练习 15.1：什么是虚成员？

解答：

- **虚成员（Virtual Member）通常指的是虚函数（Virtual Function）**，它允许在基类中定义，并在派生类中进行重写。虚函数支持**动态绑定**，即在运行时根据对象的实际类型调用派生类的实现，而不是基类的实现。通过将基类中的函数声明为 `virtual`，可以确保当基类指针或引用指向派生类对象时，正确调用派生类的版本。

  例如：

```
class Base {
public:
    virtual void display() const { std::cout << "Base" << std::endl; }
    virtual ~Base() = default;
};

class Derived : public Base {
public:
    void display() const override { std::cout << "Derived" << std::endl;
}
};

Base* ptr = new Derived();
ptr->display();  // 输出 "Derived"
```

练习 15.2：`protected` 访问说明符与 `private` 有何区别？

解答：

- `protected`：表示受保护的成员或方法，**只能被派生类和当前类访问**，但不能被类的外部直接访问。派生类可以通过继承关系访问基类中的 `protected` 成员。
- `private`：表示私有成员或方法，**只能在当前类中访问**，即使派生类也无法直接访问基类的 `private` 成员。

总结区别：

- `protected`：类内和派生类可以访问。
- `private`：仅类内可以访问，派生类和外部都不能访问。

练习 15.3：定义你自己的 `Quote` 类和 `print_total` 函数。

解答：

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

```cpp
class Quote {
public:
    Quote() = default;
    Quote(const std::string &book, double sales_price)
        : bookNo(book), price(sales_price) { }

    std::string isbn() const { return bookNo; }

    // 虚函数用于计算价格，可以在派生类中重写
    virtual double net_price(std::size_t n) const {
        return n * price;
    }

    virtual ~Quote() = default;  // 虚析构函数

private:
    std::string bookNo;  // ISBN 编号

protected:
    double price = 0.0;  // 书籍价格
};

// print_total 函数，用于计算销售总额并输出
double print_total(std::ostream &os, const Quote &item, std::size_t n) {
    // 调用 Quote 类或其派生类的 net_price 方法
    double total = item.net_price(n);
    os << "ISBN: " << item.isbn() << " # sold: " << n << " total due: " <<
total << std::endl;
    return total;
}

int main() {
    Quote basic("123-456-789", 50.0);  // 创建一个 Quote 对象
    print_total(std::cout, basic, 5);  // 打印 5 本书的总价
    return 0;
}
```

解释：

1. `Quote` **类**：定义了书籍的基本信息，包括 ISBN 编号和价格。`net_price` 是虚函数，允许派生类重写不同的定价策略。
2. `print_total` **函数**：用于计算销售总额并打印输出，它接受一个 `Quote` 对象的引用（或者派生类对象的引用），并调用 `net_price` 计算总价。
3. **虚析构函数**：确保通过基类指针删除派生类对象时，派生类的析构函数会被正确调用。

练习 15.4：下面哪些声明语句是不正确的？请解释原因。

```cpp
class Base { ... };
```

```
(a) class Derived : public Derived { ... };
(b) class Derived : private Base { ... };
(c) class Derived : public Base { ... };
```

解答:

- **(a) 错误**: `class Derived : public Derived` 是不正确的。类不能继承自身，这会导致递归定义，编译器无法解析，继承关系必须是从一个不同的基类继承。
- **(b) 正确**: `class Derived : private Base` 是正确的，表示 `Derived` 继承 `Base`，并且继承方式是 `private`，这意味着 `Base` 类中的公共和保护成员在 `Derived` 类中变为私有成员。即使继承是私有的，这也是有效的继承语法。
- **(c) 正确**: `class Derived : public Base` 是正确的，表示 `Derived` 从 `Base` 类继承，且继承方式是 `public`。这意味着 `Base` 类中的公共成员和保护成员在 `Derived` 中仍然是公共和保护的。

练习 15.5: 定义你自己的 `Bulk_quote` 类。

解答:

```cpp
class Bulk_quote : public Quote {
public:
    // 构造函数
    Bulk_quote(const std::string& book, double p, std::size_t qty, double disc)
        : Quote(book, p), min_qty(qty), discount(disc) {}

    // 重写 net_price 函数
    double net_price(std::size_t cnt) const override {
        if (cnt >= min_qty) {
            return cnt * (1 - discount) * price;  // 达到最低数量享受折扣
        } else {
            return cnt * price;  // 否则按原价销售
        }
    }

private:
    std::size_t min_qty = 0;  // 享受折扣的最小购买数量
    double discount = 0.0;    // 折扣
};
```

这个类继承自 `Quote`，并重写了 `net_price` 函数来处理批量折扣。根据购买的数量是否达到 `min_qty`，折扣会生效。

练习 15.6: 将 `Quote` 和 `Bulk_quote` 的对象传给 15.2.1 节中的 `print_total` 函数，检查该函数是否正确。

解答:
这里你需要将 `Quote` 和 `Bulk_quote` 对象传递给 `print_total` 函数，并验证它们的输出是否符合预期:

```cpp
int main() {
    Quote basic("123-456-789", 50.0);          // 创建一个 Quote 对象
    Bulk_quote bulk("123-456-789", 50.0, 10, 0.2);  // 创建一个 Bulk_quote 对象

    print_total(std::cout, basic, 5);          // 调用 print_total 函数
    print_total(std::cout, bulk, 15);          // 调用 print_total 函数

    return 0;
}
```

运行时会验证 `print_total` 能够正确处理 `Bulk_quote` 对象和 `Quote` 对象，分别输出不同的总价。

练习 15.7：定义一个类，使其实现一种数量受限的折扣策略，具体策略是：当购买书籍的数量不超过一个给定的限量时享受折扣，如果购买数量一旦超过了限量，则超出的部分将以原价销售。

解答：
可以通过继承 `Quote` 类实现一个数量受限的折扣策略：

```cpp
class Limited_quote : public Quote {
public:
    // 构造函数
    Limited_quote(const std::string& book, double p, std::size_t qty, double
disc)
        : Quote(book, p), max_qty(qty), discount(disc) {}

    // 重写 net_price 函数
    double net_price(std::size_t cnt) const override {
        if (cnt <= max_qty) {
            return cnt * (1 - discount) * price;  // 不超过限量，享受折扣
        } else {
            return max_qty * (1 - discount) * price + (cnt - max_qty) * price;
// 超出部分按原价
        }
    }

private:
    std::size_t max_qty = 0;  // 折扣限量
    double discount = 0.0;    // 折扣率
};
```

这个 `Limited_quote` 类继承自 `Quote`，并且实现了一种限量折扣的策略，超过限量部分按原价销售。

练习 15.8：给出静态类型和动态类型的定义。

- **静态类型**（Static Type）：静态类型是在**编译时**由变量声明确定的类型，也就是程序在编译时所知道的类型。它不会在程序运行时发生变化。

- **动态类型**（Dynamic Type）：动态类型是在**运行时**对象的实际类型。特别是在继承关系中，基类的指针或引用指向派生类对象时，动态类型是派生类的类型。

练习 15.9：在什么情况下表达式的静态类型可能与动态类型不同？请给出三个静态类型与动态类型不同的例子。

静态类型与动态类型不同的情况通常出现在**继承与多态**中。以下是三个例子：

1. **基类指针或引用指向派生类对象**：

   ```
   Quote *quotePtr = new Bulk_quote();  // 静态类型是 Quote*，动态类型是 Bulk_quote*
   ```

   这里，`quotePtr` 的静态类型是 `Quote*`，但它实际指向的是 `Bulk_quote` 对象，因此它的动态类型是 `Bulk_quote*`。

2. **调用虚函数时**：

   ```
   Quote &quoteRef = bulkQuote;  // 静态类型是 Quote&，动态类型是 Bulk_quote&
   quoteRef.net_price(10);  // 调用的是 Bulk_quote 中的 net_price 函数
   ```

   `quoteRef` 的静态类型是 `Quote&`，但它引用的是 `Bulk_quote` 对象，调用虚函数时，实际运行的是 `Bulk_quote` 的 `net_price` 函数。

3. **多态容器**：

   ```
   std::vector<Quote*> quotes;
   quotes.push_back(new Bulk_quote());  // 静态类型是 Quote*，动态类型是 Bulk_quote*
   ```

   这里 `quotes` 容器中存放的是 `Quote*` 类型的指针，但实际存储的对象可能是 `Bulk_quote`，因此其动态类型为 `Bulk_quote*`。

练习 15.10：回忆我们在 8.1 节（第 279 页）进行的讨论，解释第 284 页中将 `ifstream` 传递给 `Sales_data` 的 `read` 函数的程序是如何工作的。

在**第 8.1 节**，我们讨论了通过流对象来读取数据。在 `Sales_data` 类的 `read` 函数中，`ifstream` 被传递作为参数用于从文件流中读取数据。

`read` **函数的工作原理**：

- 当 `ifstream` 被传递给 `Sales_data` 的 `read` 函数时，`read` 函数会使用流对象的输入操作符（如 `>>`），从文件中读取一行数据，并将其解析为 `Sales_data` 对象的成员（如 ISBN、销量、价格等）。

- `read` 函数可以处理 `ifstream` 作为输入，因为 `ifstream` 是从 `istream` 派生的，因此它可以使用 `istream` 的所有操作符和方法。

总之，`read` 函数从文件流中读取并填充 `Sales_data` 对象的成员信息。

练习 15.18

背景说明：

1. **`d1`的类型是`Pub_Derv`**：表示通过**`public`继承**的派生类。
2. **`d2`的类型是`Priv_Derv`**：表示通过**`private`继承**的派生类。
3. **`Base`**：基类指针，尝试指向不同类型的派生类对象。

赋值语句的合法性判断：

语句 1：

```
Base *p = &d1;  // d1的类型是 Pub_Derv
```

解释：

- `Pub_Derv` 是 `Base` 的派生类，且通过 **public** 继承，这意味着 `Base` 类的 `public` 和 `protected` 成员在 `Pub_Derv` 中保持 `public` 和 `protected`。
- 由于是**公有继承**，所以可以将 `Pub_Derv` 类型的对象指针赋值给 `Base` 类型的指针。
- **合法。**

语句 2：

```
p = &d2;  // d2的类型是 Priv_Derv
```

解释：

- `Priv_Derv` 是 `Base` 的派生类，但通过 **private** 继承。
- **private 继承**意味着 `Base` 的所有 `public` 和 `protected` 成员在 `Priv_Derv` 中都变成了 `private`，因此从外部代码的角度看，`Priv_Derv` 与 `Base` 没有任何继承关系。
- 由于 `Priv_Derv` 与 `Base` 之间没有公开的继承关系，所以不能将 `Priv_Derv` 的对象指针赋值给 `Base` 类型的指针。
- **不合法。**

第二部分：

1. **`d3`的类型是`Prot_Derv`**：表示通过**`protected`继承**的派生类。
2. **`dd1`的类型是`Derived_from_Public`**：表示通过`public`继承的派生类。
3. **`dd2`的类型是`Derived_from_Private`**：表示通过`private`继承的派生类。
4. **`dd3`的类型是`Derived_from_Protected`**：表示通过`protected`继承的派生类。

合法性判断：

语句 1：

```
p = &d3;  // d3的类型是 Prot_Derv
```

解释：

- `Prot_Derv` 是通过 **protected** 继承的派生类，继承的 `Base` 成员在 `Prot_Derv` 中变成 `protected`。由于外部代码无法访问 `protected` 成员，所以 `Prot_Derv` 不能隐式转换为 `Base`。
- **不合法。**

语句 2：

```
p = &dd1;  // dd1的类型是 Derived_from_Public
```

解释：

- `Derived_from_Public` 是 `Base` 的派生类，通过 **public** 继承，因此可以将 `Derived_from_Public` 的指针赋值给 `Base` 类型的指针。
- **合法。**

语句 3：

```
p = &dd2;  // dd2的类型是 Derived_from_Private
```

解释：

- `Derived_from_Private` 是通过 **private** 继承 `Base` 的派生类，无法将 `Derived_from_Private` 类型的指针赋值给 `Base` 的指针。
- **不合法。**

语句 4：

```
p = &dd3;  // dd3的类型是 Derived_from_Protected
```

解释：

- `Derived_from_Protected` 通过 **protected** 继承 `Base`，外部代码无法访问 `protected` 成员，因此不能进行这种类型转换。
- **不合法。**

总结：

- **合法**的赋值：`Base *p = &d1;`, `p = &dd1;`
- **不合法**的赋值：`p = &d2;`, `p = &d3;`, `p = &dd2;`, `p = &dd3;`

这些赋值操作的合法性取决于派生类是如何继承基类的。如果继承方式是 `public`，则可以进行类型转换；而对于 `protected` 和 `private` 继承，则不允许在外部进行基类和派生类之间的转换。

练习 15.19：

函数定义如下：

```
void memfcn(Base &b) {
    b = *this;
}
```

判断：

- 这个成员函数将当前对象赋值给参数 `b`。该函数的作用是将派生类对象赋值给 `Base` 类型的引用。
- 由于该函数假设 `this` 是派生类对象，而参数 `b` 是基类 `Base` 的引用，能否进行这样的赋值取决于继承方式。

分析：

- 如果派生类是通过 `public` **继承** 基类的，那么派生类的对象可以赋值给基类的引用，因此函数是合法的。
- 如果派生类是通过 `protected` **或** `private` **继承** 基类的，那么基类的引用 `b` 无法被派生类对象赋值，因为这种继承限制了从派生类向基类的转换，所以函数是非法的。

练习 15.20：

编写代码验证你的答案是否正确。

示例代码：

我们可以通过定义几个继承类来验证 15.19 的结论。

```
#include <iostream>

class Base {
public:
    Base& operator=(const Base& rhs) {
        std::cout << "Base assignment operator\n";
        return *this;
    }
};

// 公有继承
```

```
class Pub_Derv : public Base {
public:
    void memfcn(Base &b) {
        b = *this;  // 合法，因为 Pub_Derv 是公有继承，允许向基类赋值
    }
};

// 受保护继承
class Prot_Derv : protected Base {
public:
    void memfcn(Base &b) {
        // b = *this;  // 不合法，不能将受保护继承的派生类对象赋值给 Base 类型
    }
};

// 私有继承
class Priv_Derv : private Base {
public:
    void memfcn(Base &b) {
        // b = *this;  // 不合法，不能将私有继承的派生类对象赋值给 Base 类型
    }
};

int main() {
    Base base;
    Pub_Derv pubD;
    Prot_Derv protD;
    Priv_Derv privD;

    pubD.memfcn(base);   // 应该合法
    // protD.memfcn(base); // 非法，不允许编译
    // privD.memfcn(base); // 非法，不允许编译

    return 0;
}
```

练习 15.21：

选择一种通用抽象概念并组织一个继承体系。

这里我们选择 **图形基元**（即形状），并将其组织成一个继承体系。

继承体系：

我们可以定义一个 `Shape`（形状）基类，并派生出不同的几何图形类型，例如矩形、圆形、球体、圆锥等。

代码示例：

```cpp
#include <iostream>
#include <cmath>

// 基类：图形基元
class Shape {
public:
    virtual double area() const = 0;    // 纯虚函数：计算面积
    virtual double volume() const = 0;  // 纯虚函数：计算体积（如果适用）
    virtual ~Shape() = default;         // 虚析构函数
};

// 派生类：矩形
class Rectangle : public Shape {
private:
    double width;
    double height;
public:
    Rectangle(double w, double h) : width(w), height(h) {}

    double area() const override {
        return width * height;
    }

    double volume() const override {
        return 0.0;  // 矩形没有体积
    }
};

// 派生类：圆
class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}

    double area() const override {
        return M_PI * radius * radius;
    }

    double volume() const override {
        return 0.0;  // 圆形没有体积
    }
};

// 派生类：球体
class Sphere : public Shape {
private:
    double radius;
public:
    Sphere(double r) : radius(r) {}

    double area() const override {
```

```
        return 4 * M_PI * radius * radius;
    }

    double volume() const override {
        return (4.0/3) * M_PI * radius * radius * radius;
    }
};

int main() {
    Rectangle rect(10, 20);
    Circle circ(15);
    Sphere sphere(10);

    std::cout << "Rectangle area: " << rect.area() << std::endl;
    std::cout << "Circle area: " << circ.area() << std::endl;
    std::cout << "Sphere area: " << sphere.area() << std::endl;
    std::cout << "Sphere volume: " << sphere.volume() << std::endl;

    return 0;
}
```

练习 15.22：

为你选择的类添加适当的虚函数及公共成员和受保护成员。

在上述代码中，`Shape` 类已经定义了两个纯虚函数 `area()` 和 `volume()`，这使得所有派生类必须实现它们。同时，还定义了虚析构函数以确保正确的资源管理。

你可以进一步扩展类结构，添加受保护的成员变量或者更多的虚函数来实现更复杂的行为。