期	在定义它们的作用域内有效。	在整个程序执行期间有效。
捕获外部变量	可以捕获外部作用域中的变量（按值或按引用）。	不能直接捕获外部变量，只能通过参数传递。
调用方式	作为函数对象，可直接调用。	像普通函数一样调用。

特性	Lambda 函数	内联函数
优化	可以被编译器自动内联化，但这取决于编译器优化策略。	明确请求编译器尝试内联，但实际内联化也取决于编译器。
可见性	通常只在定义它们的局部作用域内可见。	可以在定义它的任何作用域内可见。
使用场景	适合于一次性使用的场景，如作为回调、在算法中使用等。	适合于频繁调用的小型函数。

请注意，虽然 Lambda 函数和内联函数在某些方面有相似之处，如它们都可以被编译器优化以减少调用开销，但它们在设计和用途上有明显的不同。Lambda 函数的核心优势在于它们的匿名性和对外部变量的捕获能力，而内联函数则主要关注于提高小型函数的性能。

2.2.7 数组

在 C++ 中，数组是一种存储固定大小的相同类型元素的序列。数组的所有元素都存储在连续的内存位置上。这种数据结构非常适合于存储具有固定数量和相同数据类型的元素集合。

声明数组

声明数组的基本语法如下：

```
数据类型 数组名[数组大小];
```

例如，声明一个类型为 `int` 的数组，包含 10 个元素：

```
int myArray[10];
```

初始化数组

在声明数组时，您可以同时初始化数组：

```
int myArray[5] = {10, 20, 30, 40, 50};
```

如果您在初始化数组时没有指定所有元素的值，未初始化的元素将被自动设置为该数据类型的默认值（对于基本数据类型通常是 0）：

```
int myArray[5] = {10, 20}; // 其余元素将被初始化为 0
```

访问数组元素

您可以通过指定索引来访问数组中的元素。数组索引是从 0 开始的，所以数组的第一个元素是 `数组名[0]`，第二个元素是 `数组名[1]`，依此类推：

```
int value = myArray[2]; // 访问第三个元素
```

示例

以下是使用数组的简单示例：

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

int main() {
    int myArray[5] = {10, 20, 30, 40, 50};

    // 输出所有数组元素的值
    for(int i = 0; i < 5; ++i) {
        cout << "Element at index " << i << ": " << myArray[i] << endl;
    }

    return 0;
}
```

注意事项

- 数组的大小必须在编译时已知，且不能更改。
- 数组索引越界是常见的错误，可能会导致未定义的行为。
- 对于更复杂的用例，您可能需要使用 C++ 的标准模板库（STL）中的容器，如 `std::vector`，它提供了可以动态改变大小的数组。
- 数组的元素存储在连续的内存位置上，这使得访问数组元素非常快。

2.2.8 练习

- 计算器支持加减乘除

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

int add(int a,int b)
{
    return a+b;
}

int min(int a,int b)
{
    return a-b;
}

int mul(int a,int b)
{
    return a*b;
}

float divRet(int a,int b)
{
    return (float)a/b;
}

int main() {

    int a;
    int b;
```

```
    char calWay;
    while(1){
        cout << "请输入两个数："<< endl;
        cin >> a;
        cin >> b;

        cout<<"请输入运算符号：+ - * /" <<endl;
        cin >> calWay;

        switch(calWay){
        case '+':
            printf("两数之和是%d\n",add(a,b));
            break;
        case '-':
            printf("两数之差是%d\n",min(a,b));
            break;
        case '*':
            printf("两数之积是%d\n",mul(a,b));
            break;
        case '/':
            printf("两数之余是%f\n",divRet(a,b));
            break;
        default:
            printf("运算符输入错误，请重新输入\n");
        }
    }
}
```

```
#include <iostream>

using namespace std;

/*
int add(int a, int b)
{
    return a+b;
}

int minu(int a, int b)
{
    return a-b;
}

int mul(int a, int b)
{
    return a*b;
}

double diliv(int a, int b)
{
    return (double)a/b;
}
*/
```

```
int calculator(int a, int b, int (*p)(int a, int b))
{
    cout << "开始计算" << endl;
   // p(a,b);
}

int main()
{
    int a = 0;
    int b = 0;
    char cal;

    while(1){
        cout << "请输入两个数: " << endl;
        cin >> a;
        cin >> b;
        cout << "请输入运算符+, -, *, /" << endl;
        cin >> cal;
       // auto minu = [a,b]()->int{ return a - b;};
        switch(cal){
        case '+':
            cout << calculator(a,b,[](int a, int b){return a + b;}) << endl;
            break;
        case '-':
            cout << calculator(a,b,[](int a, int b){return a - b;}) << endl;
            break;
        case '*':
            cout << calculator(a,b,[](int a, int b){return a * b;}) << endl;
            break;
        case '/':
            cout << calculator(a,b,[](int a, int b){return a / b;}) << endl;
            break;
        }
    }

    //  cout << "Hello World!" << endl;
    return 0;
}
```

- 数组找最大值

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

void initArry(int *arry, int len)
{
    for(int i=0; i< len; i++){
        cout << "请输入第" << i+1 << "个数" << endl;
        cin >> arry[i];
    }
}
```

```
void printArry(int *arry,int len)
{
    for(int i=0; i< len; i++){
        cout << arry[i] << endl;
    }
}

int getMaxFromArray(int *arry, int len )
{
    int maxTmp = arry[0];
    for(int i=0; i< len; i++){
        if(maxTmp < arry[i])
            maxTmp = arry[i];
    }
    return maxTmp;
}

int main() {
    int arry[5];
    int len = sizeof(arry)/sizeof(arry[0]);

    initArry(arry,len);
    printArry(arry,len);

    cout << "最大数是：" << getMaxFromArray(arry,len);
}
```

2.2.9 指针

C++完全兼容C语言指针，多出一个this指针，在面向对象中再讲解。

```
#include <iostream>

using namespace std;

void swap(int *pa, int *pb)
{
    int tmp;
    tmp = *pa;
    *pa = *pb;
    *pb = tmp;
}

int main() {

    int a = 10;
    int b = 20;
    cout << a << endl;
    cout << b << endl;
    cout << "after chage:" << endl;
    swap(&a,&b);
    cout << a << endl;
    cout << b << endl;
```

```
}
```

2.2.9 字符串string类型

C语言中对字符串的表示通常用指针，新手会面临内存泄漏或者段错误等众多问题。

在 C++ 中，`string` 类是标准库的一部分，用于表示和操作字符串。它是对传统的 C 风格字符串（以空字符 '\0' 结尾的字符数组）的一个更安全、更方便的封装。`string` 类是在 `<string>` 头文件中定义的，并且位于 `std` 命名空间中。

`string` 类提供了许多有用的功能和特性，包括：

1. **动态大小**：与 C 风格的字符串不同，`string` 对象可以动态改变大小，这意味着你可以在运行时添加或移除字符，而不需要担心分配和释放内存。
2. **安全性**：由于 `string` 管理其自己的内存，因此减少了内存泄漏和缓冲区溢出的风险。
3. **方便的成员函数**：`string` 类提供了各种操作字符串的方法，如 `append()`（添加）、`insert()`（插入）、`erase()`（删除）、`substr()`（获取子字符串）等。
4. **操作符重载**：`string` 类重载了多个操作符，使得字符串比较、连接和赋值更加直观。例如，你可以使用 `+` 操作符来连接两个字符串，或者使用 `==` 操作符来比较两个字符串是否相等。
5. **迭代器支持**：像其他标准库容器一样，`string` 类也支持迭代器，使得你可以使用迭代器来遍历字符串中的字符。
6. **与 C 风格字符串的兼容性**：`string` 类提供了与 C 风格字符串互操作的功能，例如，你可以使用 `c_str()` 方法来获取一个与 C 风格字符串兼容的、以 null 结尾的字符数组。

下面是一个简单的 `string` 类的使用示例：

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

int main() {
    std::string str = "Hello, world!";
    std::cout << str << std::endl; // 输出字符串

    str += " I am a C++ string."; // 字符串连接
    std::cout << str << std::endl;

    std::string substr = str.substr(7, 5); // 获取子字符串
    std::cout << "Substring: " << substr << std::endl;

    return 0;
}
```

在这个示例中，我们创建了一个 `string` 对象 `str`，然后使用不同的方法对其进行操作。这展示了 `string` 类的灵活性和强大功能。

下面是一个表格，展示了 C++ 中 `std::string` 类的一些常用成员函数及其功能和参数：

函数名	功能	参数	返回值类型
`length()` 或 `size()`	返回字符串的长度	无	`size_t`
`empty()`	检查字符串是否为空	无	`bool`
`append(const string& str)`	向字符串末尾添加另一个字符串	要追加的字符串	`string&`
`substr(size_t pos = 0, size_t len = npos)`	返回一个子字符串	`pos`：子字符串的起始位置 `len`：子字符串的长度	`string`
`find(const string& str, size_t pos = 0)`	查找子字符串出现的位置	`str`：要查找的字符串 `pos`：搜索起始位置	`size_t`
`compare(const string& str)`	比较两个字符串	要比较的字符串	`int`
`erase(size_t pos = 0, size_t len = npos)`	删除字符串中的一部分	`pos`：起始位置 `len`：要删除的长度	`string&`
`insert(size_t pos, const string& str)`	在指定位置插入字符串	`pos`：插入位置 `str`：要插入的字符串	`string&`
`replace(size_t pos, size_t len, const string& str)`	替换字符串中的一部分	`pos`：起始位置 `len`：要替换的长度 `str`：替换的字符串	`string&`
`c_str()`	返回 C 风格字符串表示	无	`const char*`
`operator[] (size_t pos)`	访问指定位置的字符	`pos`：字符位置	`char&`

这些函数是 `std::string` 类中常用的一部分，提供了强大且灵活的字符串操作能力。使用这些函数可以方便地处理和修改字符串数据。

2.3 类

2.3.1 类的初探

C++ 中的类（class）是一种编程结构，用于创建对象。这些对象可以拥有属性（即数据成员）和行为（即成员函数或方法）。类的概念是面向对象编程的核心之一，其主要目的是将**数据和与数据相关的操作**封装在一起。例如，如果你有一个“汽车”类，它可能包含颜色、品牌、型号等属性（数据成员），以及启动、停止、加速等行为（成员函数）。每当你基于这个类创建一个对象时，你就有了一个具体的汽车，具有这些属性和行为。

C++ 类的基本结构通常包含：

1. **数据成员（Attributes）**：定义类的属性。这些是类内部的变量，用于存储对象的状态。
2. **成员函数（Methods）**：定义类的行为。这些是可以操作对象的数据成员的函数。
3. **构造函数和析构函数**：特殊的成员函数。构造函数在创建对象时自动调用，用于初始化对象。析构函数在对象销毁时调用，用于执行清理操作。
4. **访问修饰符**：如 `public`，`private`，`protected`，用于控制对类成员的访问权限。例如，`public` 成员可以在类的外部访问，而 `private` 成员只能在类内部访问。
5. **继承**：允许一个类继承另一个类的特性。这是代码重用和多态性的关键。

通过这些特性，C++ 类提供了一种强大的方式来组织和处理数据，使得代码更加模块化、易于理解和维护。

2.3.2 结构体引入类

2.3.2.1 回忆结构体

如果用C语言实现上面描述的汽车类，我们实现如下代码

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

struct Car{        //汽车“类”
    char *color;   //颜色
    char *brand;   //品牌
    char *type;    //车型
    int year;      //年限

    void (*printCarInfo)(char *color,char *brand,char *type, int year); //函数指针，指向车介绍函数
    void (*carRun)(char *type);        //函数指针，指向车运行的函数
    void (*carStop)(char *type);       //函数指针，执行车停止的函数
};

void bwmThreePrintCarInfo(char *color,char *brand,char *type, int year)
{
    printf("车的品牌是：%s，型号是：%s，颜色是：%s,上市年限是%d\n",
           brand,type,color,year);
}
void A6PrintCarInfo(char *color,char *brand,char *type, int year)
{
    printf("车的品牌是：%s,型号是：%s，颜色是：%s，上市年限是%d\n",
           brand,type,color,year);
```

```
}

int main()
{
    struct Car BWMthree;
    BWMthree.color = "白色";
    BWMthree.brand = "宝马";
    BWMthree.type  = "3系";
    BWMthree.year  = 2023;

    BWMthree.printCarInfo = bwmThreePrintCarInfo;

 BWMthree.printCarInfo(BWMthree.color,BWMthree.brand,BWMthree.type,BWMthree.year)
;

    struct Car *AodiA6;
    AodiA6 = (struct Car*)malloc(sizeof(struct Car));

    AodiA6->color = "黑色";
    AodiA6->brand = "奥迪";
    AodiA6->type = "A6";
    AodiA6->year = 2008;

    AodiA6->printCarInfo = A6PrintCarInfo;
    AodiA6->printCarInfo(AodiA6->color,AodiA6->brand,AodiA6->type,AodiA6->year);
    return 0;
}
```

2.3.2.2 新建C++工程来使用结构体

- 在C++中，字符串用string来表示，发现有个string赋值给char *的警告，所以修改所有char *为string类型

```
main.cpp:33:17: warning: ISO C++11 does not allow conversion from string literal
to 'char *'
```

- 修改后，发现printf的%s控制位，不能用于string的输出，所有有string构建了即将要输出的字符串
- C++中，通过std::tostring()函数，将整型数转化成字符串
- 在printInfo中使用cout输出汽车信息
- 发现在C++工程中，使用malloc在堆申请结构体空间有问题，所以直接在此引入类的概念，把struct改成class
- 引入新问题，class的成员数据和成员函数在不指定权限的情况下，默认private权限，类的对象无法进行直接访问

```
main.cpp:33:9: error: 'color' is a private member of 'Car'
main.cpp:5:11: note: implicitly declared private here
```

- 添加public属性

访问权限	类内部	同一个类的对象	派生类（子类）	类外部
public	✔ 可访问	✔ 可访问	✔ 可访问	✔ 可访问
private	✔ 可访问	✘ 不可访问	✘ 不可访问	✘ 不可访问
protected	✔ 可访问	✘ 不可访问	✔ 可访问	✘ 不可访问

- 把main函数中的原本结构体变量改成了类的实例化，如果变量类型是指针，把原来的malloc改成new一个对象
- 最后解决了所有问题

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

using namespace std;

class Car{       //汽车“类”

public:
    string color;  //颜色
    string brand;  //品牌
    string type;   //车型
    int year;     //年限

    void (*printCarInfo)(string color,string brand,string type, int year); //函数指针，指向车介绍函数
    void (*carRun)(string type);        //函数指针，指向车运行的函数
    void (*carStop)(string type);       //函数指针，执行车停止的函数
};

void bwmThreePrintCarInfo(string color,string brand,string type, int year)
{
    string str = "车的品牌是：" + brand
            + ",型号是：" + type
            + ",颜色是：" + color
            + ",上市年限是:" + std::to_string(year);
    cout << str << endl;
}
void A6PrintCarInfo(string color,string brand,string type, int year)
{
    string str = "车的品牌是：" + brand
            + ",型号是：" + type
            + ",颜色是：" + color
            + ",上市年限是:" + std::to_string(year);
    cout << str << endl;
}

int main()
{
    Car BWMthree;
    BWMthree.color = "白色";
    BWMthree.brand = "宝马";
```

```
    BWMthree.type  = "3系";
    BWMthree.year  = 2023;

    BWMthree.printCarInfo = bwmThreePrintCarInfo;

 BWMthree.printCarInfo(BWMthree.color,BWMthree.brand,BWMthree.type,BWMthree.year)
;

    Car *AodiA6 = new Car();
  //  AodiA6 = (struct Car*)malloc(sizeof(struct Car));


    AodiA6->color = "黑色";
    AodiA6->brand = "奥迪";
    AodiA6->type = "A6";
    AodiA6->year = 2008;
    AodiA6->printCarInfo = A6PrintCarInfo;

    AodiA6->printCarInfo(AodiA6->color,AodiA6->brand,AodiA6->type,AodiA6->year);

    return 0;
}
```

但是！我们还没有真正体验到面向对象的封装特性，仅仅感受到权限上的区别

2.3.2.3 真正的成员函数

- 上一节的案例中，`void (*printCarInfo)(string color,string brand,string type, int year);` 到底是变量函数函数呢?
- 是一个指针变量，是保存某个函数地址的变量，所以它不是成员函数，是成员数据
- 真正的成员函数遵守封装特性，在函数体内部访问成员数据的时候，不需要参数传递
- 在 C++ 中，双冒号 `::` 称为 "作用域解析运算符"（Scope Resolution Operator）。它用于指定一个成员（如函数或变量）属于特定的类或命名空间。例如，在类的外部定义成员函数时，`::` 用于指明该函数属于哪个类。

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

using namespace std;

class Car{       //汽车“类”

public:
    //成员数据
    string color;  //颜色
    string brand;  //品牌
    string type;   //车型
    int year;      //年限
    //其实也是成员数据，指针变量，指向函数的变量，并非真正的成员函数
```

```
    void (*printCarInfo)(string color,string brand,string type, int year); //函数
指针，指向车介绍函数
    void (*carRun)(string type);        //函数指针，指向车运行的函数
    void (*carStop)(string type);       //函数指针，执行车停止的函数

    void realPrintCarInfo();//声明成员函数

};

void Car::realPrintCarInfo() //在类的外部进行成员函数的实现
{
    string str = "车的品牌是：" + brand
            + ",型号是：" + type
            + ",颜色是：" + color
            + ",上市年限是:" + std::to_string(year);
    cout << str << endl;
}

void bwmThreePrintCarInfo(string color,string brand,string type, int year)
{
    string str = "车的品牌是：" + brand
            + ",型号是：" + type
            + ",颜色是：" + color
            + ",上市年限是:" + std::to_string(year);
    cout << str << endl;
}
void A6PrintCarInfo(string color,string brand,string type, int year)
{
    string str = "车的品牌是：" + brand
            + ",型号是：" + type
            + ",颜色是：" + color
            + ",上市年限是:" + std::to_string(year);
    cout << str << endl;
}

int main()
{
    Car BWMthree;
    BWMthree.color = "白色";
    BWMthree.brand = "宝马";
    BWMthree.type  = "3系";
    BWMthree.year  = 2023;

    BWMthree.printCarInfo = bwmThreePrintCarInfo;

 BWMthree.printCarInfo(BWMthree.color,BWMthree.brand,BWMthree.type,BWMthree.year)
;
    BWMthree.realPrintCarInfo();

    Car *AodiA6 = new Car();
  //  AodiA6 = (struct Car*)malloc(sizeof(struct Car));


    AodiA6->color = "黑色";
    AodiA6->brand = "奥迪";
    AodiA6->type = "A6";
```

```
    AodiA6->year = 2008;
    AodiA6->printCarInfo = A6PrintCarInfo;

    AodiA6->printCarInfo(AodiA6->color,AodiA6->brand,AodiA6->type,AodiA6->year);
    AodiA6->realPrintCarInfo();
    return 0;
}
```

2.3.4 QT中经常出现的用法

> 在 C++中，一个类包含另一个类的对象称为组合（Composition）。这是一种常见的设计模式，用于表示一个类是由另一个类的对象组成的。这种关系通常表示一种"拥有"（"has-a"）的关系。

普通变量访问成员变量或者成员函数，使用“ . ”运算符

指针变量访问成员变量或者成员函数，使用“ -> ”运算符，像C语言的结构体用法

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

using namespace std;

class Wheel
{
public:
    string brand;
    int year;

    void wheelPrintInfo();
};

void Wheel::wheelPrintInfo()
{
    cout << "我的轮胎品牌是：" << brand << endl;
    cout << "我的轮胎日期是：" << year  << endl;
}

//在 C++中，一个类包含另一个类的对象称为组合（Composition）。
class Car{       //汽车“类”

public:
    //成员数据
    string color;  //颜色
    string brand;  //品牌
    string type;   //车型
    int year;     //年限

    Wheel wl;
    Wheel *pwl;

    //其实也是成员数据，指针变量，指向函数的变量，并非真正的成员函数
```

```
    void (*printCarInfo)(string color,string brand,string type, int year); //函数指针，指向车介绍函数
    void (*carRun)(string type);        //函数指针，指向车运行的函数
    void (*carStop)(string type);       //函数指针，执行车停止的函数

    void realPrintCarInfo();//声明成员函数

};

void Car::realPrintCarInfo() //在类的外部进行成员函数的实现
{
    string str = "车的品牌是：" + brand
            + ",型号是：" + type
            + ",颜色是：" + color
            + ",上市年限是:" + std::to_string(year);
    cout << str << endl;
}

void bwmThreePrintCarInfo(string color,string brand,string type, int year)
{
    string str = "车的品牌是：" + brand
            + ",型号是：" + type
            + ",颜色是：" + color
            + ",上市年限是:" + std::to_string(year);
    cout << str << endl;
}
void A6PrintCarInfo(string color,string brand,string type, int year)
{
    string str = "车的品牌是：" + brand
            + ",型号是：" + type
            + ",颜色是：" + color
            + ",上市年限是:" + std::to_string(year);
    cout << str << endl;
}

int main()
{
    Car BWMthree;
    BWMthree.color = "白色";
    BWMthree.brand = "宝马";
    BWMthree.type  = "3系";
    BWMthree.year  = 2023;

    BWMthree.pwl = new Wheel();
    BWMthree.pwl->brand = "米其林";
    BWMthree.pwl->year = 2023;
    //BWMthree.wl.brand = "米其林";
    //BWMthree.wl.year = 2023;

    BWMthree.printCarInfo = bwmThreePrintCarInfo;

 BWMthree.printCarInfo(BWMthree.color,BWMthree.brand,BWMthree.type,BWMthree.year)
;
    BWMthree.realPrintCarInfo();
    //BWMthree.wl.wheelPrintInfo();
```

```
    Car *AodiA6 = new Car();
  //  AodiA6 = (struct Car*)malloc(sizeof(struct Car));


    AodiA6->color = "黑色";
    AodiA6->brand = "奥迪";
    AodiA6->type = "A6";
    AodiA6->year = 2008;
    AodiA6->printCarInfo = A6PrintCarInfo;

    AodiA6->pwl = new Wheel;
    AodiA6->pwl->brand = "普利司通";
    AodiA6->pwl->year = 2012;

    //AodiA6->wl.brand = "马牌";
    //AodiA6->wl.year = 2023;

    AodiA6->printCarInfo(AodiA6->color,AodiA6->brand,AodiA6->type,AodiA6->year);
    AodiA6->realPrintCarInfo();
    //AodiA6->wl.wheelPrintInfo();
    AodiA6->pwl->wheelPrintInfo();
    return 0;
}
```

2.4 权限初识

2.4.1 基本介绍

C++中的访问权限主要分为三种：`public`、`private`和`protected`。这些权限决定了类成员（包括数据成员和成员函数）的可访问性。以下是一个总结表格，说明了在不同情况下这些权限如何应用：

访问权限	类内部	同一个类的对象	派生类（子类）	类外部
public	✔ 可访问	✔ 可访问	✔ 可访问	✔ 可访问
private	✔ 可访问	✘ 不可访问	✘ 不可访问	✘ 不可访问
protected	✔ 可访问	✘ 不可访问	✔ 可访问	✘ 不可访问

使用权限（如`public`、`private`和`protected`）在C++中是一种关键的封装手段，它们旨在控制对类成员的访问。下面是一个表格，总结了使用权限的主要好处和潜在缺点：

好处 / 缺点	描述
好处	
封装性	通过隐藏类的内部实现（私有和受保护成员），提高了代码的安全性和健壮性。
接口与实现的分离	公开接口（公开成员）与私有实现分离，有助于用户仅关注于如何使用类而不是如何实现。
易于维护	修改类的内部实现不会影响使用该类的代码，从而降低了维护成本。

好处 / 缺点	描述
控制读写访问	通过设置访问权限，可以精确控制类成员的读写访问。
继承的灵活性	`protected` 成员在派生类中是可访问的，使得继承更加灵活。
缺点	
增加复杂性	过度使用或不当使用权限可能导致代码结构复杂，难以理解。
测试难度	私有成员的测试比公共成员更困难，因为它们不能从类的外部访问。
灵活性降低	过于严格的封装可能限制了某些有效的用法，降低了灵活性。
可能导致紧耦合	过多依赖 `friend` 类或函数可能导致类之间的耦合过紧。

2.4.2 目前能概况的结论

- `public` 权限相当于我们学习C语言结构体一样，不考虑访问权限的存在，但是要注意，**类中不写权限，默认是私有权限**
- `protected` 留到继承讲解的时候再提
- `private` 私有权限，通过一下案例向各位表达一下作用的意思，但需要未来实战中慢慢体会。

这个例子将阐述在类设计中使用`private`成员的必要性。我们将创建一个简单的`BankAccount`类，展示如何使用`private`来保护账户的余额，确保它只能通过指定的方法进行修改。

所以，我们可以脑部一个场景：

> 银行的账户是一个模板，是一个类，有存款人信息和账户额度，而具体的存款人视为一个对象，
>
> 一个对象不能私自修改账户额度，需要通过一个操作流程，比如去ATM或者柜台进行操作才能修改到账户额度，
>
> 所以，存款人信息和账户额度设计成私有权限，通过公有的操作流程，也就是公有函数去操作私有变量。
>
> 基于这个场景，我们编程实现代码

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

using namespace std;

/*
银行的账户是一个模板，是一个类，有存款人信息和账户额度，而具体的存款人视为一个对象，
一个对象不能私自修改账户额度，需要通过一个操作流程，比如去ATM或者柜台进行操作才能修改到账户额度，
所以，存款人信息和账户额度设计成私有权限，通过公有的操作流程，也就是公有函数去操作私有变量。
基于这个场景，我们编程实现代码
*/

class BankAccount{
private:
    //有存款人信息和账户额度
    string name;
    string addr;
```

```
    int age;
    double balance;
public:
    string bankAddr;
    //比如去ATM或者柜台进行操作才能修改到账户额度
    void registerMes(string newName, string newAddr,int newAge,double
newBalance);
    void withdraw(double amount);
    void deposit(double amount);
    double getBalance();
    void printUserInfo();
};
void BankAccount::printUserInfo()
{
    string mesTem = "账户名：" + name + "，地址：" + addr +
            ",年龄："+ std::to_string(age) + ",存款：" + std::to_string(balance);
    cout << mesTem << endl;
}

void BankAccount::registerMes(string newName, string newAddr,int newAge,double
newBalance)
{
    name = newName;
    addr = newAddr;
    age  = newAge;
    balance = newBalance;
}

// 存款方法
void BankAccount::deposit(double amount) {
    if (amount > 0) {
        balance += amount;
    } else {
        cerr << "Deposit amount must be positive." << endl;
    }
}
// 取款方法
void BankAccount::withdraw(double amount) {
    if (amount > balance) {
        cerr << "Insufficient funds." << endl;
    } else if (amount <= 0) {
        cerr << "Withdrawal amount must be positive." << endl;
    } else {
        balance -= amount;
    }
}

// 获取当前余额的方法
double BankAccount::getBalance() {
    return balance;
}

int main()
{
    BankAccount user1;
    user1.registerMes("老陈","深圳光明区",35,100);
```

```
    user1.printUserInfo();
    user1.deposit(1000);
    cout << user1.getBalance() << endl;
    user1.withdraw(30);
    cout << user1.getBalance() << endl;
    return 0;
}
```

在这个示例中，`balance` 是一个`private`成员变量，它不能被类的外部直接访问。这保证了账户余额只能通过类提供的方法（如 `deposit`，`withdraw`，和 `getBalance`）来修改和查询，从而防止了不合适的修改，比如直接设置余额为负数或任意值。这样的设计保证了类的封装性和数据的完整性。

2.4.3 提问和回答

问：为什么新手学习C++感受不到访问权限的必要性呢

答：新手学习C++时可能不会立即感受到访问权限（如`public`、`private`、`protected`）的必要性，主要有以下几个原因：

1. **简单的例子和练习**：初学者通常从简单的例子和练习开始，这些例子可能不需要复杂的封装或继承结构。在这种情况下，访问权限的作用可能不太明显。
2. **封装的概念需要时间去理解**：封装是面向对象编程中的一个核心概念，但对于初学者来说，理解封装的价值需要一定的时间和实践。在初期，更多的关注点可能放在基本语法和程序结构上。
3. **缺乏大型项目经验**：在小型项目或单文件程序中，访问权限的重要性可能不如在大型、多人协作的项目中那么显著。在复杂的软件开发中，适当的访问控制对于代码的维护性和可读性至关重要。
4. **直接操作感觉更简单**：对于初学者来说，直接访问和修改类的所有成员可能看起来更简单直接。他们可能还没有遇到由于不恰当访问控制导致的维护和调试问题。
5. **抽象和设计模式的理解**：理解何时以及如何使用访问权限通常涉及到对软件设计模式和抽象的深入理解。这些通常是随着经验积累和更深入的学习而逐渐掌握的。

随着经验的增长，学习者开始处理更复杂的项目，他们将开始意识到恰当的访问控制的重要性，特别是在保持代码的可维护性、可读性以及在团队环境中的协作方面。因此，对于教育者和学习者来说，强调并实践这些概念是很重要的，以便在编程技能成熟时能够有效地运用它们。

2.5 引用

引用变量是一个别名，也就是说，它是某个已存在变量的另一个名字。

一旦把引用初始化为某个变量，就可以使用该引用名称或变量名称来指向变量。

思维发散:

在C语言中，一个数据对应一个内存，通过由一个变量名来访问这个内存空间的数据，叫做直接访问，相对直接访问，有个间接访问的说法，叫做指针。

而引用相当于又给这个内存中的数据提供了一个新的变量名，

这个变量名功能比传统变量名更特殊，是直达地址的，后续代码验证!

2.6.1 和指针的区别

引用很容易与指针混淆，它们之间有三个主要的不同：

- 不存在空引用。引用必须连接到一块合法的内存。
- 一旦引用被初始化为一个对象，就不能被指向到另一个对象。指针可以在任何时候指向到另一个对象。
- 引用必须在创建时被初始化。指针可以在任何时间被初始化。
- 官方没有明确说明，但是引用确实不是传统意义上的独立变量，它不能“变”嘛

  试想变量名称是变量附属在内存位置中的标签，您可以把引用当成是变量附属在内存位置中的第二个标签。因此，您可以通过原始变量名称或引用来访问变量的内容。例如：

  ```
  int i = 17; int* p = &i;  *p = 20;
  ```

  我们可以为 i 声明引用变量，如下所示：

  ```
  int&  r = i;
  double& s = d;
  ```

  在这些声明中，& 读作**引用**。因此，第一个声明可以读作 "r 是一个初始化为 i 的整型引用"，第二个声明可以读作 "s 是一个初始化为 d 的 double 型引用"。下面的实例使用了 int 和 double 引用：

```
#include <iostream>

using namespace std;

int main ()
{
   // 声明简单的变量
   int    i;
   double d;

   // 声明引用变量
   int&    r = i;
   double& s = d;

   i = 5;
   cout << "Value of i : " << i << endl;
   cout << "Value of i reference : " << r  << endl;

   d = 11.7;
   cout << "Value of d : " << d << endl;
   cout << "Value of d reference : " << s  << endl;

   return 0;
}
```

2.6.2 把引用作为参数

我们已经讨论了如何使用指针来实现引用调用函数。下面的实例使用了引用来实现引用调用函数。

实例

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

// 函数声明
void swap(int& x, int& y);

int main ()
{
   // 局部变量声明
   int a = 100;
   int b = 200;

   cout << "交换前，a 的值：" << a << endl;
   cout << "交换前，b 的值：" << b << endl;

   /* 调用函数来交换值 */
   swap(a, b);

   cout << "交换后，a 的值：" << a << endl;
   cout << "交换后，b 的值：" << b << endl;

   return 0;
}

// 函数定义
void swap(int& x, int& y)
{
   int temp;
   temp = x; /* 保存地址 x 的值 */
   x = y;    /* 把 y 赋值给 x */
   y = temp; /* 把 x 赋值给 y  */

   return;
}
```

运行结果：

```
交换前，a 的值： 100
交换前，b 的值： 200
交换后，a 的值： 200
交换后，b 的值： 100
```

2.6.3 把引用作为返回值

通过使用引用来替代指针，会使 C++ 程序更容易阅读和维护。C++ 函数可以返回一个引用，方式与返回一个指针类似。

当函数返回一个引用时，则返回一个指向返回值的隐式指针。这样，**函数就可以放在赋值语句的左边。**例如，请看下面这个简单的程序：

```
#include <iostream>

using namespace std;

double vals[] = {10.1, 12.6, 33.1, 24.1, 50.0};

double& setValues(int i) {
   double& ref = vals[i];
   return ref;   // 返回第 i 个元素的引用，ref 是一个引用变量，ref 引用 vals[i]


}

// 要调用上面定义函数的主函数
int main ()
{

   cout << "改变前的值" << endl;
   for ( int i = 0; i < 5; i++ )
   {
       cout << "vals[" << i << "] = ";
       cout << vals[i] << endl;
   }

   setValues(1) = 20.23; // 改变第 2 个元素
   setValues(3) = 70.8;  // 改变第 4 个元素

   cout << "改变后的值" << endl;
   for ( int i = 0; i < 5; i++ )
   {
       cout << "vals[" << i << "] = ";
       cout << vals[i] << endl;
   }
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
改变前的值
vals[0] = 10.1
vals[1] = 12.6
vals[2] = 33.1
vals[3] = 24.1
vals[4] = 50
改变后的值
vals[0] = 10.1
vals[1] = 20.23
vals[2] = 33.1
vals[3] = 70.8
vals[4] = 50
```

当返回一个引用时，要注意被引用的对象不能超出作用域。所以返回一个对局部变量的引用是不合法的，但是，可以返回一个对静态变量的引用。

```
int& func() {
   int q;
   //! return q; // 在编译时发生错误
   static int x;
   return x;     // 安全，x 在函数作用域外依然是有效的
}
```

2.6 重载

2.5.1 函数重载

在同一个作用域内，可以声明几个功能类似的同名函数，

这些同名函数的形式参数（指参数的个数、类型或者顺序）必须不同。您不能仅通过返回类型的不同来重载函数。

下面的实例中，同名函数 **print()** 被用于输出不同的数据类型：

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

class printData
{
   public:
      void print(int i) {
        cout << "整数为: " << i << endl;
      }

      void print(double  f) {
        cout << "浮点数为: " << f << endl;
      }

      void print(char c[]) {
        cout << "字符串为: " << c << endl;
      }
};

int main(void)
{
   printData pd;

   // 输出整数
   pd.print(5);
   // 输出浮点数
   pd.print(500.263);
   // 输出字符串
   char c[] = "Hello C++";
   pd.print(c);

   return 0;
}
```

2.5.2 运算符重载

在C++中，运算符重载是一个允许程序员自定义各种运算符（如 `+`, `-`, `==`, `!=` 等）在自定义类型（类或结构体）上的行为的特性。这意味着你可以定义类似于内置类型的运算符行为，使你的自定义类型更加直观和易于使用。

基本原则

1. **不可以创建新的运算符**：只能重载已经存在的运算符。
2. **至少有一个操作数是用户定义的类型**：不能重载两个基本类型的运算符。
3. **不能更改运算符的优先级**：重载的运算符保持其原有的优先级和结合性。

示例1：假设我们有一个Person 类，我们可以重载 `==` 运算符来实现两个Person是否相等的判断。

```
#include <iostream>

using namespace std;


class Person
{
public:
    string name;
    int inNumberTail;

    bool operator==(Person pTmp);

};
bool Person::operator==(Person pTmp){

    return pTmp.name == name && pTmp.inNumberTail == inNumberTail;
}

int main()
{
    //假设我们认定名字和身份证尾号6位一样的两个对象是同一个人！
    Person p1;
    p1.name = "张三";
    p1.inNumberTail = 412508;

    Person p2;
    p2.name = "张三";
    p2.inNumberTail = 412508;

    bool ret = p1 == p2;
    cout << ret << endl;

    return 0;
}
```

示例2：假设我们有一个简单的 `Point` 类，我们可以重载 `+` 运算符来实现两个点的加法。

```
class Point {
```

```
public:
    int x, y;

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

int main() {
    Point p1;
    p1.x = 1;
    p1.y = 2;

    Point p2;
    p2.x = 2;
    p2.y = 3;

    Point p3 = p1 + p2;  // 使用重载的 + 运算符
    std::cout << "p3.x: " << p3.x << ", p3.y: " << p3.y << std::endl; // 输出
p3.x: 4, p3.y: 6
    return 0;
}
```

在这个例子中，`operator+` 被重载为一个成员函数，接受一个 `Point` 类型的常量引用作为参数，并返回两个点相加的结果。

这里的 `const` 表明这个 `operator+` 函数不会修改调用它的 `Point` 对象。它只是读取对象的 `x` 和 `y` 成员，并返回一个新的 `Point` 对象。这种做法在设计类的时候是很有用的，因为它可以确保某些函数不会意外地改变对象的状态，同时也使得这个函数可以在常量对象上被调用。

注意事项

- **一致性**：重载的运算符应与其原始意图和常见用法保持一致。例如，`+` 运算符通常应该实现加法，而不是其他意外的操作。
- **复杂性**：过度使用运算符重载可能导致代码难以理解和维护。确保它们的使用直观且合理。

运算符重载是C++中提高代码可读性和表达力的强大工具，但需要谨慎使用，以保证代码的清晰性和维护性。

2.7 构造函数

2.7.1 什么是构造函数

类的**构造函数**是类的一种特殊的成员函数，它会在每次创建类的新对象时执行。

构造，那构造的是什么呢?

构造成员变量的初始化值，内存空间等

构造函数的名称与类的名称是完全相同的，并且不会返回任何类型，也不会返回 void。构造函数可用于为某些成员变量设置初始值。

下面的实例有助于更好地理解构造函数的概念：

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

using namespace std;  // 使用std命名空间

class Car {
public:
    string brand;  // 不需要使用std::string
    int year;

    // 无参构造函数
    Car() {
        brand = "未知";
        year = 0;
        cout << "无参构造函数被调用" << endl;  // 不需要使用std::cout和std::endl
    }

    void display() {
        cout << "Brand: " << brand << ", Year: " << year << endl;
    }
};

int main() {
    Car myCar; // 创建Car对象
    myCar.display(); // 显示车辆信息

    return 0;
}
```

2.7.2 带参数构造函数

默认的构造函数没有任何参数，但如果需要，构造函数也可以带有参数。这样在创建对象时就会给对象赋初始值，如下面的例子所示：

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

using namespace std;

class Car {
public:
    string brand;
    int year;

    // 带参数的构造函数，使用常规的赋值方式
    Car(string b, int y) {
        brand = b;
        year = y;
    }

    void display() {
        cout << "Brand: " << brand << ", Year: " << year << endl;
    }
```

```
};

int main() {
    Car myCar("Toyota", 2020); // 使用带参数的构造函数创建Car对象
    myCar.display(); // 显示车辆信息

    return 0;
}
```

2.7.3 拷贝构造函数

2.7.3.1 基本概念及发生条件

拷贝构造函数是 C++ 中的一种特殊的构造函数，用于创建一个新对象作为现有对象的副本。它在以下几种情况下被调用：

1. **当一个新对象被创建为另一个同类型的现有对象的副本时**：
   - 例如：`MyClass obj1 = obj2;` 或 `MyClass obj1(obj2);`，其中 `obj2` 是现有的对象。
2. **将对象作为参数传递给函数时**（按值传递）：
   - 当对象作为参数传递给函数，并且参数不是引用时，会使用拷贝构造函数创建函数内部的对象副本。
3. **从函数返回对象时**（按值返回）：
   - 当函数返回对象，并且没有使用引用或指针时，拷贝构造函数用于从函数返回值创建副本。
4. **初始化数组或容器中的元素时**：
   - 例如，在创建一个包含对象的数组时，数组中的每个对象都是通过拷贝构造函数初始化的。

拷贝构造函数的典型声明如下：

```
class MyClass {
public:
    MyClass(const MyClass& other);
};
```

其中，`other` 是对同类型对象的引用，通常是常量引用。

示例代码

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

using namespace std;

class Car {
public:
    string brand;
    int year;

    // 常规构造函数
    Car(string b, int y) : brand(b), year(y) {}

    // 拷贝构造函数
```

```
    Car(const Car& other) {
        brand = other.brand;
        year = other.year;
        cout << "拷贝构造函数被调用" << endl;
    }

    void display() {
        cout << "Brand: " << brand << ", Year: " << year << endl;
    }
};

int main() {
    Car car1("Toyota", 2020);  // 使用常规构造函数
    Car car2 = car1;           // 使用拷贝构造函数

    car1.display();
    car2.display();

    return 0;
}
```

2.7.3.2 浅拷贝

在 C++ 中，深拷贝和浅拷贝是处理对象拷贝时的两种不同方法，尤其是在对象包含指针或动态分配的内存时。我将分别给出深拷贝和浅拷贝的例子。

浅拷贝 (Shallow Copy)

浅拷贝只复制对象的成员变量的值。如果成员变量是指针，则复制指针的值（即内存地址），而不是指针所指向的实际数据。这会导致多个对象共享相同的内存地址。

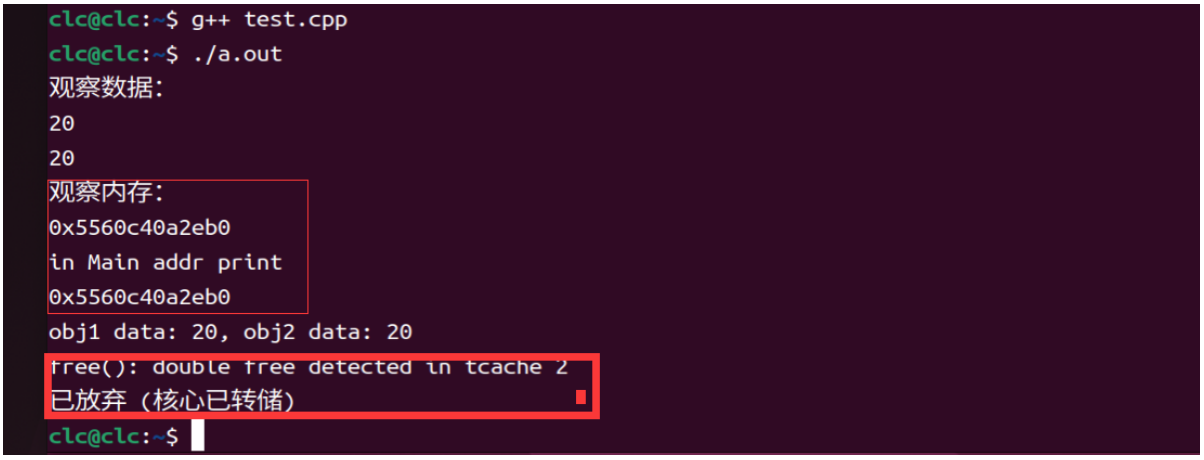
```
#include <iostream>
using namespace std;

class Shallow {
public:
    int* data;


    Shallow(int d) {
         //(d)：这是初始化表达式。在这里，分配的 int 类型内存被初始化为 d 的值。如果 d 的值是20，那么分配的内存将被初始化为 20。
        data = new int(d); // 动态分配内存
        cout << "观察数据：" << endl;
        cout << d << endl;
        cout << *data << endl;

        cout << "观察内存在构造函数中：" << endl;
        cout << data << endl;
    }

    // 默认的拷贝构造函数是浅拷贝
    ~Shallow() {
        delete data; // 释放内存
    }
```

```
};

int main() {
    Shallow obj1(20);

    Shallow obj2 = obj1; // 浅拷贝

    cout << "观察内存在main函数obj2的data地址：" << endl;
    cout << obj2.data << endl;

    cout << "obj1 data: " << *obj1.data << ", obj2 data: " << *obj2.data << endl;

    return 0;
}
```

在这个例子中，`obj2` 是通过浅拷贝 `obj1` 创建的。这意味着 `obj1.data` 和 `obj2.data` 指向相同的内存地址。

当 `obj1` 和 `obj2` 被销毁时，同一内存地址会被尝试释放两次，导致潜在的运行时错误。

在QT中我们不能直观看见，在Linux中我们获得如下运行结果：



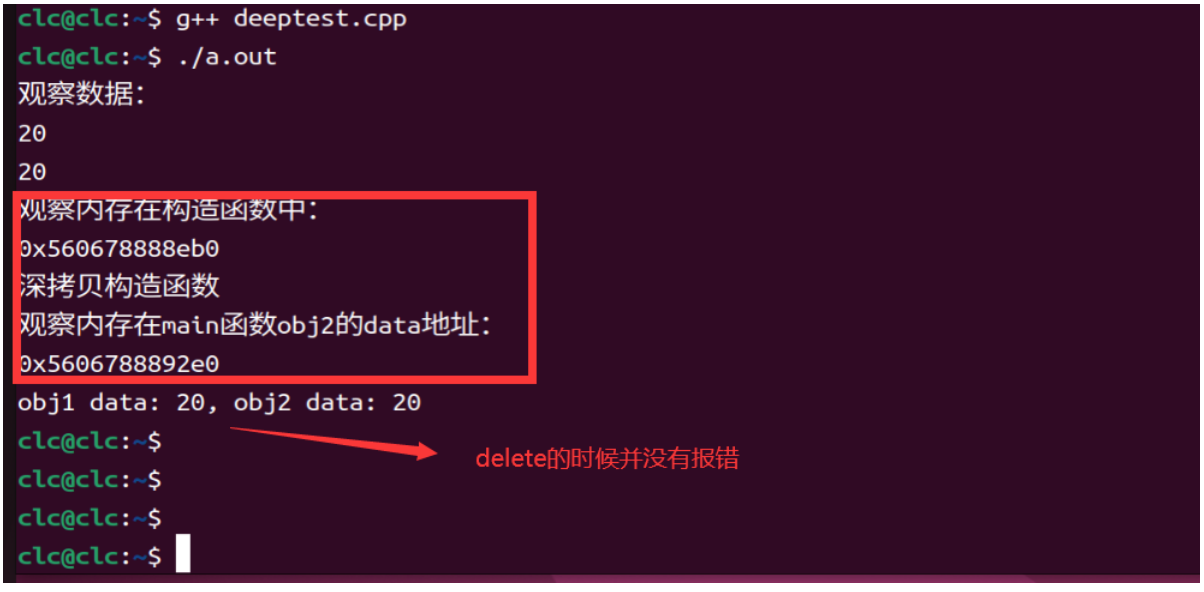
2.7.3.3 深拷贝

深拷贝复制对象的成员变量的值以及指针所指向的实际数据。这意味着创建新的独立副本，避免了共享内存地址的问题。

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

class Deep {
public:
    int* data;

    Deep(int d) {
        data = new int(d); // 动态分配内存

        cout << "观察数据：" << endl;
        cout << d << endl;
        cout << *data << endl;

        cout << "观察内存在构造函数中：" << endl;
```

```
        cout << data << endl;
    }

    // 显式定义深拷贝的拷贝构造函数
    Deep(const Deep& source) {
        data = new int(*source.data); // 复制数据，而不是地址
        cout << "深拷贝构造函数\n";
    }

    ~Deep() {
        delete data; // 释放内存
    }
};

int main() {
    Deep obj1(20);

    Deep obj2 = obj1; // 深拷贝
    cout << "观察内存在main函数obj2的data地址：" << endl;
    cout << obj2.data << endl;


    cout << "obj1 data: " << *obj1.data << ", obj2 data: " << *obj2.data << endl;

    return 0;
}
```

在这个例子中，`obj2` 是通过深拷贝 `obj1` 创建的。这意味着 `obj1.data` 和 `obj2.data` 指向不同的内存地址。每个对象有自己的内存副本，因此不会相互影响，避免了潜在的运行时错误。



2.7.3.4 规则三则

在 C++ 中，规则三则（Rule of Three）是一个面向对象编程原则，它涉及到类的拷贝控制。规则三则指出，如果你需要显式地定义或重载类的任何一个拷贝控制操作（拷贝构造函数、拷贝赋值运算符、析构函数），那么你几乎肯定需要显式地定义或重载所有三个。这是因为这三个功能通常都是用于管理动态分配的资源，比如在堆上分配的内存。

下面是一个遵循规则三则的简单示例：

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

class MyClass {
private:
    char* buffer;

public:
    // 构造函数
    MyClass(const char* str) {
        if (str) {
            buffer = new char[strlen(str) + 1];
            strcpy(buffer, str);
        } else {
            buffer = nullptr;
        }
    }

    // 析构函数
    ~MyClass() {
        delete[] buffer;
    }

    // 拷贝构造函数
    MyClass(const MyClass& other) {
        if (other.buffer) {
            buffer = new char[strlen(other.buffer) + 1];
            strcpy(buffer, other.buffer);
        } else {
            buffer = nullptr;
        }
    }

    // 拷贝赋值运算符
    MyClass& operator=(const MyClass& other) {
        if (this != &other) {
            delete[] buffer; // 首先删除当前对象的资源
            if (other.buffer) {
                buffer = new char[strlen(other.buffer) + 1];
                strcpy(buffer, other.buffer);
            } else {
                buffer = nullptr;
            }
        }
        return *this;
    }
};

int main() {
    MyClass obj1("Hello");
    MyClass obj2 = obj1; // 调用拷贝构造函数
    MyClass obj3("World");
    obj3 = obj1; // 调用拷贝赋值运算符

    return 0;
}
```

在这个例子中：

- 构造函数为成员变量 `buffer` 分配内存，并复制给定的字符串。
- 析构函数释放 `buffer` 所占用的内存，以避免内存泄露。
- 拷贝构造函数创建一个新对象作为另一个现有对象的副本，并为其分配新的内存，以避免多个对象共享同一内存。
- 拷贝赋值运算符更新对象时，首先释放原有资源，然后根据新对象的状态分配新资源。

这个类遵循规则三则，确保了动态分配资源的正确管理，避免了内存泄露和浅拷贝问题。

2.7.3.5 避免不必要的拷贝

避免不必要的拷贝是 C++ 程序设计中的一个重要原则，尤其是在处理大型对象或资源密集型对象时。使用引用（包括常量引用）和移动语义（C++11 引入）是实现这一目标的两种常见方法。下面是两个示例：

1. 使用引用传递对象

通过使用引用（尤其是常量引用）来传递对象，可以避免在函数调用时创建对象的副本。

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

class LargeObject {
    // 假设这是一个占用大量内存的大型对象
};

void processLargeObject(const LargeObject& obj) {
    // 处理对象，但不修改它
    cout << "Processing object..." << endl;
}

int main() {
    LargeObject myLargeObject;
    processLargeObject(myLargeObject); // 通过引用传递，避免拷贝
    return 0;
}
```

在这个例子中，`processLargeObject` 函数接受一个对 `LargeObject` 类型的常量引用，避免了在函数调用时复制整个 `LargeObject`。

2. 使用移动语义

C++11 引入了移动语义，允许资源（如动态分配的内存）的所有权从一个对象转移到另一个对象，这避免了不必要的拷贝。

```
#include <iostream>
#include <utility> // 对于 std::move
using namespace std;

class MovableObject {
public:
    MovableObject() {
```

```
        // 构造函数
    }

    MovableObject(const MovableObject& other) {
        // 拷贝构造函数（可能很昂贵）
    }

    MovableObject(MovableObject&& other) noexcept {
        // 移动构造函数（轻量级）
        // 转移资源的所有权
    }

    MovableObject& operator=(MovableObject&& other) noexcept {
        // 移动赋值运算符
        // 转移资源的所有权
        return *this;
    }
};

MovableObject createObject() {
    MovableObject obj;
    return obj; // 返回时使用移动语义，而非拷贝
}

int main() {
    MovableObject obj = createObject(); // 使用移动构造函数
    return 0;
}
```

在这个例子中，`MovableObject` 类有一个移动构造函数和一个移动赋值运算符，它们允许对象的资源（如动态分配的内存）在赋值或返回时被"移动"而非复制。这减少了对资源的不必要拷贝，提高了效率。

通过这些方法，你可以在 C++ 程序中有效地减少不必要的对象拷贝，尤其是对于大型或资源密集型的对象。

2.7.3.6 拷贝构造函数的隐式调用

在 C++ 中，拷贝构造函数可能会在几种不明显的情况下被隐式调用。这种隐式调用通常发生在对象需要被复制时，但代码中并没有明显的赋值或构造函数调用。了解这些情况对于高效和正确地管理资源非常重要。下面是一些典型的隐式拷贝构造函数调用的例子：

1. **作为函数参数传递（按值传递）**

当对象作为函数参数按值传递时，会调用拷贝构造函数来创建参数的本地副本。

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

class MyClass {
public:
    MyClass() {}

    MyClass(const MyClass &) {
        cout << "拷贝构造函数被隐式调用" << endl;
    }
};
```

```
void function(MyClass obj) {
    // 对 obj 的操作
}

int main() {
    MyClass myObject;
    function(myObject); // 调用 function 时，拷贝构造函数被隐式调用
    return 0;
}
```

2. 从函数返回对象（按值返回）

当函数返回一个对象时，拷贝构造函数会被用于创建返回值的副本。

```
MyClass function() {
    MyClass tempObject;
    return tempObject; // 返回时，拷贝构造函数被隐式调用
}

int main() {
    MyClass myObject = function(); // 接收返回值时可能还会有一次拷贝（或移动）
    return 0;
}
```

3. 初始化另一个对象

当用一个对象初始化另一个同类型的新对象时，会使用拷贝构造函数。

```
int main() {
    MyClass obj1;
    MyClass obj2 = obj1; // 初始化时，拷贝构造函数被隐式调用
    return 0;
}
```

在所有这些情况下，如果类包含资源管理（例如，动态内存分配），那么正确地实现拷贝构造函数是非常重要的，以确保资源的正确复制和管理，防止潜在的内存泄漏或其他问题。此外，随着 C++11 的引入，移动语义提供了对资源的高效管理方式，可以减少这些场景中的资源复制。

2.7.3.7 禁用拷贝构造函数

在 C++ 中，禁用拷贝构造函数是一种常用的做法，尤其是在设计那些不应该被复制的类时。这可以通过将拷贝构造函数声明为 `private` 或使用 C++11 引入的 `delete` 关键字来实现。这样做的目的是防止类的对象被拷贝，从而避免可能导致的问题，如资源重复释放、无意义的资源复制等。

使用 `delete` 关键字

在 C++11 及更高版本中，可以使用 `delete` 关键字明确指定不允许拷贝构造：

```
class NonCopyable {
public:
    NonCopyable() = default;  // 使用默认构造函数

    // 禁用拷贝构造函数
    NonCopyable(const NonCopyable&) = delete;
```

```
    // 禁用拷贝赋值运算符
    NonCopyable& operator=(const NonCopyable&) = delete;
};

int main() {
    NonCopyable obj1;
    // NonCopyable obj2 = obj1;  // 编译错误，拷贝构造函数被禁用
    return 0;
}
```

这种方法清晰明了，它向编译器和其他程序员直接表明该类的对象不能被拷贝。

使用 `private` 声明（C++98/03）

在 C++11 之前，常见的做法是将拷贝构造函数和拷贝赋值运算符声明为 `private`，并且不提供实现：

```
class NonCopyable {
private:
    // 将拷贝构造函数和拷贝赋值运算符设为私有
    NonCopyable(const NonCopyable&);
    NonCopyable& operator=(const NonCopyable&);

public:
    NonCopyable() {}
};

int main() {
    NonCopyable obj1;
    // NonCopyable obj2 = obj1;  // 编译错误，因为无法访问私有的拷贝构造函数
    return 0;
}
```

在这个例子中，任何尝试拷贝 `NonCopyable` 类型对象的操作都会导致编译错误，因为拷贝构造函数和拷贝赋值运算符是私有的，外部代码无法访问它们。

通过这些方法，你可以确保你的类的对象不会被意外地拷贝，从而避免可能出现的资源管理相关的错误。

2.7.3.8 拷贝构造函数总结

C++ 中拷贝构造函数需要注意的七个要点的表格：

要点	描述
定义和作用	拷贝构造函数在创建对象作为另一个现有对象副本时调用，**通常有一个对同类型对象的常量引用参数。**
语法	典型声明为 `ClassName(const ClassName &other)`。
深拷贝与浅拷贝	浅拷贝复制值，深拷贝创建资源的独立副本。对于包含指针的类，深拷贝通常必要。
规则三则 (Rule of Three)	如果实现了拷贝构造函数、拷贝赋值运算符或析构函数中的任何一个，通常应该实现所有三个。

要点	描述
避免不必要的拷贝	对于大型对象，使用移动语义避免不必要的拷贝，并在传递对象时使用引用或指针。
拷贝构造函数的隐式调用	不仅在显式复制时调用，也可能在将对象作为函数参数传递、从函数返回对象时隐式调用。
禁用拷贝构造函数	对于某些类，可以通过将拷贝构造函数声明为私有或使用 `delete` 关键字禁用拷贝。

2.7.4 使用初始化列表

在C++中，使用初始化列表来初始化类的字段是一种高效的初始化方式，尤其在构造函数中。初始化列表直接在对象的构造过程中初始化成员变量，而不是先创建成员变量后再赋值。这对于提高性能尤其重要，特别是在涉及到复杂对象或引用和常量成员的情况下。

初始化列表紧跟在构造函数参数列表后面，以冒号（`:`）开始，后跟一个或多个初始化表达式，每个表达式通常用逗号分隔。下面是使用初始化列表初始化字段的例子：

```
class MyClass {
private:
    int a;
    double b;
    std::string c;

public:
    // 使用初始化列表来初始化字段
    MyClass(int x, double y, const std::string& z) : a(x), b(y), c(z) {
        // 构造函数体
    }
};
```

在这个例子中，`MyClass` 有三个成员变量：`a`（`int` 类型）、`b`（`double` 类型）和 `c`（`std::string` 类型）。当创建 `MyClass` 的一个实例时，我们通过构造函数传递三个参数，这些参数被用于通过初始化列表直接初始化成员变量。初始化列表 `: a(x), b(y), c(z)` 的意思是用 `x` 初始化 `a`，用 `y` 初始化 `b`，用 `z` 初始化 `c`。

初始化列表的优点包括：

1. **效率**：对于非基本类型的对象，使用初始化列表比在构造函数体内赋值更高效，因为它避免了先默认构造然后再赋值的额外开销。
2. **必要性**：对于引用类型和常量类型的成员变量，必须使用初始化列表，因为这些类型的成员变量在构造函数体内不能被赋值。
3. **顺序**：成员变量的初始化顺序是按照它们在类中声明的顺序，而不是初始化列表中的顺序。

使用初始化列表是C++中推荐的初始化类成员变量的方式，因为它提供了更好的性能和灵活性。

2.7.5 this关键字

在 C++ 中，`this` 关键字是一个**指向调用对象的指针**。它在成员函数内部使用，用于引用调用该函数的对象。使用 `this` 可以明确指出成员函数正在操作的是哪个对象的数据成员。下面是一个使用 `Car` 类来展示 `this` 关键字用法的示例：

示例代码

下面的代码展示了如何使用 `this` 关键字：

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

using namespace std;

class Car {
private:
    string brand;
    int year;

public:
    Car(string brand, int year) {
        this->brand = brand;
        this->year = year;
        //        cout << "构造函数中：" << endl;
        //        cout << this << endl;
    }

    void display() const {
        cout << "Brand: " << this->brand << ", Year: " << this->year << endl;
        // 也可以不使用 this->，直接写 brand 和 year
    }

    Car& setYear(int year) {
        this->year = year; // 更新年份
        return *this; // 返回调用对象的引用
    }
};

int main()
{
    Car car("宝马",2024);
    car.display();

    // 链式调用
    car.setYear(2023).display();

    //    cout << "main函数中：" << endl;
    //    cout << &car << endl;

    //    Car car2("宝马",2024);
    //    cout << "main函数中：" << endl;
    //    cout << &car2 << endl;
    return 0;
}
```

在这个例子中，`Car` 类的构造函数使用 `this` 指针来区分成员变量和构造函数参数。同样，`setYear` 成员函数使用 `this` 指针来返回调用该函数的对象的引用，这允许链式调用，如 `myCar.setYear(2021).display();`。在 `main` 函数中创建了 `Car` 类型的对象，并展示了如何使用这些成员函数。

2.7.6 new关键字

在C++中，`new` 关键字用于动态分配内存。它是C++中处理动态内存分配的主要工具之一，允许在程序运行时根据需要分配内存。

基本用法

分配单个对象：使用 `new` 可以在堆上动态分配一个对象。例如，`new int` 会分配一个 `int` 类型的空间，并返回一个指向该空间的指针。

```
int* ptr = new int; //C语言中，int *p = (int *)malloc(sizeof(int));
```

分配对象数组：`new` 也可以用来分配一个对象数组。例如，`new int[10]` 会分配一个包含10个整数的数组。

```
int* arr = new int[10]; //C语言中，int *arr = (int *)malloc(sizeof(int)*10);
```

初始化：可以在 `new` 表达式中使用初始化。对于单个对象，可以使用构造函数的参数：

```
MyClass* obj = new MyClass(arg1, arg2);
```

与 `delete` 配对使用

使用 `new` 分配的内存必须显式地通过 `delete`（对于单个对象）或 `delete[]`（对于数组）来释放，以避免内存泄露：

释放单个对象：

```
delete ptr;  // 释放 ptr 指向的对象
```

释放数组：

```
delete[] arr;  // 释放 arr 指向的数组
```

注意事项

- **异常安全**：如果 `new` 分配内存失败，它会抛出 `std::bad_alloc` 异常（除非使用了 `nothrow` 版本）。
- **内存泄露**：忘记释放使用 `new` 分配的内存会导致内存泄露。
- **匹配使用 `delete` 和 `delete[]`**：为避免未定义行为，使用 `new` 分配的单个对象应该使用 `delete` 释放，使用 `new[]` 分配的数组应该使用 `delete[]` 释放。

示例代码

```
class MyClass {
public:
    MyClass() {
```

```
        std::cout << "Object created" << std::endl;
    }
};
int main() {
    // 分配单个对象
    MyClass* myObject = new MyClass();
    // 分配对象数组
    int* myArray = new int[5]{1, 2, 3, 4, 5};
    // 使用对象和数组...
    // 释放内存
    delete myObject;
    delete[] myArray;

    return 0;
}
```

在这个例子中，`new` 被用来分配一个 `MyClass` 类型的对象和一个整数数组，然后使用 `delete` 和 `delete[]` 来释放内存。每个 `new` 都对应一个 `delete`，保证了动态分配的内存被适当管理。

2.8 析构函数

2.8.1 什么是析构函数

析构函数是C++中的一个特殊的成员函数，它在对象生命周期结束时被自动调用，用于执行对象销毁前的清理工作。析构函数特别重要，尤其是在涉及动态分配的资源（如内存、文件句柄、网络连接等）的情况下。

基本特性

1. **名称**：析构函数的名称由波浪号（`~`）后跟类名构成，如 `~MyClass()`。
2. **无返回值和参数**：析构函数不接受任何参数，也不返回任何值。
3. **自动调用**：当对象的生命周期结束时（例如，一个局部对象的作用域结束，或者使用 `delete` 删除一个动态分配的对象），析构函数会被自动调用。
4. **不可重载**：每个类只能有一个析构函数。
5. **继承和多态**：如果一个类是多态基类，其析构函数应该是虚的。

示例

假设我们有一个类 `MyClass`，它包含了动态分配的内存或其他资源：

```
#include <iostream>

using namespace std;

class MyClass{

private:
    int* datas;

public:
    MyClass(int size){
        datas = new int[size];
```

```
    }

    ~MyClass(){
        cout << "析构函数被调用" << endl;
        delete[] datas;
    }
};
int main()
{
    MyClass m1(5);
    MyClass *m2 = new MyClass(10);

    delete m2;
    return 0;
}
```

在这个示例中，`MyClass` 的构造函数分配了一块内存，而析构函数释放了这块内存。当 `obj` 的生命周期结束时（即离开了它的作用域），`MyClass` 的析构函数被自动调用，负责清理资源，防止内存泄露。

重要性

析构函数在管理资源方面非常重要。没有正确实现析构函数，可能导致资源泄露或其他问题。在基于RAII（资源获取即初始化）原则的C++编程实践中，确保资源在对象析构时被适当释放是非常关键的。当使用智能指针和其他自动资源管理技术时，可以减少显式编写析构函数的需要，但了解析构函数的工作原理仍然很重要。

以下是关于 C++ 中析构函数需要了解的十个要点的表格：

标注粗体部分，是能快速上手的内容，方便后续QT的学习，而没有粗体的部分，会在QT结束后，如果安排C++深入讲解的课程的话，会安排到。

同理拷贝构造函数，考虑到学习策略安排，这里先陈列出来。

要点	**描述**
定义和作用	析构函数在对象生命周期结束时自动调用，用于清理对象可能持有的资源。
语法	析构函数名称由波浪线 (~) 后跟类名构成，例如 MyClass 的析构函数为 `~MyClass()`。
资源管理	用于释放对象在生命周期中分配的资源，如动态内存、文件句柄、网络连接等。
自动调用机制	当对象离开其作用域或通过 `delete` 删除时，将自动调用其析构函数。
防止资源泄露	正确实现析构函数对防止资源泄露至关重要，特别是在涉及动态资源分配的情况。
虚析构函数	如果类作为基类设计，应有一个虚析构函数，以确保正确调用派生类的析构函数。
析构函数与异常	析构函数不应抛出异常，如果可能抛出，应在函数内捕获。

要点	描述
删除的析构函数	可以通过将析构函数声明为删除（`~MyClass() = delete;`）来禁止删除某类对象。
与构造函数的关系	每个类只能有一个析构函数，不可重载，与构造函数相比。
规则三则/五则	如果类需要自定义析构函数、拷贝构造函数或拷贝赋值运算符，可能也需要自定义另外两个（规则三则）。在 C++11 后还包括移动构造函数和移动赋值运算符（规则五则）。

这个表格概括了在学习和使用 C++ 析构函数时应考虑的主要点

2.9 静态成员

2.9.1 静态成员的定义

静态成员在C++类中是一个重要的概念，它包括静态成员变量和静态成员函数。静态成员的特点和存在的意义如下：

静态成员变量

1. **定义**：静态成员变量是类的所有对象共享的变量。与普通成员变量相比，无论创建了多少个类的实例，静态成员变量只有一份拷贝。
2. **初始化**：静态成员变量需要在类外进行初始化，通常在类的实现文件中。
3. **访问**：静态成员变量可以通过类名直接访问，不需要创建类的对象。也可以通过类的对象访问。
4. **用途**：常用于存储类级别的信息（例如，计数类的实例数量）或全局数据需要被类的所有实例共享。

静态成员函数

1. **定义**：静态成员函数是可以不依赖于类的实例而被调用的函数。它不能访问类的非静态成员变量和非静态成员函数。
2. **访问**：类似于静态成员变量，静态成员函数可以通过类名直接调用，也可以通过类的实例调用。
3. **用途**：常用于实现与具体对象无关的功能，或访问静态成员变量。

示例代码

```
class MyClass {
public:
    static int staticValue; // 静态成员变量

    MyClass() {
        // 每创建一个对象，静态变量增加1
        staticValue++;
    }

    static int getStaticValue() {
        // 静态成员函数
```

```
        return staticValue;
    }
};

// 类外初始化静态成员变量
int MyClass::staticValue = 0;

int main() {
    MyClass obj1, obj2;
    std::cout << MyClass::getStaticValue(); // 输出2
}
```

存在的意义

- **共享数据**：允许对象之间共享数据，而不需要每个对象都有一份拷贝。
- **节省内存**：对于频繁使用的类，使用静态成员可以节省内存。
- **独立于对象的功能**：静态成员函数提供了一种在不创建对象的情况下执行操作的方法，这对于实现工具函数或管理类级别状态很有用。

2.9.2 静态成员变量的作用

静态成员变量在C++中的一个典型应用是用于跟踪类的实例数量。这个案例体现了静态成员变量的特性：它们在类的所有实例之间共享，因此适合于存储所有实例共有的信息。

下面是一个示例，展示了如何使用静态成员变量来计数一个类的实例数量：

```
#include <iostream>

using namespace std;

class Myclass{

private:
    static int staticNumofInstance;

public:
    Myclass(){
        staticNumofInstance++;
    }

    ~Myclass(){
        staticNumofInstance--;
    }

    static int getNunofInstance(){
        return staticNumofInstance;
    }
};

int Myclass::staticNumofInstance = 0;

int main()
{
    Myclass m1;
```

```
    cout << Myclass::getNunofInstance() << endl;
    Myclass m2;
    cout << m2.getNunofInstance() << endl;

    {
        Myclass m3;
        cout << Myclass::getNunofInstance() << endl;
        Myclass m4;
        cout << Myclass::getNunofInstance() << endl;
    }
    cout << Myclass::getNunofInstance() << endl;
    Myclass *m5 = new Myclass;
    cout << Myclass::getNunofInstance() << endl;
    delete m5;
    cout << Myclass::getNunofInstance() << endl;
    return 0;
}
```

在这个例子中：

- `Myclass` 类有一个静态成员变量 `staticNumofInstance`，用来跟踪该类的实例数量。
- 每当创建 `Myclass` 的新实例时，构造函数会增加 `staticNumofInstance`。
- 每当一个 `Myclass` 实例被销毁时，析构函数会减少 `staticNumofInstance`。
- 通过静态成员函数 `getNunofInstance` 可以随时获取当前的实例数量。
- 静态成员变量 `staticNumofInstance` 在类外初始化为0。

这个案例展示了静态成员变量如何在类的所有实例之间共享，并为所有实例提供了一个共同的状态（在这个例子中是实例的数量）。这种技术在需要跟踪对象数量或实现某种形式的资源管理时特别有用。

2.10 继承

2.10.1 继承基本概念

继承是面向对象编程（OOP）中的一个核心概念，特别是在C++中。它允许一个类（称为派生类或子类）继承另一个类（称为基类或父类）的属性和方法。继承的主要目的是实现代码重用，以及建立一种类型之间的层次关系。

特点

1. **代码重用**：子类继承了父类的属性和方法，减少了代码的重复编写。
2. **扩展性**：子类可以扩展父类的功能，添加新的属性和方法，或者重写（覆盖）现有的方法。
3. **多态性**：通过继承和虚函数，C++支持多态，允许在运行时决定调用哪个函数。

基本用法

在C++中，继承可以是公有（public）、保护（protected）或私有（private）的，这决定了基类成员在派生类中的访问权限。

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

//基类，父类
```

```
class Vehicle{ //交通工具，车,抽象的概念
public:
    string type;
    string contry;
    string color;
    double price;
    int numOfwheel;

    void run(){
        cout << "车跑起来了" << endl;
    }
    void stop();
};

//派生类，子类
class Bickle : public Vehicle{

};

//派生类，子类
class Roadster : public Vehicle{ //跑车，也是抽象，比父类感觉上范围缩小了点
public:
    int stateOfTop;

    void openTopped();
    void pdrifting();
};

int main()
{
    Roadster ftype;
    ftype.type = "捷豹Ftype";
    ftype.run();
    Bickle bike;
    bike.type = "死飞";
    bike.run();
    return 0;
}
```

在这个例子中，`Vehicle` 类公有地继承自 `Vehicle` 类，这意味着所有 `Vehicle` 类的公有成员在 `Vehicle` 类中也是公有的。

让我们用一个简单而有趣的案例来说明继承的概念：动物园中的动物。

想象我们正在创o在这个程序中，我们有一个基类 `Animal`，它定义了所有动物共有的特性和行为。然后，我们可以创建几个派生类，如 `Lion`、`Elephant` 和 `Bird`，这些类继承自 `Animal` 类，并添加或修改特定于它们自己的特性和行为。

基类：Animal

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

class Animal {
protected:
```

```
    std::string name;
    int age;

public:
    Animal(std::string n, int a) : name(n), age(a) {}

    virtual void makeSound() {
        std::cout << name << " makes a sound." << std::endl;
    }

    virtual void display() {
        std::cout << "Animal: " << name << ", Age: " << age << std::endl;
    }
};
```

派生类：Lion

```
class Lion : public Animal {
public:
    Lion(std::string n, int a) : Animal(n, a) {}

    void makeSound() override {
        std::cout << name << " roars." << std::endl;
    }

    void display() override {
        std::cout << "Lion: " << name << ", Age: " << age << std::endl;
    }
};
```

派生类：Elephant

```
class Elephant : public Animal {
public:
    Elephant(std::string n, int a) : Animal(n, a) {}

    void makeSound() override {
        std::cout << name << " trumpets." << std::endl;
    }

    void display() override {
        std::cout << "Elephant: " << name << ", Age: " << age << std::endl;
    }
};
```

派生类：Bird

```
class Bird : public Animal {
public:
    Bird(std::string n, int a) : Animal(n, a) {}

    void makeSound() override {
        std::cout << name << " sings." << std::endl;
    }

    void display() override {
        std::cout << "Bird: " << name << ", Age: " << age << std::endl;
    }
};
```

使用这些类

```
int main() {
    Lion lion("Leo", 5);
    Elephant elephant("Ella", 10);
    Bird bird("Bella", 2);

    lion.display();
    lion.makeSound();

    elephant.display();
    elephant.makeSound();

    bird.display();
    bird.makeSound();

    return 0;
}
```

在这个例子中：

- `Animal` 是基类，定义了所有动物共有的属性（如 `name` 和 `age`）和方法（如 `makeSound` 和 `display`）。
- `Lion`、`Elephant` 和 `Bird` 是派生类，它们继承了 `Animal` 的特性，并根据自身的特性重写了 `makeSound` 和 `display` 方法。
- 在 `main` 函数中，创建了各种动物的实例，并展示了它们的行为。

这个例子展示了继承如何使代码更有组织、更易于管理，并且如何通过重写基类方法来实现多态性。

2.10.2 权限对继承的影响

在C++中，访问控制符对继承的影响可以通过下表来清晰地展示。这个表格展示了不同类型的继承（`public`、`protected`、`private`）如何影响基类的不同类型成员（`public`、`protected`、`private`）在派生类中的访问级别。

基类成员类型	**`public` 继承**	**`protected` 继承**	**`private` 继承**
`public`	`public`	`protected`	`private`
`protected`	`protected`	`protected`	`private`

基类成员类型	`public` 继承	`protected` 继承	`private` 继承
`private`	不可访问	不可访问	不可访问

解释：

- `public` **继承**：基类的 `public` 成员在派生类中仍然是 `public` 的，`protected` 成员仍然是 `protected` 的。基类的 `private` 成员在派生类中不可访问。
- `protected` **继承**：基类的 `public` 和 `protected` 成员在派生类中都变成 `protected` 的。基类的 `private` 成员在派生类中不可访问。
- `private` **继承**：基类的 `public` 和 `protected` 成员在派生类中都变成 `private` 的。基类的 `private` 成员在派生类中不可访问。

这个表格提供了一个快速参考，帮助理解在不同类型的继承中基类成员的访问级别是如何变化的。记住，无论继承类型如何，基类的 `private` 成员始终不可直接在派生类中访问。

访问权限回顾

访问权限	类内部	同一个类的对象	派生类（子类）	类外部
public	✔ 可访问	✔ 可访问	✔ 可访问	✔ 可访问
private	✔ 可访问	✘ 不可访问	✘ 不可访问	✘ 不可访问
protected	✔ 可访问	✘ 不可访问	✔ 可访问	✘ 不可访问

课程上写的验证代码

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

//基类，父类
class Vehicle{ //交通工具，车,抽象的概念
public:
    string type;
    string contry;
    string color;
    double price;
    int numOfWheel;
protected:
    int protectedData;
private:
    int privateData;

public:
    void run(){
        cout << "车跑起来了" << endl;
    }
    void stop();
};

//私有继承测试
class TestClass : private Vehicle{
public:
```

```
    void tsetFunc(){
        price = 10; //基类的公有数据被私有继承后，在派生类中权限编程私有，只限在类内部使用
    }
};
//公有继承测试
class Truck : protected Vehicle{

public:
    void testFunc(){
        type = "数据测试"; //编程了公有权限
        protectedData = 10; //保持公有权限
        privateData = 10; //报错了，基类的私有成员，不管哪种方式的继承都是不可访问的。
    }
};

//公有继承，基类的公有权限和保护权限不变，私有成员不能访问
class Bickle : public Vehicle{

public:
    void testFunc(){
        protectedData = 10;
    }
};

//派生类，子类
class Roadster : public Vehicle{ //跑车，也是抽象，比父类感觉上范围缩小了点
public:
    int stateOfTop;

    void openTopped();
    void pdrifting();
};

int main()
{
    TestClass test;
    test.price = 3.3; //报错了，基类的公有成员被私有继承后，降为私有权限

    Truck t;
    t.type = "测试"; //报错了，基类的公有成员被保护继承后，降为保护权限
    t.protectedData = 10; //从报错信息看出，保护继承造成基类的保护成员还是保持保护权限
    Roadster ftype;
    ftype.type = "捷豹Ftype";
    ftype.run();
    Bickle bike;
    bike.type = "死飞";
    bike.run();
    return 0;
}
```

2.10.3 基类构造函数

在C++中，派生类可以通过其构造函数的初始化列表来调用基类的构造函数。这是在构造派生类对象时初始化基类部分的**标准做法**。

当创建派生类的对象时，基类的构造函数总是在派生类的构造函数之前被调用。如果没有明确指定，将调用基类的默认构造函数。如果基类没有默认构造函数，或者你需要调用一个特定的基类构造函数，**就需要在派生类构造函数的初始化列表中明确指定**。

示例

假设我们有一个基类 `Base` 和一个派生自 `Base` 的类 `Derived`：

```
class Base {
public:
    int data;

    Base(int x) {
        std::cout << "Base constructor with x = " << x << std::endl;
    }
};

class Derived : public Base {
public:
    double ydata;
    Derived(int x, double y) : Base(x) {  // 调用 Base 类的构造函数
        std::cout << "Derived constructor with y = " << y << std::endl;
    }
};

int main() {
    Derived obj(10, 3.14);  // 首先调用 Base(10)，然后调用 Derived 的构造函数
    return 0;
}
```

在这个例子中：

- `Base` 类有一个接受一个整数参数的构造函数。
- `Derived` 类继承自 `Base`，它的构造函数接受一个整数和一个双精度浮点数。在其初始化列表中，它调用 `Base` 类的构造函数，并传递整数参数。
- 当 `Derived` 类的对象被创建时，首先调用 `Base` 类的构造函数，然后调用 `Derived` 类的构造函数。

通过这种方式，派生类能够确保其基类部分被正确初始化。在继承层次结构中，这是非常重要的，特别是当基类需要一些特定的初始化操作时。

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

//基类，父类
class Vehicle{ //交通工具，车,抽象的概念
public:
    string contry;
    double price;
```

```
    Vehicle(string contry, double price){
        cout << "基类的构造函数被调用" << endl;
        this->contry = contry;
        this->price = price;
    };

    void run(){
        cout << "车跑起来了" << endl;
    }
    void stop();
};


//派生类，子类
class Roadster : public Vehicle{ //跑车，也是抽象，比父类感觉上范围缩小了点
public:
    int stateOfTop;
    Roadster(string contry, double price, int state) : Vehicle(contry, price){
        cout << "派生类的构造函数被调用" << endl;
        stateOfTop = state;
    }

    void openTopped();
    void pdrifting();
};

int main()
{

    Roadster FTYPE("法国",70,0);
    return 0;
}
```

2.10.4 虚函数

在C++中，`virtual` 和 `override` 关键字用于支持多态，尤其是在涉及类继承和方法重写的情况下。正确地理解和使用这两个关键字对于编写可维护和易于理解的面向对象代码至关重要。

virtual 关键字

1. **使用场景**：在基类中声明虚函数。
2. **目的**：允许派生类重写该函数，实现多态。
3. **行为**：当通过基类的指针或引用调用一个虚函数时，调用的是对象实际类型的函数版本。
4. **示例**：

```
class Base {
public:
    virtual void func() {
        std::cout << "Function in Base" << std::endl;
    }
};
```

override 关键字

1. **使用场景**：在派生类中重写虚函数。
2. **目的**：明确指示函数意图重写基类的虚函数。
3. **行为**：确保派生类的函数确实重写了基类中的一个虚函数。如果没有匹配的虚函数，编译器会报错。
4. **示例**：

```
class Derived : public Base {
public:
    void func() override {
        std::cout << "Function in Derived" << std::endl;
    }
};
```

注意点

- **只在派生类中使用 override**：`override` 应仅用于派生类中重写基类的虚函数。
- **虚析构函数**：如果类中有虚函数，通常应该将析构函数也声明为虚的。
- **默认情况下，成员函数不是虚的**：在C++中，成员函数默认不是虚函数。只有显式地使用 `virtual` 关键字才会成为虚函数。
- **继承中的虚函数**：一旦在基类中声明为虚函数，该函数在所有派生类中自动成为虚函数，无论是否使用 `virtual` 关键字。

正确使用 `virtual` 和 `override` 关键字有助于清晰地表达程序员的意图，并利用编译器检查来避免常见的错误，如签名不匹配导致的非预期的函数重写。

2.10.5 多重继承

在C++中，多重继承是一种允许一个类同时继承多个基类的特性。这意味着派生类可以继承多个基类的属性和方法。多重继承增加了语言的灵活性，但同时也引入了额外的复杂性，特别是当多个基类具有相同的成员时。

基本概念

在多重继承中，派生类继承了所有基类的特性。这包括成员变量和成员函数。如果不同的基类有相同名称的成员，则必须明确指出所引用的是哪个基类的成员。

示例

假设有两个基类 `ClassA` 和 `ClassB`，以及一个同时从这两个类继承的派生类 `Derived`：

```
class ClassA {
public:
    void displayA() {
```

```
        std::cout << "Displaying ClassA" << std::endl;
    }
};

class ClassB {
public:
    void displayB() {
        std::cout << "Displaying ClassB" << std::endl;
    }
};

class Derived : public ClassA, public ClassB {
public:
    void display() {
        displayA(); // 调用 ClassA 的 displayA
        displayB(); // 调用 ClassB 的 displayB
    }
};

int main() {
    Derived obj;
    obj.displayA();  // 调用 ClassA 的 displayA
    obj.displayB();  // 调用 ClassB 的 displayB
    obj.display();   // 调用 Derived 的 display
    return 0;
}
```

在这个示例中，`Derived` 类同时继承了 `ClassA` 和 `ClassB`。因此，它可以使用这两个类中定义的方法。

注意事项

- **菱形继承问题**：如果两个基类继承自同一个更高层的基类，这可能导致派生类中存在两份基类的副本，称为菱形继承（或钻石继承）问题。这可以通过虚继承来解决。
- **复杂性**：多重继承可能会使类的结构变得复杂，尤其是当继承层次较深或类中有多个基类时。
- **设计考虑**：虽然多重继承提供了很大的灵活性，但过度使用可能导致代码难以理解和维护。在一些情况下，使用组合或接口（纯虚类）可能是更好的设计选择。

多重继承是C++的一个强大特性，**但应谨慎使用**。合理地应用多重继承可以使代码更加灵活和强大，但不当的使用可能导致设计上的问题和维护困难。

2.10.6 虚继承

虚继承是C++中一种特殊的继承方式，主要用来解决多重继承中的菱形继承问题。在菱形继承结构中，一个类继承自两个具有共同基类的类时，会导致共同基类的成员在派生类中存在两份拷贝，这不仅会导致资源浪费，还可能引起数据不一致的问题。虚继承通过确保共同基类的单一实例存在于继承层次中，来解决这一问题。

菱形继承问题示例

考虑以下的类结构：

```
class Base {
public:
    int data;
```

```
};

class Derived1 : public Base {
    // 继承自 Base
};

class Derived2 : public Base {
    // 继承自 Base
};

class FinalDerived : public Derived1, public Derived2 {
    // 继承自 Derived1 和 Derived2
};
```

在这个例子中，`FinalDerived` 类通过 `Derived1` 和 `Derived2` 间接地继承自 `Base` 类两次。因此，它包含了两份 `Base` 的成员拷贝。

使用虚继承解决菱形继承问题

要解决这个问题，应使用虚继承：

```
class Base {
public:
    int data;
};

class Derived1 : virtual public Base {
    // 虚继承 Base
};

class Derived2 : virtual public Base {
    // 虚继承 Base
};

class FinalDerived : public Derived1, public Derived2 {
    // 继承自 Derived1 和 Derived2
};
```

通过将 `Derived1` 和 `Derived2` 对 `Base` 的继承声明为虚继承（`virtual public Base`），`FinalDerived` 类中只会有一份 `Base` 类的成员。无论通过 `Derived1` 还是 `Derived2` 的路径，访问的都是同一个 `Base` 类的成员。

特点和注意事项

- **初始化虚基类**：在使用虚继承时，虚基类（如上例中的 `Base` 类）只能由最派生的类（如 `FinalDerived`）初始化。
- **内存布局**：虚继承可能会改变类的内存布局，通常会增加额外的开销，比如虚基类指针。
- **设计考虑**：虚继承应谨慎使用，因为它增加了复杂性。在实际应用中，如果可以通过其他设计（如组合或接口）避免菱形继承，那通常是更好的选择。

虚继承是C++语言中处理复杂继承关系的一种重要机制，但它也带来了一定的复杂性和性能考虑。正确地使用虚继承可以帮助你建立清晰、有效的类层次结构。

C++ 继承相关的学习内容整理成表格的形式：

学习内容	描述
继承的基础	理解基类和派生类的概念，以及如何通过继承扩展类功能。了解不同继承类型（公有、私有、保护）及其影响。
构造函数和析构函数在继承中的行为	学习派生类如何调用基类的构造函数和析构函数，以及它们的调用顺序。
访问控制和继承	理解公有、私有和保护继承对成员访问权限的影响。掌握继承中的访问修饰符（public, protected, private）。
函数重写和多态	学习多态和如何通过虚函数实现它，了解如何重写基类方法，以及纯虚函数和抽象类的概念。
虚继承和解决菱形问题	理解菱形继承问题及其解决方式，学习如何使用虚继承。
C++11 新特性中的继承相关内容	理解和应用 `override` 和 `final` 关键字，了解移动语义在继承中的应用。
设计原则与最佳实践	学习正确使用继承的方法，区分何时使用继承，何时使用组合，以及面向对象设计原则的应用。
实际案例分析	通过分析和编写实际代码示例加深理解，研究设计模式中继承的应用。

这个表格概述了学习 C++ 继承的关键方面和内容，有助于系统地理解和应用继承的概念。

2.11 多态

多态的基本概念(polymorphic)

想象一下，你有一个遥控器（这就像是一个基类的指针），这个遥控器可以控制不同的电子设备（这些设备就像是派生类）。无论是电视、音响还是灯光，遥控器上的“开/关”按钮（这个按钮就像是一个虚函数）都能控制它们，但具体的操作（打开电视、播放音乐、开灯）则取决于你指向的设备。

2.11.1 如何实现多态

1. **使用虚函数（Virtual Function）**：
    - 我们在基类中定义一个虚函数，这个函数可以在任何派生类中被“重写”或者说“定制”。
    - 使用关键字 `virtual` 来声明。
2. **创建派生类并重写虚函数**：
    - 在派生类中，我们提供该虚函数的具体实现。这就像是告诉遥控器，“当你控制我的这个设备时，这个按钮应该这样工作”。
3. **通过基类的引用或指针调用虚函数**：
    - 当我们使用基类类型的指针或引用来调用虚函数时，实际调用的是对象的实际类型（派生类）中的函数版本。

视频课程中的手写案例

```
#include <iostream>

using namespace std;
```

```
class RemoteCon{
public:
    virtual void openUtils(){
        cout << "遥控器的开被按下" << endl;
    }
};

class TvRemoteCon : public RemoteCon{

public:
    void openUtils() override{
        cout << "电视遥控器的开被按下" << endl;
    }

    void testFunc(){

    }
};

class RoundspeakerCon : public RemoteCon{

public:
    void openUtils() override{
        cout << "音响遥控器的开被按下" << endl;
    }
};

class LightCon : public RemoteCon{

public:
    void openUtils() override{
        cout << "灯光遥控器的开被按下" << endl;
    }
};

void test(RemoteCon& r)//引用的方式
{
    r.openUtils();
}

int main()
{

    RemoteCon *remoteCon = new TvRemoteCon; //多态
    remoteCon->openUtils();

    RemoteCon *remoteCon2 = new TvRemoteCon; //多态
    remoteCon2->openUtils();

    RemoteCon *remoteCon3 = new LightCon; //多态
    remoteCon3->openUtils();

    TvRemoteCon tvRemote;
    test(tvRemote);
```

```
    return 0;
}
```

在这个例子中，不同的对象（`TvRemoteCon` 和 `TvRemoteCon`）以它们自己的方式"开"，尽管调用的是相同的函数 `openUtils`。这就是多态的魅力——相同的接口，不同的行为。

为什么使用多态

- **灵活性**：允许我们编写可以处理不确定类型的对象的代码。
- **可扩展性**：我们可以添加新的派生类而不必修改使用基类引用或指针的代码。
- **接口与实现分离**：我们可以设计一个稳定的接口，而将具体的实现留给派生类去处理。

2.11.2 抽象类

抽象类的基本概念

想象一下，你有一个"交通工具"的概念。这个概念告诉你所有交通工具都应该能做什么，比如移动（move），但它并不具体说明怎么移动。对于不同的交通工具，比如汽车和自行车，它们的移动方式是不同的。在这个意义上，"交通工具"是一个抽象的概念，因为它本身并不能直接被使用。你需要一个具体的交通工具，比如"汽车"或"自行车"，它们根据"交通工具"的概念具体实现了移动的功能。

在 C++ 中，抽象类就像是这样的一个抽象概念。它定义了一组方法（比如移动），但这些方法可能没有具体的实现。这意味着，抽象类定义了派生类应该具有的功能，但不完全实现这些功能。

抽象类的特点

1. **包含至少一个纯虚函数**：
   - 抽象类至少有一个纯虚函数。这是一种特殊的虚函数，在抽象类中没有具体实现，而是留给派生类去实现。
   - 纯虚函数的声明方式是在函数声明的末尾加上 `= 0`。
2. **不能直接实例化**：
   - 由于抽象类不完整，所以不能直接创建它的对象。就像你不能直接使用"交通工具"的概念去任何地方，你需要一个具体的交通工具。
3. **用于提供基础结构**：
   - 抽象类的主要目的是为派生类提供一个共同的基础结构，确保所有派生类都有一致的接口和行为。

```
#include <iostream>

using namespace std;

class Teacher{

public:
    string name;
    string shool;
    string major;

    virtual void goInClass() = 0;
    virtual void startTeaching() = 0;
```

```
    virtual void afterTeaching() = 0;
};

class EnglishTeacher : public Teacher{

public:
    void goInClass() override{
        cout << "英语老师开始进入教室" << endl;
    }
    void startTeaching() override{
        cout << "英语老师开始教学" << endl;
    }
    void afterTeaching() override{

    };
};

class ProTeacher : public Teacher{
public:
    void goInClass() override{
        cout << "编程老师开始进入教室" << endl;
    }
    void startTeaching() override{
        cout << "编程老师开始撸代码了，拒绝读PPT" << endl;
    }
    void afterTeaching() override{
        cout << "编程老师下课后手把手教x学员写代码" << endl;
    };
};

int main()
{
    // Teacher t;//抽象类，不支持被实例化
    EnglishTeacher e;
    e.goInClass();
    ProTeacher t;
    t.startTeaching();
    t.afterTeaching();
    //抽象类，多态
    Teacher *teacher = new ProTeacher;
    teacher->startTeaching();
    return 0;
}
```

2.11.3 纯虚函数-接口

在 C++ 中，虽然没有像其他编程语言（比如 Java 中的接口Interface）一样直接定义接口的关键字，但可以通过抽象类和纯虚函数的方式来实现接口的概念。

接口通常用于定义类应该实现的方法，但不提供具体实现。这样的实现方式允许多个类共享相同的接口，同时让每个类根据需要去实现这些接口。

一个类作为接口可以通过以下步骤来实现：

1. **定义抽象类**：创建一个包含纯虚函数的抽象类，这些函数构成了接口的一部分。这些函数在抽象类中只有声明而没有具体的实现。
2. **派生类实现接口**：派生类继承抽象类，并实现其中的纯虚函数，以具体实现接口定义的方法。

以下是一个简单的示例来展示如何使用抽象类来模拟接口：

```
#include <iostream>

using namespace std;

class BasketBallMove{
  public:
    virtual void passTheBall() = 0;
};

class LiveMove{

public:
        virtual void eat() = 0;
        virtual void bite() = 0;
        virtual void drink() = 0;
        virtual void la() = 0;
};

class Human : public LiveMove,BasketBallMove{
public:
    void eat() override{};
    void bite() override{};
    void drink() override{};
    void la() override{};

    void passTheBall() override{};
};

class Dog : public LiveMove{
public:
    void eat() override{};
    void bite() override{};
    void drink() override{};
    void la() override{};
};

int main()
{
    Human h;
    Dog g;
  //  LiveMove *l = new LiveMove;
    return 0;
}
```

通过这种方式，您可以在 C++ 中模拟出类似接口的行为，允许多个类共享相同的接口并提供各自的实现。

2.12 友元

2.12.1 什么是友元

在C++中，友元（friend）关键字用于给特定的外部函数或类访问某类的私有（private）和保护（protected）成员的权限。友元关系不是相互的，也不是可继承的。这意味着被声明为友元的函数或类可以访问原始类的私有和保护成员，但原始类不能访问友元的私有成员，除非它们也被声明为友元。

友元主要有三种类型：

1. **友元函数**：一个普通函数（不是类的成员函数）可以被声明为一个类的友元函数。它可以访问该类的所有成员（公有、保护和私有）。
2. **友元类**：一个类可以被声明为另一个类的友元。这意味着友元类的所有成员函数都可以访问原始类的保护和私有成员。
3. **友元成员函数**：一个类的成员函数可以被声明为另一个类的友元。这意味着该成员函数可以访问另一个类的保护和私有成员。

为什么使用友元

友元提供了一种机制，允许某些外部函数或类直接访问类的私有或保护成员。这在某些情况下非常有用，例如：

- 实现运算符重载，如重载输入输出运算符（`<<` 和 `>>`）。
- 允许两个不同类的对象之间进行密切的交互。
- 当类设计需要一种安全控制的方式来允许非成员函数或类访问私有成员时。

2.12.2 友元函数

示例

下面是一个展示友元函数的例子：在这个例子中，`showValue` 函数被声明为 `MyClass` 的友元函数，它可以访问 `MyClass` 的私有成员 `value`。

```
class MyClass {
private:
    int value;

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

    // 声明友元函数
    friend void showValue(MyClass& obj);
};

// 友元函数的定义
void showValue(MyClass& obj) {
    std::cout << "Value of MyClass: " << obj.value << std::endl;
}

int main() {
    MyClass obj(42);
    showValue(obj);  // 可以访问 MyClass 的私有成员
    return 0;
}
```

友元函数实现运算符重载，如重载输入输出运算符（`<<` 和 `>>`）。

在C++中，重载输入输出运算符（`<<` 和 `>>`）通常需要使用友元函数。这是因为输入输出运算符通常需要访问类的私有成员，同时又需要符合标准库中流对象（如 `std::ostream` 和 `std::istream`）的用法。

下面是一个如何使用友元函数来重载 `<<` 和 `>>` 运算符的例子：

示例

假设我们有一个 `Point` 类，表示二维空间中的一个点，我们想要重载 `<<` 和 `>>` 运算符以输出和输入点的坐标。

```
#include <iostream>

class Point {
private:
    int x, y;

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

    // 重载 << 运算符（输出）
    friend std::ostream& operator<<(std::ostream& os, const Point& point) {
        os << "(" << point.x << ", " << point.y << ")";
        return os;
    }
    // 重载 >> 运算符（输入）
    friend std::istream& operator>>(std::istream& is, Point& point) {
        is >> point.x >> point.y;
        return is;
    }
};

int main() {
    Point p1(1, 2);
    std::cout << "Point p1: " << p1 << std::endl;  // 输出点 p1

    Point p2;
    std::cout << "Enter coordinates for p2 (x y): ";
    std::cin >> p2;  // 输入点 p2
    std::cout << "Point p2: " << p2 << std::endl;  // 输出点 p2

    return 0;
}
```

在这个例子中：

- `<<` 运算符用于将 `Point` 对象的内容输出到输出流（如 `std::cout`）。
- `>>` 运算符用于从输入流（如 `std::cin`）中读取值到 `Point` 对象。
- 这两个运算符都被声明为 `Point` 类的友元函数，以便它们可以访问类的私有成员变量 `x` 和 `y`。

注意事项

虽然友元提供了强大的功能，但也应谨慎使用：

- 过度使用友元可能会破坏封装，使得代码难以维护和理解。
- 保持友元的数量最小，仅在确实需要时使用，以维持良好的封装性。

2.12.3 友元类

在C++中，一个类可以被声明为另一个类的友元类（friend class）。当一个类被声明为另一个类的友元类时，它可以访问后者的所有成员，包括私有（private）和保护（protected）成员。

友元类是一种强大的特性，它允许在保持封装性的同时，提供对类内部的深入访问。然而，由于它允许对另一个类的内部细节进行直接操作，因此应谨慎使用，以免破坏封装性。

示例

假设我们有两个类 `ClassA` 和 `ClassB`。我们可以使 `ClassB` 成为 `ClassA` 的友元类，这样 `ClassB` 就可以访问 `ClassA` 的所有成员，包括私有成员。

```
class ClassA {
private:
    int value;

public:
    ClassA(int v) : value(v) {}

    // 声明 ClassB 为友元类
    friend class ClassB;
};

class ClassB {
public:
    void showValue(ClassA& a) {
        // 可以访问 ClassA 的私有成员
        std::cout << "Value of ClassA: " << a.value << std::endl;
    }
};

int main() {
    ClassA a(100);
    ClassB b;
    b.showValue(a);  // 输出 "Value of ClassA: 100"
    return 0;
}
```

在这个示例中，`ClassB` 能够访问 `ClassA` 的私有成员 `value`，因为它被声明为 `ClassA` 的友元类。

注意事项

- **谨慎使用**：友元类应该谨慎使用，因为它们可能会破坏对象的封装性和隐藏性。
- **非相互性**：如果 `ClassA` 是 `ClassB` 的友元类，这并不意味着 `ClassB` 自动成为 `ClassA` 的友元类。友元关系是单向的。
- **非继承性**：友元关系不会被继承。如果类 `C` 继承自类 `B`，并且 `B` 是 `A` 的友元类，那么 `C` 不自动成为 `A` 的友元类，除非 `A` 明确声明 `C` 为友元类。

友元类提供了一种机制，允许其他类访问私有成员，但应在不破坏封装性的前提下谨慎使用。

2.12.4 友元成员函数

在C++中，除了可以将整个类或单独的函数声明为友元外，还可以将特定类中的某个成员函数单独声明为另一个类的友元。这样做可以让这个成员函数访问另一个类的所有成员（包括私有和受保护的成员），而无需将整个类声明为友元，从而提供了更细粒度的访问控制。

示例

假设我们有两个类 `ClassA` 和 `ClassB`，我们希望 `ClassB` 的一个特定成员函数 `showValue` 能够访问 `ClassA` 的私有成员。我们可以将 `ClassB` 的 `showValue` 函数声明为 `ClassA` 的友元。

```
#include <iostream>

class ClassB;  // 前向声明

class ClassA {
private:
    int value;

public:
    ClassA(int v) : value(v) {}

    // 声明 ClassB 的成员函数为友元
    friend void ClassB::showValue(ClassA& a);
};

class ClassB {
public:
    void showValue(ClassA& a) {
        // 访问 ClassA 的私有成员
        std::cout << "Value of ClassA: " << a.value << std::endl;
    }
};

int main() {
    ClassA a(100);
    ClassB b;
    b.showValue(a);  // 输出 "Value of ClassA: 100"
    return 0;
}
```

在这个示例中，`ClassB::showValue` 成为 `ClassA` 的友元函数。这意味着 `showValue` 可以访问 `ClassA` 的私有成员 `value`。注意，我们在 `ClassA` 前提供了 `ClassB` 的前向声明。这是必须的，因为在声明 `ClassA` 时 `ClassB` 还未完全定义。

注意事项

- **精确控制**：与将整个类声明为友元相比，将特定成员函数声明为友元可以更精确地控制访问权限。
- **破坏封装**：即使是单个成员函数，过度使用友元也可能破坏类的封装性。应当谨慎使用。
- **前向声明**：在一个类中声明另一个类的成员函数为友元时，可能需要前向声明另一个类，特别是在两个类相互引用对方的成员函数时。

友元成员函数是C++中一种强大的特性，允许开发者在保持类封装性的同时提供必要的访问权限。然而，应当谨慎使用，以保持代码的清晰性和维护性。

2.13 模板

2.13.1 类模板

在 C++ 中，模板（Template）是一种通用的编程工具，允许程序员编写**泛型代码**，使得**类或函数能够适用于多种不同的数据类型**而不需要重复编写相似的代码。C++ 提供了两种主要类型的模板：类模板和函数模板。

类模板（Class Templates）：

类模板允许定义通用的类，其中某些类型可以作为参数。这样的类可以处理不同类型的数据，而不需要为每个数据类型编写单独的类。

```
// 定义一个通用的类模板
template <typename T>
class MyTemplate {
private:
    T data;

public:
    MyTemplate(T d) : data(d) {}

    T getData() {
        return data;
    }
};

int main() {
    // 使用类模板创建对象
    MyTemplate<int> intObject(5);
    MyTemplate<std::string> stringObject("Hello");

    // 调用类模板中的函数
    std::cout << intObject.getData() << std::endl; // 输出 5
    std::cout << stringObject.getData() << std::endl; // 输出 Hello

    return 0;
}
```

2.13.2 函数模板

函数模板允许编写通用的函数，可以处理多种不同类型的数据。

```
// 定义一个通用的函数模板
template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    // 使用函数模板调用通用函数
    int result1 = add(5, 10);
    double result2 = add(3.5, 2.7);
```

```
    // 输出函数模板调用结果
    std::cout << "Result 1: " << result1 << std::endl; // 输出 Result 1: 15
    std::cout << "Result 2: " << result2 << std::endl; // 输出 Result 2: 6.2

    return 0;
}
```

模板提供了一种在编写代码时更具通用性的方法，能够处理不同类型的数据而无需为每种类型编写特定的函数或类。通过使用模板，可以提高代码的重用性和可维护性。

2.13.3 模板特化

模板特化（Template Specialization）是 C++ 中模板的一个概念，它允许针对特定的数据类型或特定的模板参数提供定制化的实现。模板特化允许您为模板提供一个特殊的实现，以覆盖或扩展默认的模板行为。

有两种类型的模板特化：完全特化（Full Specialization）和部分特化（Partial Specialization）。

完全特化（Full Specialization）：

完全特化是对模板中的所有模板参数都进行特化的情况。在完全特化中，模板参数被指定为特定的类型，为特定的类型提供独特的实现。

以下是一个示例，演示了对模板函数的完全特化：

```
#include <iostream>

// 定义一个通用的模板函数
template <typename T>
T maximum(T a, T b) {
    return (a > b) ? a : b;
}

// 对模板函数进行完全特化，针对 char* 类型
template <>
const char* maximum<const char*>(const char* a, const char* b) {
    return strcmp(a, b) > 0 ? a : b;
}

int main() {
    int intMax = maximum(3, 5); // 调用模板函数
    std::cout << "Maximum of integers: " << intMax << std::endl;

    const char* charMax = maximum("apple", "orange"); // 调用特化的函数
    std::cout << "Maximum of strings: " << charMax << std::endl;

    return 0;
}
```

在这个示例中，`maximum` 是一个模板函数，可以比较不同类型的数据，并返回较大的值。然后，为了特化针对 `const char*` 类型，我们提供了一个特化版本的 `maximum` 函数，该函数使用 `strcmp` 函数来比较字符串，并返回较大的字符串。

部分特化（Partial Specialization）：

部分特化是指对模板中的部分参数进行特化，允许更具体地特化某些模板参数。这通常在模板类中使用。

以下是一个示例，展示了对模板类的部分特化：

```
#include <iostream>

// 定义一个通用的模板类
template <typename T, typename U>
class MyTemplate {
public:
    void display() {
        std::cout << "Generic Display" << std::endl;
    }
};

// 对模板类进行部分特化，针对特定的类型组合
template <typename T>
class MyTemplate<T, int> {
public:
    void display() {
        std::cout << "Specialized Display for T and int" << std::endl;
    }
};

int main() {
    MyTemplate<float, double> obj1;
    obj1.display(); // 输出 "Generic Display"

    MyTemplate<int, int> obj2;
    obj2.display(); // 输出 "Specialized Display for T and int"

    return 0;
}
```

在这个示例中，`MyTemplate` 是一个模板类，然后我们对 `MyTemplate` 进行部分特化，当第二个模板参数是 `int` 类型时，提供了特殊的实现。因此，对于特定的类型组合，我们可以提供自定义的实现。

2.14 标准模板库STL

2.14.1 容器

当谈到 C++ 的标准模板库（STL）容器时，通常有一些常见的容器类型。以下是一些常见的 STL 容器及其特点的简要总结：

容器类型	描述	特点
vector	动态数组，支持快速随机访问和尾部插入/删除	支持动态大小调整，适用于需要随机访问和动态增删元素的场景
list	双向链表，支持快速插入/删除操作	插入/删除元素快速，但不支持随机访问元素

容器类型	描述	特点
deque	双端队列，支持在两端快速插入/删除操作	支持在头尾快速插入/删除元素，不同于 vector 在头部操作时效率较低，但随机访问效率比 list 高
stack	后进先出（LIFO）的堆栈数据结构	基于 deque 或 vector 实现，只允许在栈顶进行插入和删除操作
queue	先进先出（FIFO）的队列数据结构	基于 deque 或 list 实现，只允许在队尾进行插入，在队头进行删除操作
priority_queue	优先队列，按照一定顺序维护元素	基于 vector 实现，默认情况下是最大堆，可以通过自定义比较函数来实现不同的优先级顺序
set	有序不重复元素集合，基于红黑树实现	自动排序，插入/查找元素的平均时间复杂度为 O(log n)，不允许重复元素
multiset	有序可重复元素集合，基于红黑树实现	允许存储重复元素，按序存储，插入/查找元素的平均时间复杂度为 O(log n)
map	键值对集合，基于红黑树实现	存储键值对，按键自动排序，不允许重复键，插入/查找元素的平均时间复杂度为 O(log n)
multimap	键值对集合，允许重复键，基于红黑树实现	允许存储重复键值对，按键自动排序，插入/查找元素的平均时间复杂度为 O(log n)
unordered_set	无序不重复元素集合，基于哈希表实现	不按顺序存储元素，插入/查找/删除元素的平均时间复杂度为 O(1)，不允许重复元素
unordered_multiset	无序可重复元素集合，基于哈希表实现	允许存储重复元素，不按顺序存储，插入/查找/删除元素的平均时间复杂度为 O(1)
unordered_map	无序键值对集合，基于哈希表实现	键值对无序存储，插入/查找/删除元素的平均时间复杂度为 O(1)，不允许重复键
unordered_multimap	无序键值对集合，允许重复键，基于哈希表实现	允许存储重复键值对，键值对无序存储，插入/查找/删除元素的平均时间复杂度为 O(1)

2.14.2 vector

在C++的标准模板库（STL）中，`std::vector` 是一种动态数组容器。它提供了动态大小的数组功能，能够在运行时根据需要自动调整大小，允许在其尾部高效地进行元素的插入和删除操作。

1. **动态大小：** `std::vector` 允许动态增加或减少其大小。它会自动处理内存分配和释放，无需手动管理内存。
2. **随机访问：** 支持使用索引进行快速的随机访问，因为它底层基于连续的内存块。
3. **尾部插入/删除：** 在数组的尾部插入或删除元素的操作非常高效，时间复杂度为常数时间（Amortized Constant Time）。
4. **连续内存存储：** `std::vector` 中的元素在内存中是连续存储的，这有助于提高访问速度和缓存利用率。

5. **动态增长策略：** 当向 `std::vector` 添加元素时，如果当前容量不足，它会动态地重新分配更大的内存空间，并将现有元素复制到新的内存位置。这种动态增长策略确保了插入操作的高效性。

`std::vector` 的缺点是，在执行插入或删除操作时，如果不是在容器的末尾进行，可能会导致较高的时间复杂度，因为需要移动后续元素。

以下是一个使用 C++ STL 中的 `vector` 容器的简单示例：

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

int main() {
    // 创建一个空的 vector 容器
    std::vector<int> myVector;

    // 向 vector 容器尾部添加元素
    myVector.push_back(3);
    myVector.push_back(7);
    myVector.push_back(12);

    // 使用迭代器遍历 vector 容器并输出其中的元素
    // 在 C++ 中，auto 是一个关键字，用于声明变量时的类型推断。
    // 它允许编译器根据变量的初始化表达式推断出变量的类型，从而简化代码书写过程。
    std::cout << "Vector elements: ";
    for (auto it = myVector.begin(); it != myVector.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    // 获取 vector 容器的大小和访问特定位置的元素
    std::cout << "Vector size: " << myVector.size() << std::endl;
    std::cout << "Element at index 1: " << myVector[1] << std::endl;

    // 修改特定位置的元素
    myVector[2] = 20;

    // 使用范围-based for 循环遍历 vector 并输出元素
    std::cout << "Modified Vector elements: ";
    for (int num : myVector) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    // 清空 vector 容器
    myVector.clear();

    // 检查 vector 是否为空
    if (myVector.empty()) {
        std::cout << "Vector is empty." << std::endl;
    } else {
        std::cout << "Vector is not empty." << std::endl;
    }

    return 0;
}
```

这个例子展示了如何创建 `vector` 容器、向其尾部添加元素、使用迭代器和索引访问元素、修改元素值、清空容器以及检查容器是否为空。`vector` 是一个动态数组，可以根据需要动态地增加或减少其大小，适合需要随机访问元素且频繁进行尾部插入/删除操作的场景。

当涉及到 `std::vector` 的 API 方法时，参数列表、返回值以及参数的说明可以帮助更清楚地了解每个方法的使用和含义。以下是 `std::vector` 常用 API 列表，包括参数列表、返回值和参数说明：

API	描述	函数原型	参数说明
`push_back()`	在 `vector` 尾部添加一个元素	`void push_back(const T& value);`	`value`：要添加到尾部的元素
`pop_back()`	删除 `vector` 尾部的一个元素	`void pop_back();`	无参数
`size()`	返回 `vector` 中元素的数量	`size_type size() const noexcept;`	无参数
`capacity()`	返回 `vector` 当前可容纳的元素数量	`size_type capacity() const noexcept;`	无参数
`resize()`	改变 `vector` 的大小，可以增加或减少元素数量	`void resize(size_type count);` `void resize(size_type count, const T& value);`	`count`：新的 `vector` 大小 `value`：若添加元素，初始化值为 `value`
`reserve()`	修改 `vector` 的容量，预留足够的存储空间	`void reserve(size_type new_cap);`	`new_cap`：新的 `vector` 容量
`clear()`	清空 `vector` 中的所有元素	`void clear() noexcept;`	无参数
`empty()`	检查 `vector` 是否为空	`bool empty() const noexcept;`	无参数
`at()`	返回指定位置的元素，并进行边界检查	`reference at(size_type pos);` `const_reference at(size_type pos) const;`	`pos`：要访问的位置。如果超出范围，会引发 `std::out_of_range` 异常

API	描述	函数原型	参数说明
`operator[]`	重载操作符，用于访问指定位置的元素	`reference operator[](size_type pos);` `const_reference operator[](size_type pos) const;`	`pos`：要访问的位置。不进行边界检查，如果超出范围，行为未定义
`front()`	返回 `vector` 中第一个元素的引用	`reference front();` `const_reference front() const;`	无参数
`back()`	返回 `vector` 中最后一个元素的引用	`reference back();` `const_reference back() const;`	无参数
`begin()`	返回指向 `vector` 第一个元素的迭代器	`iterator begin() noexcept;` `const_iterator begin() const noexcept;`	无参数
`end()`	返回指向 `vector` 末尾（最后一个元素的后面）的迭代器	`iterator end() noexcept;` `const_iterator end() const noexcept;`	无参数
`rbegin()`	返回指向 `vector` 最后一个元素的逆向迭代器（逆向开始迭代）	`reverse_iterator rbegin() noexcept;` `const_reverse_iterator rbegin() const noexcept;`	无参数
`rend()`	返回指向 `vector` 第一个元素之前的逆向迭代器（逆向结束迭代）	`reverse_iterator rend() noexcept;` `const_reverse_iterator rend() const noexcept;`	无参数
`insert()`	在指定位置插入一个或多个元素	`iterator insert(const_iterator pos, const T& value);` `void insert(const_iterator pos, size_type count, const T& value);`	`pos`：插入位置 `value`：要插入的元素 `count`：要插入的元素个数

API	描述	函数原型	参数说明
`erase()`	删除指定位置或指定范围内的一个或多个元素	`iterator erase(const_iterator pos);` `iterator erase(const_iterator first, const_iterator last);`	`pos`：要删除的元素位置或范围的起始位置 `first`、`last`：要删除的范围
`swap()`	交换两个 `vector` 容器的内容	`void swap(vector& other);`	`other`：要交换内容的另一个 `vector` 容器
`emplace()`	在指定位置就地构造一个元素	`iterator emplace(const_iterator pos, Args&&... args);`	`pos`：就地构造的位置 `args`：构造元素所需的参数
`emplace_back()`	在 `vector` 尾部就地构造一个元素	`void emplace_back(Args&&... args);`	`args`：构造元素所需的参数

这些详细说明包括每个函数的参数、返回值以及对参数的解释，有助于更清晰地理解 `std::vector` 中常用 API 的使用方式和含义。

2.14.3 list

STL 中的 `list` 是双向链表（doubly linked list）的实现，它是 C++ 标准模板库中的一个容器，提供了一种能够高效进行插入、删除操作的数据结构。与 `vector` 不同，`list` 不支持随机访问，但它允许在任意位置快速插入和删除元素。

以下是关于 `std::list` 的一些特点和说明：

- **双向链表结构：** `std::list` 使用双向链表来组织其元素，每个节点都包含指向前一个节点和后一个节点的指针，因此在任意位置进行插入和删除操作的开销较小。
- **不支持随机访问：** 与 `vector` 不同，`list` 不支持通过索引直接访问元素，因为它不具备随机访问能力。要访问 `list` 中的元素，需要使用迭代器进行顺序遍历。
- **动态大小调整：** `list` 具有动态大小调整的特性，可以动态增加或减少元素的数量。对于大量的插入和删除操作，`list` 往往比 `vector` 更高效。
- **迭代器操作：** 使用迭代器可以对 `list` 中的元素进行访问、插入和删除。`list` 提供了 `begin()`、`end()`、`rbegin()`、`rend()` 等迭代器相关方法，支持正向和逆向迭代。
- **插入和删除操作效率高：** 在 `list` 中，在任意位置进行插入和删除操作的时间复杂度是 O(1)，因为只需要调整相邻节点的指针，无需移动大量元素。
- **空间开销：** 相比于 `vector`，`list` 需要额外的空间来存储指向前一个和后一个节点的指针，可能会导致更高的存储开销。

下面是一个简单的示例，演示了如何使用 STL 中的 `std::list` 容器。在这个案例中，我们创建了一个 `std::list` 来存储整数，并展示了一些基本的操作，如插入、删除、迭代等。

```cpp
#include <iostream>
#include <list>

int main() {
    // 创建一个存储整数的 list 容器
    std::list<int> myList;

    // 在 list 尾部插入元素
    myList.push_back(10);
    myList.push_back(20);
    myList.push_back(30);

    // 在 list 头部插入元素
    myList.push_front(5);
    myList.push_front(15);

    // 使用迭代器遍历 list 并输出元素
    std::cout << "List elements: ";
    for (auto it = myList.begin(); it != myList.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    // 删除 list 中特定的元素值
    myList.remove(20);

    // 使用范围-based for 循环遍历 list 并输出元素
    std::cout << "List elements after removal: ";
    for (int num : myList) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    // 检查 list 是否为空
    if (myList.empty()) {
        std::cout << "List is empty." << std::endl;
    } else {
        std::cout << "List is not empty." << std::endl;
    }

    return 0;
}
```

这个示例演示了如何创建 `std::list` 容器，并对其进行插入、删除和迭代操作。在实际应用中，`std::list` 还有许多其他的功能和方法可以使用，比如 `splice()`、`merge()`、`sort()` 等，用于更复杂的操作。

以下是 `std::list` 常用的 API 方法，包括参数说明，整理成表格形式：

API	描述	函数原型	参数说明
`push_back()`	在 `list` 尾部添加一个元素	`void push_back(const T& value);`	`value`：要添加到尾部的元素

API	描述	函数原型	参数说明
`push_front()`	在 `list` 头部添加一个元素	`void push_front(const T& value);`	`value`：要添加到头部的元素
`pop_back()`	删除 `list` 尾部的一个元素	`void pop_back();`	无参数
`pop_front()`	删除 `list` 头部的一个元素	`void pop_front();`	无参数
`size()`	返回 `list` 中元素的数量	`size_type size() const noexcept;`	无参数
`empty()`	检查 `list` 是否为空	`bool empty() const noexcept;`	无参数
`clear()`	清空 `list` 中的所有元素	`void clear() noexcept;`	无参数
`begin()`	返回指向 `list` 第一个元素的迭代器	`iterator begin() noexcept;` `const_iterator begin() const noexcept;`	无参数
`end()`	返回指向 `list` 末尾（最后一个元素的后面）的迭代器	`iterator end() noexcept;` `const_iterator end() const noexcept;`	无参数
`rbegin()`	返回指向 `list` 最后一个元素的逆向迭代器（逆向开始迭代）	`reverse_iterator rbegin() noexcept;` `const_reverse_iterator rbegin() const noexcept;`	无参数
`rend()`	返回指向 `list` 第一个元素之前的逆向迭代器（逆向结束迭代）	`reverse_iterator rend() noexcept;` `const_reverse_iterator rend() const noexcept;`	无参数
`insert()`	在指定位置插入一个或多个元素	`iterator insert(const_iterator pos, const T& value);` `void insert(const_iterator pos, size_type count, const T& value);`	`pos`：插入位置 `value`：要插入的元素 `count`：要插入的元素个数

API	描述	函数原型	参数说明
`erase()`	删除指定位置或指定范围内的一个或多个元素	`iterator erase(const_iterator pos);` `iterator erase(const_iterator first, const_iterator last);`	`pos`：要删除的元素位置或范围的起始位置 `first`、`last`：要删除的范围

`splice()`	在指定位置插入另一个 `list` 中的元素	`void splice(const_iterator pos, list& other);`
		`void splice(const_iterator pos, list& other, const_iterator it);`
		`void splice(const_iterator pos, list& other, const_iterator first, const_iterator last);`
			`pos`：插入位置
			`other`：要插入的另一个 `list`
			`it`：要插入的元素
			`first`、`last`：要插入的范围

`merge()`	合并两个已排序的 `list`	`void merge(list& other);` `void merge(list&& other);`	`other`：要合并的另一个 `list`
`unique()`	移除 `list` 中重复的元素	`void unique();` `void unique(BinaryPredicate p);`	`p`：可选的谓词函数，用于比较元素是否相等
`sort()`	对 `list` 进行排序	`void sort();` `void sort(Compare comp);`	`comp`：可选的比较函数，用于元素排序

这些方法使得在 `std::list` 上进行插入、删除、迭代和操作变得方便。每个方法都有不同的参数和作用，可根据需要选择合适的方法来操作 `std::list` 容器。

2.14.4 set

`std::set` 是 C++ 标准模板库中的关联容器，用于存储唯一值的集合。它基于红黑树实现，保持了元素的有序性，且不允许重复的元素存在。

以下是关于 `std::set` 的一些特点和说明：

- **唯一性**：`std::set` 中的元素是唯一的，不允许有重复的元素存在。当尝试向 `set` 中插入重复的元素时，新元素将不会被插入。
- **有序性**：`std::set` 中的元素是根据元素值进行排序的，这使得元素按照一定顺序存储，并且支持对元素的快速搜索。
- **红黑树实现**：`std::set` 的底层实现通常是基于红黑树的，这保证了插入、删除和查找操作的时间复杂度为对数时间（O(log n)）。

- **动态操作：** 可以对 `std::set` 进行动态操作，如插入、删除和查找元素。插入和删除操作的性能较好，不会影响其他元素的位置。

以下是 `std::set` 常用的一些方法：

- `insert()`：向 `set` 中插入一个元素。
- `erase()`：删除 `set` 中指定值的元素。
- `find()`：查找指定值在 `set` 中的位置。
- `size()`：返回 `set` 中元素的数量。
- `empty()`：检查 `set` 是否为空。
- `clear()`：清空 `set` 中的所有元素。

下面是一个简单的示例，演示了如何使用 `std::set`：

```
#include <iostream>
#include <set>

int main() {
    // 创建一个存储 int 类型值的 set 容器
    std::set<int> mySet;

    // 向 set 中插入元素
    mySet.insert(10);
    mySet.insert(20);
    mySet.insert(30);

    // 尝试插入重复元素
    mySet.insert(20); // 不会插入重复的元素

    // 使用迭代器遍历 set 并输出元素
    std::cout << "Set elements: ";
    for (auto it = mySet.begin(); it != mySet.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    // 查找特定值在 set 中的位置
    int searchValue = 20;
    auto found = mySet.find(searchValue);
    if (found != mySet.end()) {
        std::cout << "Found " << searchValue << " in the set." << std::endl;
    } else {
        std::cout << searchValue << " not found in the set." << std::endl;
    }

    // 删除特定值的元素
    mySet.erase(30);

    // 检查 set 是否为空
    if (mySet.empty()) {
        std::cout << "Set is empty." << std::endl;
    } else {
        std::cout << "Set is not empty." << std::endl;
```

```
    }

    return 0;
}
```

这个示例演示了如何创建 `std::set` 容器，并对其进行插入、删除、查找等操作。`std::set` 是一个非常有用的容器，适用于需要存储唯一值的场景。

2.14.5 map

`std::map` 是 C++ 标准模板库中的关联容器，用于**存储键值对**。它基于红黑树实现，保持了元素的有序性，其中每个元素都是一个键值对，键和值之间存在映射关系。

以下是关于 `std::map` 的一些特点和说明：

- **有序性：** `std::map` 中的元素是根据键值排序的，这使得元素按照一定顺序存储，并且支持对元素的快速搜索。
- **唯一键：** `std::map` 中的键是唯一的，每个键对应一个值。如果尝试使用相同的键向 `map` 中插入值，则会更新键对应的值。
- **红黑树实现：** `std::map` 的底层实现通常是基于红黑树的，这保证了插入、删除和查找操作的时间复杂度为对数时间（O(log n)）。
- **动态操作：** 可以对 `std::map` 进行动态操作，如插入、删除和查找键值对。插入和删除操作的性能较好，不会影响其他元素的位置。

以下是 `std::map` 常用的一些方法：

- `insert()`：向 `map` 中插入一个键值对。
- `erase()`：删除 `map` 中指定键的键值对。
- `find()`：查找指定键在 `map` 中的位置。
- `operator[]`：通过键访问对应的值。
- `size()`：返回 `map` 中键值对的数量。
- `empty()`：检查 `map` 是否为空。
- `clear()`：清空 `map` 中的所有键值对。

下面是一个简单的示例，演示了如何使用 `std::map`：

```
#include <iostream>
#include <map>

int main() {
    // 创建一个存储 string 类型键和 int 类型值的 map 容器
    std::map<std::string, int> myMap;

    // 向 map 中插入键值对
    myMap["Alice"] = 25;
    myMap["Bob"] = 30;
    myMap["Charlie"] = 20;

    // 使用迭代器遍历 map 并输出键值对
```

```
    std::cout << "Map elements: " << std::endl;
    for (auto it = myMap.begin(); it != myMap.end(); ++it) {
        std::cout << it->first << ": " << it->second << std::endl;
    }

    // 查找特定键在 map 中的位置
    std::string searchKey = "Bob";
    auto found = myMap.find(searchKey);
    if (found != myMap.end()) {
        std::cout << "Found " << searchKey << " with value: " << found->second <<
std::endl;
    } else {
        std::cout << searchKey << " not found in the map." << std::endl;
    }

    // 删除特定键的键值对
    myMap.erase("Charlie");

    // 检查 map 是否为空
    if (myMap.empty()) {
        std::cout << "Map is empty." << std::endl;
    } else {
        std::cout << "Map is not empty." << std::endl;
    }

    return 0;
}
```

这个示例演示了如何创建 `std::map` 容器，并对其进行插入、删除、查找等操作。`std::map` 是一个非常有用的容器，适用于需要键值对存储和检索的场景。

2.15 异常

2.15.1 异常基本

在 C++ 中，异常处理是一种机制，用于处理程序在运行时发生的异常情况。异常是指程序执行期间发生的意外事件，比如除以零、访问无效的内存地址等。通过使用异常处理机制，可以使程序更健壮，并能够处理这些意外情况，避免程序崩溃或产生不可预测的结果。

在 C++ 中，异常处理通常包括以下关键词和概念：

- **try-catch 块**：`try` 块用于标识可能会引发异常的代码块，而 `catch` 块用于捕获和处理异常。`catch` 块可以针对不同类型的异常进行处理。
- **throw 关键词**：`throw` 用于在程序中显式抛出异常。当发生异常情况时，可以使用 `throw` 来抛出一个特定的异常类型。
- **异常类型**：异常可以是任何类型的数据，但通常是标准库中的异常类或自定义的异常类。标准库提供了一些常见的异常类，如 `std::exception` 及其派生类，用于表示不同类型的异常情况。

下面是一个简单的示例，演示了异常处理的基本用法：

```
#include <iostream>

void divide(int numerator, int denominator) {
    try {
```

```
        if (denominator == 0) {
            throw "Division by zero is not allowed!";
        }
        int result = numerator / denominator;
        std::cout << "Result of division: " << result << std::endl;
    } catch (const char* errorMessage) {
        std::cout << "Exception caught: " << errorMessage << std::endl;
    }
}

int main() {
    int a = 10;
    int b = 0;

    divide(a, b);

    return 0;
}
```

在这个示例中，`divide()` 函数尝试对 `numerator` 除以 `denominator` 进行除法运算。如果 `denominator` 为零，就会抛出一个字符串类型的异常。在 `main()` 函数中调用 `divide()` 函数时，由于 `b` 的值为零，因此会抛出异常，然后在 `catch` 块中捕获并处理异常，输出错误消息。

在实际的程序开发中，可以根据具体情况设计和抛出自定义的异常类，以及使用多个 `catch` 块来处理不同类型的异常，使程序能够更好地处理各种异常情况。

2.15.2 自定义异常

在 C++ 中，你可以通过继承标准库的 `std::exception` 类或其派生类来自定义异常类。自定义异常类通常用于表示特定类型的异常情况，并允许你提供有关异常的额外信息。

以下是一个示例，演示了如何创建自定义的异常类：

```
#include <iostream>
#include <exception>

// 自定义异常类，继承自 std::exception
class MyException : public std::exception {
private:
    const char* message; // 异常消息

public:
    // 构造函数，接受异常消息作为参数
    MyException(const char* msg) : message(msg) {}

    // 覆盖基类的 what() 方法，返回异常消息
    virtual const char* what() const throw() {
        return message;
    }
};

// 一个函数，演示如何抛出自定义异常
void myFunction() {
    // 在这个示例中，函数总是抛出自定义异常
    throw MyException("This is a custom exception!");
```

```
}

int main() {
    try {
        myFunction();
    } catch (const MyException& e) {
        std::cout << "Caught MyException: " << e.what() << std::endl;
    }

    return 0;
}
```

在这个示例中，`MyException` 类继承自 `std::exception` 类，并重写了基类的 `what()` 方法，以返回异常消息。在 `myFunction()` 中，我们抛出了一个 `MyException` 类型的异常，并在 `main()` 函数中的 `try-catch` 块中捕获并处理该异常。

通过自定义异常类，你可以根据需要添加其他成员变量、方法或构造函数，以便更好地描述和处理特定类型的异常情况。这种方式可以提高程序的可读性和可维护性，并允许你更精确地控制异常处理。

P3 记事本项目

3.1 项目概述

3.1.1 功能介绍

- 支持文本创建，打开，保存，关闭的功能
- UI样式美化
- 添加打开快捷键，添加保存快捷
- 底部显示行列号及文本字符编码
- Ctrl加鼠标滚轮支持字体放大缩小