第六章 类

目录

学习目标

1. 理解类的概念和基本设计原则；
2. 理解构造函数和析构函数的作用与执行过程；
3. 掌握常见运算符重载技术和静态成员的使用方法；
4. 能够运用面向对象程序设计方法设计和实现简单类；

6.1 类的定义

类

类是对一个事物的属性和操作的描述。

- 用户自定义类型；
- 基本思想是抽象（abstract）和封装（encapsulation）；
- 面向对象程序设计（object-oriented programming，OOP）的基础。

6.1 类的定义

类

类是对一个事物的属性和操作的描述。

- 用户自定义类型；
- 基本思想是抽象（abstract）和封装（encapsulation）；
- 面向对象程序设计（object-oriented programming，OOP）的基础。

类的要素

- 抽象：数据（属性）抽象和函数（操作）抽象；
- 封装：将数据和操作结合在一起。

6.1.1 定义一个类

设计分数类

设计一个分数类：基本属性有分子和分母，操作包括约分、计算分数值等操作。

6.1.1 定义一个类

设计分数类

设计一个分数类：基本属性有分子和分母，操作包括约分、计算分数值等操作。

分数类

```
class Fraction {
    //数据成员，访问控制属性默认是私有
    int m_numerator = 0; // 分子默认为0;  C++11
    int m_denominator = 1; //分母默认为1;
public://公有成员函数
    int numerator() const { return m_numerator; }
    int denominator() const { return
     m_denominator; }
    double value() const; //计算分数值
    void reduce(); //约分
private:
    int gcd(int x, int y); //求分子分母最大公约数
};
//辅助函数
void makeCommon(Fraction &a, Fraction &b);
ostream &print(ostream &out, const Fraction &f);
```

6.1.1 定义一个类

设计分数类

设计一个分数类：基本属性有分子和分母，操作包括约分、计算分数值等操作。

分数类

```
class Fraction {
    //数据成员，访问控制属性默认是私有
    int m_numerator = 0; // 分子默认为0;  C++11
    int m_denominator = 1; //分母默认为1;
public://公有成员函数
    int numerator() const { return m_numerator; }
    int denominator() const { return
     m_denominator; }
    double value() const; //计算分数值
    void reduce(); //约分
private:
    int gcd(int x, int y); //求分子分母最大公约数
};
//辅助函数
void makeCommon(Fraction &a, Fraction &b);
ostream &print(ostream &out, const Fraction &f);
```

类的语法

- 类名紧随 class 关键字；
- 数据成员和成员函数放到一对花括号里面；
- 分号结尾。

6.1.1 定义一个类

设计分数类

设计一个分数类：基本属性有分子和分母，操作包括约分、计算分数值等操作。

分数类

```
class Fraction {
    //数据成员，访问控制属性默认是私有
    int m_numerator = 0; // 分子默认为0;  C++11
    int m_denominator = 1; //分母默认为1;
public://公有成员函数
    int numerator() const { return m_numerator; }
    int denominator() const { return
     m_denominator; }
    double value() const; //计算分数值
    void reduce(); //约分
private:
    int gcd(int x, int y); //求分子分母最大公约数
};
//辅助函数
void makeCommon(Fraction &a, Fraction &b);
ostream &print(ostream &out, const Fraction &f);
```

类的语法

- 类名紧随 class 关键字；
- 数据成员和成员函数放到一对花括号里面；
- 分号结尾。

建议：成员命名

- 一个类的数据成员名通常以 m_ 开头；
- 成员函数名尽量使用动词短语，如 reverse。

6.1.2 定义和使用成员函数

定义成员函数

- 成员函数的声明必须放到类的内部
- 成员函数的定义
  - 类的内部：建议为内联函数；
  - 类的外部：可利用 inline 指明其为内联函数。
- 需要使用作用域符标明成员函数所属的类；

6.1.2 定义和使用成员函数

定义成员函数

- 成员函数的声明必须放到类的内部
- 成员函数的定义
  - 类的内部：建议为内联函数；
  - 类的外部：可利用 inline 指明其为内联函数。
- 需要使用作用域符标明成员函数所属的类；

类外定义成员函数

```
inline double Fraction::value() const{
    return static_cast<double>(m_numerator) / m_denominator;
}
void Fraction::reduce(){
    int n=gcd(m_numerator, m_denominator); //获取分子分母的最大公约数
    m_denominator /= n;
    m_numerator /= n;
}
```

6.1.2 定义和使用成员函数 — this 指针

使用 value 成员函数

```
Fraction a;
cout << a.value() <<endl; //打印默认值0
```

6.1.2 定义和使用成员函数 — this 指针

使用 value 成员函数

```
Fraction a;
cout << a.value() <<endl; //打印默认值0
```

a 的数据成员如何传到 value 成员函数中？

6.1.2 定义和使用成员函数 — this 指针

使用 value 成员函数

```
Fraction a;
cout << a.value() <<endl; //打印默认值0
```

a 的数据成员如何传到 value 成员函数中？

编译器转化 value 成员函数声明

```
double value(Fraction *const this) const;
```

value 成员函数等价调用

```
Fraction a;
cout << a.value(&a) <<endl;
```

6.1.2 定义和使用成员函数 — this 指针

使用 value 成员函数

```
Fraction a;
cout << a.value() <<endl; //打印默认值0
```

a 的数据成员如何传到 value 成员函数中？

编译器转化 value 成员函数声明

```
double value(Fraction *const this) const;
```

value 成员函数等价调用

```
Fraction a;
cout << a.value(&a) <<endl;
```

this 指针

- 指向调用成员函数对象的指针常量；

```
inline double Fraction::value(Fraction *const this) const{
    return static_cast<double>(this->m_numerator) / this->m_denominator;
}
```

6.1.2 定义和使用成员函数 — const 成员函数

常量成员函数（const member function）

- 函数参数列表后面（圆括号后面）引入 const 关键字；
- 函数体内部写操作是非法的。

6.1.2 定义和使用成员函数 — const 成员函数

常量成员函数（const member function）

- 函数参数列表后面（圆括号后面）引入 const 关键字；
- 函数体内部写操作是非法的。

示例

```
double value() const;
```

6.1.2 定义和使用成员函数 — const 成员函数

常量成员函数（const member function）

- 函数参数列表后面（圆括号后面）引入 const 关键字；
- 函数体内部写操作是非法的。

示例

```
double value() const;
```

等价于

```
double value(const Fraction *const this);
```

this 指针是一个指向 const 对象的 const 指针。

6.1.3 定义辅助函数

辅助函数

一个类需要一些普通函数来辅助完成某些操作，不是类的成员函数，从概念上属于类的接口。

- 通常将它们的声明和类的定义放到同一个头文件内；
- 定义和类成员函数的定义放到同一个源文件内。

6.1.3 定义辅助函数

辅助函数

一个类需要一些普通函数来辅助完成某些操作，不是类的成员函数，从概念上属于类的接口。

- 通常将它们的声明和类的定义放到同一个头文件内；
- 定义和类成员函数的定义放到同一个源文件内。

示例

```
ostream &print(ostream &out, const Fraction &f) {
    out << f.numerator() << "/" << f.denominator();
    return out;
}
Fraction a;
print(cout,a);//打印0/1
```

6.1.4 访问控制

封装的意义

- 类的开发者：有利于程序的模块化设计，提高代码的重用性；
- 类的使用者：隐藏类的实现细节，使使用者不能在类的外面操控数据成员，从而形成一种保护机制。

6.1.4 访问控制

封装的意义

- 类的开发者：有利于程序的模块化设计，提高代码的重用性；
- 类的使用者：隐藏类的实现细节，使使用者不能在类的外面操控数据成员，从而形成一种保护机制。

访问控制

- 如何实现访问控制？

6.1.4 访问控制

封装的意义

- 类的开发者：有利于程序的模块化设计，提高代码的重用性；
- 类的使用者：隐藏类的实现细节，使使用者不能在类的外面操控数据成员，从而形成一种保护机制。

访问控制

- 如何实现访问控制？
- 访问限定符（access specifiers）：控制成员对外的可见性。

6.1.4 访问控制

封装的意义

- 类的开发者：有利于程序的模块化设计，提高代码的重用性；
- 类的使用者：隐藏类的实现细节，使使用者不能在类的外面操控数据成员，从而形成一种保护机制。

访问控制

- 如何实现访问控制？
- 访问限定符（access specifiers）：控制成员对外的可见性。

访问限定符

- public ：成员对外是公开的，可以在程序的任何地方访问；
- private ：成员不对外公开，只在类的内部使用（成员函数内）。

6.1.4 访问控制

下面函数是否正确？为什么？

```
ostream &print(ostream &out, const Fraction &f) {
    out << f.m_numerator << "/" << f.m_denominator;
    return out;
}
```

6.1.4 访问控制

下面函数是否正确？为什么？

```
ostream &print(ostream &out, const Fraction &f) {
    out << f.m_numerator << "/" << f.m_denominator;
    return out;
}
```

利用 class 定义的类，如果没有显式指明成员的访问属性，它们的访问属性默认为 private，在类的外部直接访问它们是非法的。

6.1.4 访问控制

下面函数是否正确？为什么？

```
ostream &print(ostream &out, const Fraction &f) {
    out << f.m_numerator << "/" << f.m_denominator;
    return out;
}
```

利用 class 定义的类，如果没有显式指明成员的访问属性，它们的访问属性默认为 private，在类的外部直接访问它们是非法的。

class 和 struct

- 都可以定义一个类。
- 使用 class 定义的类中成员的访问属性默认为 private；
- 使用 struct 定义的类中成员的访问属性默认为 public；

6.1.4 访问控制

下面函数是否正确？为什么？

```
ostream &print(ostream &out, const Fraction &f) {
    out << f.m_numerator << "/" << f.m_denominator;
    return out;
}
```

利用 class 定义的类，如果没有显式指明成员的访问属性，它们的访问属性默认为 private，在类的外部直接访问它们是非法的。

class 和 struct

- 都可以定义一个类。
- 使用 class 定义的类中成员的访问属性默认为 private；
- 使用 struct 定义的类中成员的访问属性默认为 public；

私有成员函数

```
private:
    int gcd(int x, int y);
```

私有成员函数，对外是隐藏的，只能在类内部的成员函数中使用。

6.1.5 类的定义 — 友元函数

友元函数

类的辅助函数，声明为类的友元之后，它们就可以访问类的非公有成员。

- 辅助函数的声明放到类的内部；
- 使用关键字 friend 进行说明。

6.1.5 类的定义 — 友元函数

友元函数的声明

```
class Fraction {
    //类的辅助函数声明为友元;
    friend ostream& print(ostream &out, const Fraction &f);
    friend Fraction operator/(int left, const Fraction &right);
    //其它成员与之前一致
};
//Fraction类辅助函数声明
void makeCommon(Fraction &a, Fraction &b);
ostream& print(ostream &out, const Fraction &f);
```

6.1.5 类的定义 — 友元函数

友元函数的声明

```
class Fraction {
    //类的辅助函数声明为友元;
    friend ostream& print(ostream &out, const Fraction &f);
    friend Fraction operator/(int left, const Fraction &right);
    //其它成员与之前一致
};
//Fraction类辅助函数声明
void makeCommon(Fraction &a, Fraction &b);
ostream& print(ostream &out, const Fraction &f);
```

友元函数的定义

```
ostream& print(ostream &out, const Fraction &f) {
    out << f.m_numerator << "/" << f.m_denominator;//正确: 可以访问私有成员
    return out;
}
```

6.1.5 类的定义 — 友元函数

友元函数的声明

```
class Fraction {
    //类的辅助函数声明为友元;
    friend ostream& print(ostream &out, const Fraction &f);
    friend Fraction operator/(int left, const Fraction &right);
    //其它成员与之前一致
};
//Fraction类辅助函数声明
void makeCommon(Fraction &a, Fraction &b);
ostream& print(ostream &out, const Fraction &f);
```

友元函数的定义

```
ostream& print(ostream &out, const Fraction &f) {
    out << f.m_numerator << "/" << f.m_denominator;//正确: 可以访问私有成员
    return out;
}
```

友元函数的优点

通过友元声明，免去了成员函数的调用。

6.1.5 类的定义 — 友元类

友元类

将其它类或其它类的成员函数声明为该类的友元。

6.1.5 类的定义 — 友元类

友元类

将其它类或其它类的成员函数声明为该类的友元。

问题

下面定义两个类：Point 类和 Circle 类，并将 Circle 类声明为 Point 类的友元，使 Circle 类的成员函数可以直接访问 Point 类的私有成员。

6.1.5 友元 — 友元类

友元类的定义

```
class Circle; //前向声明
class Point {
    friend class Circle;
private:
    double m_x = 0, m_y = 0;//x和y坐标
};
class Circle {
    Point m_center; //圆心
    double m_radius = 1.0;//半径
public:
    void moveXTo(double val) {//将圆心沿x轴移动到指定位置
       m_center.m_x = val;//直接访问Point私有成员m_x
    }
};
```

6.1.5 友元 — 友元类

前向声明（forward declaration）

将 Circle 类声明为 Point 类的友元之前，需要先声明 Circle 类，否则会出现语法错误。

6.1.5 友元 — 友元类

前向声明（forward declaration）

将 Circle 类声明为 Point 类的友元之前，需要先声明 Circle 类，否则会出现语法错误。

不完全类型（incomplete type）

- 只告诉编译器 Circle 为类类型，但 Circle 类的成员此时还未知，如前向声明；
- 不完全类开型可以定义指向该类型的指针或引用。

6.1.5 友元 — 友元类

前向声明（forward declaration）

将 Circle 类声明为 Point 类的友元之前，需要先声明 Circle 类，否则会出现语法错误。

不完全类型（incomplete type）

- 只告诉编译器 Circle 为类类型，但 Circle 类的成员此时还未知，如前向声明；
- 不完全类开型可以定义指向该类型的指针或引用。

提示

友元单向且不可传递。

6.2.1 构造函数

构造函数

- 在对象创建时为数据成员执行初始化操作；
- 只要创建类类型对象，就会执行类的构造函数。

6.2.1 构造函数

构造函数

- 在对象创建时为数据成员执行初始化操作；
- 只要创建类类型对象，就会执行类的构造函数。

构造函数的语法

- 函数名必须和类名一致；
- 无返回值类型说明；
- 不能声明为 const 成员函数。

6.2.1 构造函数

示例

为 Fraction 类显式定义一个带有两个参数的构造函数：

```
class Fraction {//注意, Fraction 类其它成员在此没有列出
public:
   Fraction(int above, int below) :
      m_numerator(above), m_denominator(below) {}
};
```

6.2.1 构造函数

示例

为 Fraction 类显式定义一个带有两个参数的构造函数：

```
class Fraction {//注意, Fraction 类其它成员在此没有列出
public:
   Fraction(int above, int below) :
      m_numerator(above), m_denominator(below) {}
};
```

为什么构造函数不能声明为常量成员函数?

6.2.1 构造函数

示例

为 Fraction 类显式定义一个带有两个参数的构造函数：

```
class Fraction {//注意, Fraction 类其它成员在此没有列出
public:
   Fraction(int above, int below) :
       m_numerator(above), m_denominator(below) {}
};
```

为什么构造函数不能声明为常量成员函数?

只有当构造函数执行完毕之后，对象才创建完毕，在创建的过程中要为数据成员分配存储空间并执行初始化操作。

6.2.1 默认构造函数

默认构造函数

- C++ 默认提供；
- 没有参数或者所有参数都具有默认值；
- 如果类内数据成员存在初始值，则用此值初始化数据成员；否则采用默认方式初始化。；
- 如果显式地定义了构造函数，那么编译器将不会合成默认的构造函数。

6.2.1 默认构造函数 — 定义构造函数

下面程序是否正确？

```
class Fraction {
public:
   Fraction(int above, int below) :
       m_numerator(above), m_denominator(below) {}
};
Fraction a;
```

6.2.1 默认构造函数 — 定义构造函数

下面程序是否正确？

```
class Fraction {
public:
   Fraction(int above, int below) :
       m_numerator(above), m_denominator(below) {}
};
Fraction a;
```

- 如果显式地定义了构造函数，那么编译器将不会合成默认的构造函数。

6.2.1 默认构造函数 — 定义构造函数

使用默认构造函数

C++11 允许在显式定义构造函数的情况下，使用默认的构造函数

使用默认构造函数

C++11 允许在显式定义构造函数的情况下，使用默认的构造函数

```
class Fraction {//注意, Fraction 类其它成员在此没有列出
public:
   Fraction() = default ;
   Fraction(int above, int below) :
      m_numerator(above), m_denominator(below) {}
};
Fraction a; //正确: 默认的构造函数。
```

6.2.1 默认构造函数 — 定义构造函数

使用默认构造函数

C++11 允许在显式定义构造函数的情况下，使用默认的构造函数

```
class Fraction {//注意, Fraction 类其它成员在此没有列出
public:
   Fraction() = default ;
   Fraction(int above, int below) :
      m_numerator(above), m_denominator(below) {}
};
Fraction a; //正确: 默认的构造函数。
```

警告：切勿乱用圆括号

```
Fraction a();
```

上面代码为一个函数声明，函数名为 a，返回值类型为 Fraction。

6.2.1 默认构造函数 — 初始值列表

下面代码有什么问题？

```
Fraction(int above, int below){
    m_numerator = above;//赋值语句
    m_denominator = below;//赋值语句
}
```

6.2.1 默认构造函数 — 初始值列表

下面代码有什么问题？

```
Fraction(int above, int below){
    m_numerator = above;//赋值语句
    m_denominator = below;//赋值语句
}
```

对象成员先构造后赋值，效率低。

6.2.1 默认构造函数 — 初始值列表

初始值列表的语法

- 在构造函数参数列表后面和左花括号之间；
- 以冒号开始；
- 利用形参值直接初始化数据成员；
- 数据成员之间用逗号隔开。

6.2.1 默认构造函数 — 初始值列表

初始值列表的语法

- 在构造函数参数列表后面和左花括号之间；
- 以冒号开始；
- 利用形参值直接初始化数据成员；
- 数据成员之间用逗号隔开。

示例

```
Fraction(int above, int below) :
   m_numerator(above), m_denominator(below) {}
```

6.2.1 默认构造函数 — 初始值列表

初始值列表的语法

- 在构造函数参数列表后面和左花括号之间；
- 以冒号开始；
- 利用形参值直接初始化数据成员；
- 数据成员之间用逗号隔开。

示例

```
Fraction(int above, int below) :
  m_numerator(above), m_denominator(below) {}
```

提示

数据成员的构造顺序取决于数据成员在类内定义的顺序。

6.2.1 默认构造函数 — 初始值列表

为什么下面必须使用初值值列表进行初始化？

```
class Foo {
    int &m_ref;
    const int m_con;
public:
    Foo(int &i) :m_ref(i), m_con(i)/*必须在此初始化*/ {}
};
```

6.2.1 默认构造函数 — 初始值列表

为什么下面必须使用初值值列表进行初始化？

```
class Foo {
    int &m_ref;
    const int m_con;
public:
    Foo(int &i) :m_ref(i), m_con(i)/*必须在此初始化*/ {}
};
```

引用类型成员或者有 const 修饰符的成员，必须要利用初始值列表进行初始化。

6.2.1 默认构造函数 — 简化构造函数

简化构造函数

将编译器合成的默认构造函数和程序员自己定义的构造函数合并为一个带默认值的构造函数。

6.2.1 默认构造函数 — 简化构造函数

简化构造函数

将编译器合成的默认构造函数和程序员自己定义的构造函数合并为一个带默认值的构造函数。

示例

```
 class Fraction {
public:
    Fraction(int above = 0, int below = 1):
       m_numerator(above), m_denominator(below) {}
};
```

6.2.1 默认构造函数 — 委托构造函数

委托构造函数 (delegating constructor)

委托构造函数将使用其它的构造函数来完成数据成员的初始化。

6.2.1 默认构造函数 — 委托构造函数

示例

```
class Employee {
    int m_id;
    string m_name;
public:
    Employee(int id = 0, const string &name = "") :
        m_id(id), m_name(name) {}
    Employee(const string &name) :Employee(0, name) {}
};
Employee member("Kevin");
```

6.2.1 默认构造函数 — 委托构造函数

示例

```
class Employee {
    int m_id;
    string m_name;
public:
    Employee(int id = 0, const string &name = "") :
        m_id(id), m_name(name) {}
    Employee(const string &name) :Employee(0, name) {}
};
Employee member("Kevin");
```

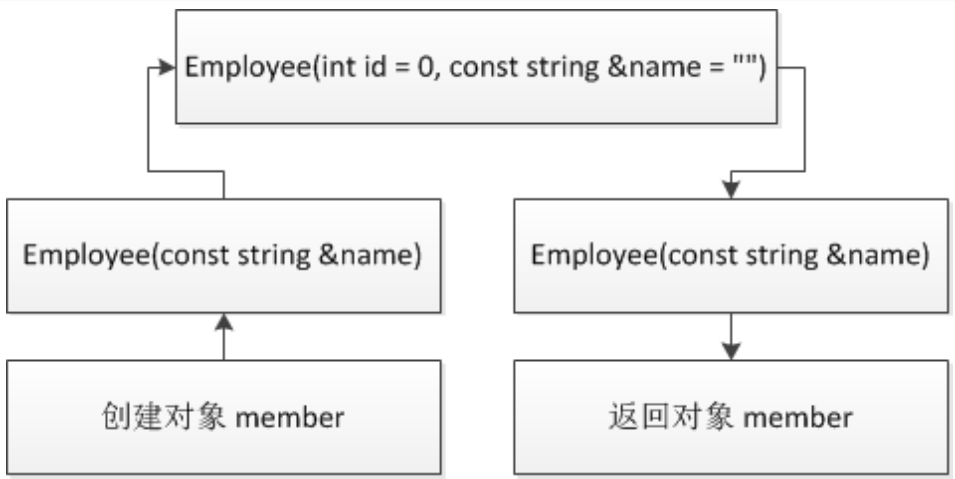

图: 创建 Employee 对象 member 的过程

6.2.2 复制构造函数

示例

复制一个已存在对象的内容来创建对象

```
Fraction a(1, 5); //直接初始化
Fraction b(a);//直接初始化, b 为对象 a 的拷贝
```

6.2.2 复制构造函数

示例

复制一个已存在对象的内容来创建对象

```
Fraction a(1, 5); //直接初始化
Fraction b(a);//直接初始化, b 为对象 a 的拷贝
```

复制构造函数（copy constructor）

- 功能是将给定对象的数据成员依次复制给正在创建的对象；
- 复制构造函数的形参必须是引用类型，即给定实参的引用；
- 编译器会自动合成。

6.2.2 复制构造函数

示例

复制一个已存在对象的内容来创建对象

```
Fraction a(1, 5); //直接初始化
Fraction b(a);//直接初始化, b 为对象 a 的拷贝
```

复制构造函数（copy constructor）

- 功能是将给定对象的数据成员依次复制给正在创建的对象；
- 复制构造函数的形参必须是引用类型，即给定实参的引用；
- 编译器会自动合成。

定义形式

```
class Fraction {
public:
    Fraction(const Fraction &rhs) : m_numerator(rhs.m_numerator),
        m_denominator(rhs.m_denominator) { }
};
```

6.2.2 复制构造函数

问题：下面代码是否正确？

```
class Fraction {
public:
    Fraction(Fraction rhs):
        m_numerator(rhs.m_numerator),
         m_denominator(rhs.m_denominator) {}
};
int main(){
    Fraction a;
    Fraction b(a);
}
```

6.2.2 复制构造函数

问题：下面代码是否正确？

```
class Fraction {
public:
    Fraction(Fraction rhs):
        m_numerator(rhs.m_numerator),
         m_denominator(rhs.m_denominator) {}
};
int main(){
    Fraction a;
    Fraction b(a);
}
```

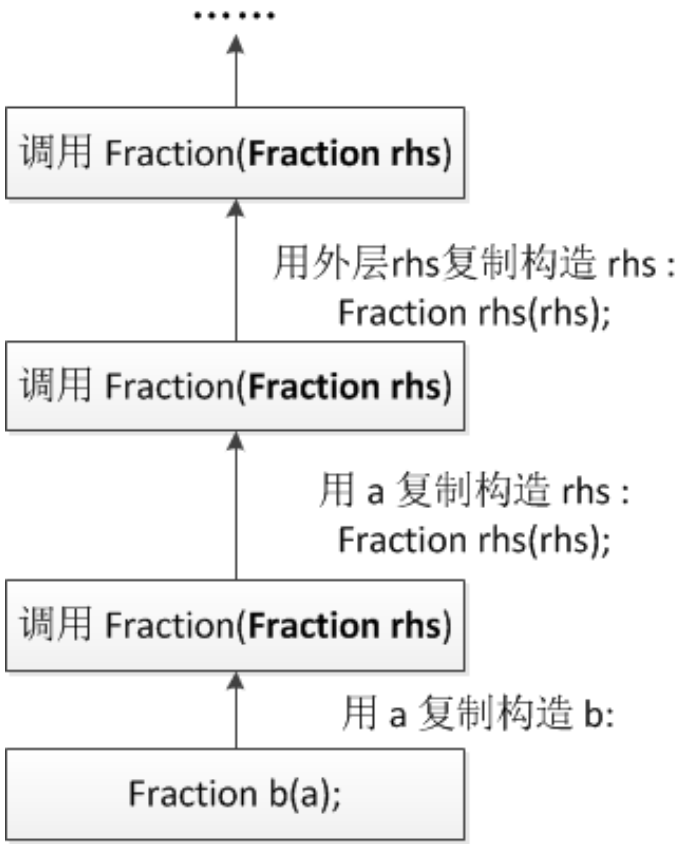

图: 无穷调用

6.2.2 复制构造函数 — 复制初始化

复制初始化例子

```
Fraction c = Fraction(3, 2); //复制初始化
Fraction d = 7;//复制初始化
```

6.2.2 复制构造函数 — 复制初始化

复制初始化例子

```
Fraction c = Fraction(3, 2); //复制初始化
Fraction d = 7;//复制初始化
```

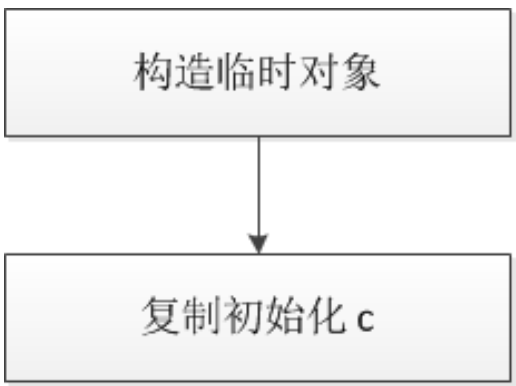

图: Fraction c = Fraction(3, 2);

6.2.2 复制构造函数 — 复制初始化

复制初始化例子

```
Fraction c = Fraction(3, 2); //复制初始化
Fraction d = 7;//复制初始化
```

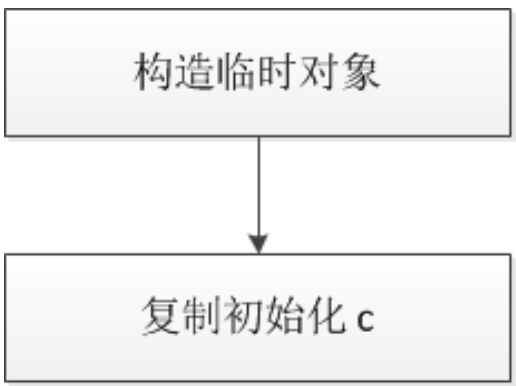

图: Fraction c = Fraction(3, 2);

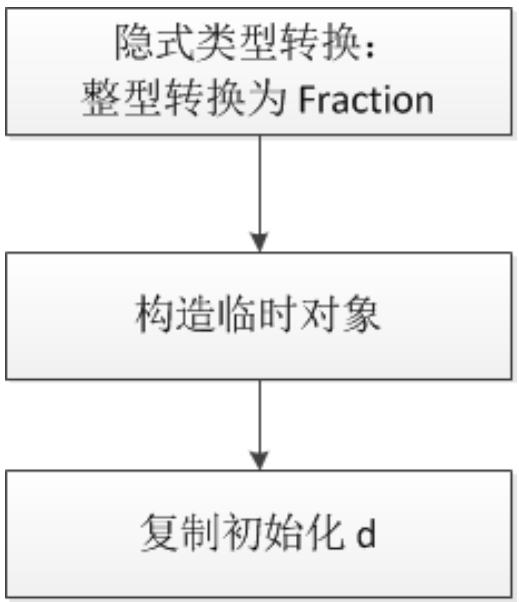

图: Fraction d = 7;

6.2.2 复制构造函数 — 复制初始化

复制初始化

- 将 = 右侧的对象复制给待创建对象；
- 类型不一致，编译器尝试通过构造函数进行类型转换；
- 如果没有可用构造函数，将产生类型无法转换的错误。

6.2.2 复制构造函数 — 抑制隐式类型转换

隐式类型转换的缺点

- 可能产生临时对象，带来程序性能的下降；
- 给用户带来意外，引起理解上的困难。。

抑制隐式类型转换

explicit 关键字来阻止隐式类型转换，例子：

```
class Fraction {
public:
    explicit Fraction(int above = 0, int below = 1) :
        m_numerator(above), m_denominator(below) { }
};
```

6.2.2 复制构造函数 — 抑制隐式类型转换

隐式类型转换的缺点

- 可能产生临时对象，带来程序性能的下降；
- 给用户带来意外，引起理解上的困难。。

抑制隐式类型转换

explicit 关键字来阻止隐式类型转换，例子：

```
class Fraction {
public:
    explicit Fraction(int above = 0, int below = 1) :
        m_numerator(above), m_denominator(below) { }
};


    Fraction d2 = 7; //错误: 无法将 int 类型转换为 Fraction 类型
```

6.2.2 复制构造函数 — 抑制隐式类型转换

隐式类型转换的缺点

- 可能产生临时对象，带来程序性能的下降；
- 给用户带来意外，引起理解上的困难。。

抑制隐式类型转换

explicit 关键字来阻止隐式类型转换，例子：

```
class Fraction {
public:
    explicit Fraction(int above = 0, int below = 1) :
        m_numerator(above), m_denominator(below) { }
};


    Fraction d2 = 7; //错误: 无法将 int 类型转换为 Fraction 类型

    Fraction d2 = static_cast<Fraction>(7);//正确, 强制类型转换
```

6.2.2 复制构造函数 — 阻止复制

复制构造函数的使用

- 将一个对象作为实参传递给一个非引用类型形参；
- 函数返回值类型为非引用类型对象；

6.2.2 复制构造函数 — 阻止复制

示例

divide 函数：用来实现两个分数的相除运算，返回结果为一个分数类型。

```
Fraction divide(Fraction dividend, Fraction divisor) {
    Fraction result(divident.numerator()*divisor.denominator(),
        divident.denominator()*divisor.numerator());
    return result;
}

Fraction e = divide(b, c);
```

图: divide 函数

6.2.2 复制构造函数 — 阻止复制

阻止复制

```
class Employee {
public:
    Employee(const Employee &) = delete;//阻止复制
    Employee &operator=(const Employee &) = delete; //阻止赋值
};

Employee e1; //调用默认构造函数
Employee e2(e1); //错误: 复制构造函数是删除的, 不能调用
```

6.2.2 复制构造函数 — 复制优化

以下代码会输出什么？

```
class Fraction {
public:
    Fraction(const Fraction &rhs) : m_numerator(rhs.m_numerator),
        m_denominator(rhs.m_denominator) {
        cout; << ; "Copy constructor called"; << ; endl;
    }
};

Fraction c2 = Fraction(3, 2);
```

6.2.2 复制构造函数 — 复制优化

以下代码会输出什么？

```
class Fraction {
public:
    Fraction(const Fraction &rhs) : m_numerator(rhs.m_numerator),
        m_denominator(rhs.m_denominator) {
        cout; << ; "Copy constructor called"; << ; endl;
    }
};

Fraction c2 = Fraction(3, 2);
```

复制优化

C++ 编译器使用一种名为拷贝去除（copy ellision）的优化技术：创建临时对象和调用拷贝构造函数这两个过程完全可以用直接初始化的方式来完成。

6.2.3 析构函数

析构函数（destructor）

- 一个对象生命期结束时，编译器会自动调用；
- 销毁对象的所有成员；
- 名字由波浪号紧接类名构成，不能有返回值，也不能包含形参。

6.2.3 析构函数

析构函数（destructor）

- 一个对象生命期结束时，编译器会自动调用；
- 销毁对象的所有成员；
- 名字由波浪号紧接类名构成，不能有返回值，也不能包含形参。

示例

```
 class Fraction {
public:
    ~Fraction() { } //析构函数
};
```

6.2.3 析构函数

析构函数（destructor）

- 一个对象生命期结束时，编译器会自动调用；
- 销毁对象的所有成员；
- 名字由波浪号紧接类名构成，不能有返回值，也不能包含形参。

示例

```
 class Fraction {
public:
    ~Fraction() { } //析构函数
};
```

析构函数的行为

- 先执行函数体，然后按照初始化的顺序逆序销毁每个成员；
- 类类型的成员执行自己的析构函数，内置类型的成员什么也不需要做；
- 释放对象成员在生命期内被分配的所有资源。

6.3 运算符重载

打印一个对象 obj

为 Fraction 类定义了一个辅助打印函数 print

```
Fraction obj ;
print(cout, obj);
```

6.3 运算符重载

打印一个对象 obj

为 Fraction 类定义了一个辅助打印函数 print

```
Fraction obj ;
print(cout, obj);
```

打印一个对象 obj

重载了 << 运算符

```
Fraction obj ;
cout << obj ;
```

6.3 运算符重载

打印一个对象 obj

为 Fraction 类定义了一个辅助打印函数 print

```
Fraction obj ;
print(cout, obj);
```

打印一个对象 obj

重载了 << 运算符

```
Fraction obj ;
cout << obj ;
```

运算符重载

- 将运算符作用于类类型对象，重新赋予新的含义；
- 增强代码的可读性。

6.3.1 基本概念

运算符基本概念

- 由关键字 operator 和需要重载的运算符共同组成；
- 重载的运算符的返回值类型以及参数列表由运算符的性质来决定：
  - 一元运算符有一个参数；
  - 二元运算符有两个参数，左侧运算对象传递给第一个参数，右侧运算对象传递给第二个参数

6.3.1 基本概念

为 Fraction 类重载算术运算符 /，实现两个分数的除法运算

```
Fraction operator/(const Fraction &left, const Fraction &right) {
    Fraction result(left.numerator()*right.denominator(),
        left.denominator()*right.numerator());
    return result;
}
```

6.3.1 基本概念

为 Fraction 类重载算术运算符 /，实现两个分数的除法运算

```
Fraction operator/(const Fraction &left, const Fraction &right) {
    Fraction result(left.numerator()*right.denominator(),
        left.denominator()*right.numerator());
    return result;
}
```

调用方式 1：

```
Fraction a, b;
a \ b;
```

6.3.1 基本概念

为 Fraction 类重载算术运算符 /，实现两个分数的除法运算

```
Fraction operator/(const Fraction &left, const Fraction &right) {
    Fraction result(left.numerator()*right.denominator(),
        left.denominator()*right.numerator());
    return result;
}
```

调用方式 1：

```
Fraction a, b;
a \ b;
```

调用方式 2：

```
Fraction a, b;
operator/(a, b); //与 a / b 等价
```

6.3.1 基本概念

运算符函数作为类的成员函数：

- 成员运算符函数的显式参数比运算符的目数少一个；
- 运算符函数的第一个运算对象与隐含的 this 指针绑定。

6.3.1 基本概念

示例

```
class Fraction {
public:
    Fraction &operator*=(const Fraction &rhs) {
        m_numerator *= rhs.m_numerator;
        m_denominator *= rhs.m_denominator;
        return *this;
    }
};
```

6.3.1 基本概念

示例

```
class Fraction {
public:
    Fraction &operator*=(const Fraction &rhs) {
        m_numerator *= rhs.m_numerator;
        m_denominator *= rhs.m_denominator;
        return *this;
    }
};
```

调用方式 1：

```
Fraction a, b;
a *= b;
```

6.3.1 基本概念

示例

```
class Fraction {
public:
    Fraction &operator*=(const Fraction &rhs) {
        m_numerator *= rhs.m_numerator;
        m_denominator *= rhs.m_denominator;
        return *this;
    }
};
```

调用方式 1：

```
Fraction a, b;
a *= b;
```

调用方式 2：

```
Fraction a, b;
a.operator*=(b); //与 a *= b 等价
```

6.3.1 基本概念

示例

```
class Fraction {
public:
    Fraction &operator*=(const Fraction &rhs) {
        m_numerator *= rhs.m_numerator;
        m_denominator *= rhs.m_denominator;
        return *this;
    }
};
```

调用方式 1：

```
Fraction a, b;
a *= b;
```

调用方式 2：

```
Fraction a, b;
a.operator*=(b); //与 a *= b 等价
```

提示

不可重载运算符：::、.、?: 和 .*。

6.3.2 重载原则

1. 含义与内置类型一致

```
bool operator==(const Fraction &left, const Fraction &right) {
   return left.numerator()*right.denominator() ==
       left.denominator()*right.numerator();
}
```

6.3.2 重载原则

1. 含义与内置类型一致

```
bool operator==(const Fraction &left, const Fraction &right) {
    return left.numerator()*right.denominator() ==
        left.denominator()*right.numerator();
}
```

2. 行为与内置类型一致

```
Fraction &Fraction::operator=(const Fraction &rhs) {//返回左值对象的引用
    if (&rhs == this) return *this; //不会给自己赋值，例如， a = a;
    m_numerator = rhs.m_numerator;
    m_denominator = rhs.m_denominator;
    return *this;
}
```

6.3.2 重载原则

默认赋值运算符

没有显式重载赋值运算符时，编译器会自动合成一个默认的赋值运算符。

- 参数与返回值类型与自定义的函数一致；
- 使用赋值运算符将右侧对象的每个数据成员逐个赋值给左侧对象相应的数据成员；
- 对于数组成员，逐元素赋值。

6.3.2 重载原则

默认赋值运算符

没有显式重载赋值运算符时，编译器会自动合成一个默认的赋值运算符。

- 参数与返回值类型与自定义的函数一致；
- 使用赋值运算符将右侧对象的每个数据成员逐个赋值给左侧对象相应的数据成员；
- 对于数组成员，逐元素赋值。

提示：运算符属性继承

- 重载的运算符函数不会改变运算符的优先级、结合性以及目数等属性；
- 运算符的求值次序无法应用于重载的运算符。

6.3.2 运算符重载 — 重载原则

3. 成员函数的选择

类成员函数还是类的辅助函数：

- 赋值（=）、下标（[]）、函数调用（()）和成员访问箭头（->）运算符必须是类成员；
- 改变运算对象自身状态的运算符应该是类成员，比如复合赋值、自增、自减运算符等；
- 具有对称性的运算符一般应作为类的辅助函数，比如算术、关系、逻辑运算符等。

6.3.2 重载原则

提示

- 如果运算符作为类的成员函数，左侧对象一定是该类类型对象；

6.3.2 重载原则

提示

- 如果运算符作为类的成员函数，左侧对象一定是该类类型对象；
- 含有类类型的混合类型表达式，运算符应该定义成辅助函数，通常定义为友元函数。

```
class Fraction {
    friend Fraction operator/(int left, const Fraction &right);
};

Fraction operator/ (int left, const Fraction &right) {
    return Fraction(left*right.m_denominator, right.m_numerator);
}
```

6.3.2 重载原则

4. 其它原则

- 只能重载 C++ 语言已经存在的运算符，不能更改或创造新的运算符；
- 重载运算符的运算对象至少有一个是类类型。

6.3.2 重载原则

4. 其它原则

- 只能重载 C++ 语言已经存在的运算符，不能更改或创造新的运算符；
- 重载运算符的运算对象至少有一个是类类型。

下面代码是否正确？

```
int operator-(int, int);
```

6.3.2 重载原则

4. 其它原则

- 只能重载 C++ 语言已经存在的运算符，不能更改或创造新的运算符；
- 重载运算符的运算对象至少有一个是类类型。

下面代码是否正确？

```
int operator-(int, int);
```

错误，不能重新定义内置运算符。

6.3.3 输入和输出运算符

输入和输出运算符

重载 IO 标准库运算符 << 和 >> 来实现输出和输入操作，为了与 IO 标准库兼容，IO 运算符被声明为友元。

6.3.3 输入和输出运算符

输入和输出运算符

重载 IO 标准库运算符 << 和 >> 来实现输出和输入操作，为了与 IO 标准库兼容，IO 运算符被声明为友元。

示例

```
class Fraction {
    friend ostream& operator << (ostream &os, const Fraction &a);
    friend istream& operator >> (istream &is, Fraction &a);
};
```

6.3.3 输入和输出运算符

重载输出运算符

- 第一个形参是一个非常量 ostream 类对象，需要向其写入数据；
- 第二个形参是类的常量引用，避免对实参进行写操作；
- 返回 ostream 类对象的引用。

6.3.3 输入和输出运算符

重载输出运算符

- 第一个形参是一个非常量 ostream 类对象，需要向其写入数据；
- 第二个形参是类的常量引用，避免对实参进行写操作；
- 返回 ostream 类对象的引用。

示例

```
ostream& operator << (ostream &os, const Fraction &a) {
    os << a.m_numerator << a.m_denominator;
    return os;
}
```

6.3.3 输入和输出运算符

重载输出运算符

- 第一个形参是一个非常量 ostream 类对象，需要向其写入数据；
- 第二个形参是类的常量引用，避免对实参进行写操作；
- 返回 ostream 类对象的引用。

示例

```
ostream& operator << (ostream &os, const Fraction &a) {
    os << a.m_numerator << a.m_denominator;
    return os;
}
```

问题：为什么要返回 ostream 类对象的引用？

6.3.3 输入和输出运算符

重载输出运算符

- 第一个形参是一个非常量 ostream 类对象，需要向其写入数据；
- 第二个形参是类的常量引用，避免对实参进行写操作；
- 返回 ostream 类对象的引用。

示例

```
ostream& operator << (ostream &os, const Fraction &a) {
    os << a.m_numerator << a.m_denominator;
    return os;
}
```

问题：为什么要返回 ostream 类对象的引用？

因为可以实现连续输出：

```
Fraction a, b;
cout << a << b << endl;
```

6.3.3 输入和输出运算符

重载输出运算符

- 第一个形参为 istream 类对象的引用，能返回 istream 类对象的引用；
- 第二个形参是类对象的常量引用，要对类对象进行写操作；
- 返回 istream 类对象的引用。

6.3.3 输入和输出运算符

重载输出运算符

- 第一个形参为 istream 类对象的引用，能返回 istream 类对象的引用；
- 第二个形参是类对象的常量引用，要对类对象进行写操作；
- 返回 istream 类对象的引用。

示例

```
istream& operator >> (istream &is, Fraction &a) {
    is >> a.m_numerator >> a.m_denominator;
    return is;
}
```

6.3.3 输入和输出运算符

下面程序是否有问题？应如何修改？

```
class Fraction {
    int m_numerator = 0;
    int m_denominator = 1;
public:
    istream& operator>> (istream &is) {
        is >> m_numerator >> m_denominator;
        return is;
    }
};
int main() {
    Fraction a;
    cin >> a;
    return 0;
}
```

6.3.3 输入和输出运算符

答案

```
class Fraction {
    int m_numerator = 0;
    int m_denominator = 1;
public:
    istream& operator>> (istream &is) {
        is >> m_numerator >> m_denominator;
        return is;
    }
};
int main() {
    Fraction a;
    a.operator>>(cin);//类的成员函数的调用方式
    return 0;
}
```

6.3.4 递增和递减运算符

递增运算符的声明

```
class Fraction {
public:
    Fraction operator++(); //前置版本
    Fraction operator++(int);//后置版本
};
```

6.3.4 递增和递减运算符

递增运算符的声明

```
class Fraction {
public:
    Fraction operator++(); //前置版本
    Fraction operator++(int);//后置版本
};
```

前置递增

```
Fraction& Fraction::operator++() {
    ++m_numerator; //分子加1
    return *this; //为了与内置版本行为一致，前置递增运算符返回对象的引用
}
```

6.3.4 递增和递减运算符

递增运算符的声明

```
class Fraction {
public:
    Fraction operator++(); //前置版本
    Fraction operator++(int);//后置版本
};
```

前置递增

```
Fraction& Fraction::operator++() {
    ++m_numerator; //分子加1
    return *this; //为了与内置版本行为一致，前置递增运算符返回对象的引用
}
```

后置递增

```
Fraction Fraction::operator++(int) {
    Fraction a(*this); //保存当前值
    m_numerator++; //分子加1
    return a; //返回保存值，返回临时值，只能返回非引用类型
}
```

6.3.4 递增和递减运算符

问题：为什么后置版本采用一个额外的 int 形参？

6.3.4 递增和递减运算符

问题：为什么后置版本采用一个额外的 int 形参？

区别前置和后置版本。

6.3.5 函数调用运算符

函数调用运算符

一个类重载了函数调用运算符，可以通过函数的方式来使用该类的对象，该类的对象称为函数对象（function object）。

6.3.5 函数调用运算符

函数调用运算符

一个类重载了函数调用运算符，可以通过函数的方式来使用该类的对象，该类的对象称为函数对象（function object）。

示例

在 Fraction 类里面重载函数调用运算符，该运算符函数有两个形参，分别用来设置分子和分母，返回调用对象的 const 引用

```
class Fraction {
public:
    const Fraction& operator()(int a, int b) {
        m_numerator = a;
        m_denominator = b;
        return *this;
    }
};

Fraction f;
f(3, 5); //调用函数调用运算符
```

6.3.5 函数调用运算符

函数调用运算符和普通函数

和普通函数相比，函数对象不但能以函数方式使用，它还能存储状态，因此它比普通函数更加灵活。

6.3.5 函数调用运算符

函数调用运算符和普通函数

和普通函数相比，函数对象不但能以函数方式使用，它还能存储状态，因此它比普通函数更加灵活。

提示：函数调用运算符可以重载

一个类可以重载多个版本的函数调用运算符，但它们必须为类的非静态成员函数。

6.3.5 类型转换运算符

类型转换运算符（conversion operator）

- 将类类型数据转换成其它类型数据；
- 语法格式如下：
  operator type () const;

6.3.5 类型转换运算符

类型转换运算符（conversion operator）

- 将类类型数据转换成其它类型数据；
- 语法格式如下：
  operator type () const;

示例

为 Fraction 类添加一个转换为 double 类型的类型转换运算符：

```
class Fraction {
public:
    Fraction(int above = 0, int below = 1) :
        m_numerator(above), m_denominator(below) {}
    operator double() const {
        return 1.*m_numerator/m_denominator;
    }
};
```

6.3.5 类型转换运算符

问题：下面代码是否正确，为什么？

```
Fraction f = 2;
```

6.3.5 类型转换运算符

问题：下面代码是否正确，为什么？

```
Fraction f = 2;
```

正确，调用默认构造函数，将 int 类型隐式转换成 Fraction 类型。

6.3.5 类型转换运算符

问题：下面代码是否正确，为什么？

```
Fraction f = 2;
```

正确，调用默认构造函数，将 int 类型隐式转换成 Fraction 类型。

```
double x = 1.5 + f;
```

6.3.5 类型转换运算符

问题：下面代码是否正确，为什么？

```
Fraction f = 2;
```

正确，调用默认构造函数，将 int 类型隐式转换成 Fraction 类型。

```
double x = 1.5 + f;
```

正确，调用类型转换运算符，将 f 隐式转换成 double 类型。

6.3.5 类型转换运算符

问题：下面代码是否正确，为什么？

```
Fraction f = 2;
```

正确，调用默认构造函数，将 int 类型隐式转换成 Fraction 类型。

```
double x = 1.5 + f;
```

正确，调用类型转换运算符，将 f 隐式转换成 double 类型。

```
int i = f;
```

6.3.5 类型转换运算符

问题：下面代码是否正确，为什么？

```
Fraction f = 2;
```

正确，调用默认构造函数，将 int 类型隐式转换成 Fraction 类型。

```
double x = 1.5 + f;
```

正确，调用类型转换运算符，将 f 隐式转换成 double 类型。

```
int i = f;
```

正确：首先将 f 隐式转换成 double 类型，然后将 double 类型转换为 int 类型。

6.3.5 类型转换运算符

类型转换运算符的特点

- 以隐式的方式调用；
- 没有显式的返回值类型，也没有形参；
- 必须为类的成员函数；
- 一般将类型转换运算符声明为 const 成员。

6.3.5 类型转换运算符

类型转换运算符的特点

- 以隐式的方式调用；
- 没有显式的返回值类型，也没有形参；
- 必须为类的成员函数；
- 一般将类型转换运算符声明为 const 成员。

提示

不要过度依赖类型转换运算符。

6.4 静态成员

静态成员

- 类对象共享；
- 与类相关联；
- 存储于一个公用的内存中。

6.4.1 声明静态成员

声明静态成员

- 在成员声明之前加上关键字 static 使其与类相关联；
- 静态数据成员在程序启动时创建，在程序结束时消亡；
- 类型可以为常量、引用、指针、类类型。

6.4.1 声明静态成员

声明静态成员

- 在成员声明之前加上关键字 static 使其与类相关联；
- 静态数据成员在程序启动时创建，在程序结束时消亡；
- 类型可以为常量、引用、指针、类类型。

示例

为 Fraction 类添加一个转换为 double 类型的类型转换运算符：

```
class PartTimeWorker {
    string m_name; //员工姓名
    double m_hours;//工作小时数
    static double ms_payRate;//小时工资
public:
    double salary();
    static double rate() {
        return ms_payRate;
    }
    static void initRate(double rate);
};
```

6.4.1 声明静态成员

静态数据和类对象

一个类的静态数据成员不属于类对象，类对象不包含任何静态数据成员。

6.4.1 声明静态成员

静态数据和类对象

一个类的静态数据成员不属于类对象，类对象不包含任何静态数据成员。

在静态成员函数内部能不能直接使用 this 指针?

6.4.1 声明静态成员

静态数据和类对象

一个类的静态数据成员不属于类对象，类对象不包含任何静态数据成员。

在静态成员函数内部能不能直接使用 this 指针?

不能，静态成员函数不与任何对象绑定，因此不包含 this 指针，也就是说不能直接访问普通成员，而且也不能声明为常量成员函数。

6.4.2 使用静态成员

定义静态成员

为静态数据成员不属于类对象，因此他们并不是由构造函数来初始化。它们必须放到类的外部定义和初始化。

- 在类外定义的时候必须要指明所属的类；
- 不能重复 static 关键字。

6.4.2 使用静态成员

定义静态成员：在类外部提供初始值

```
class PartTimeWorker {//其他成员省略
    static double ms_payRate;//小时工资
};
double PartTimeWorker::ms_payRate = 7.53;
```

6.4.2 使用静态成员

定义静态成员：在类外部提供初始值

```
class PartTimeWorker {//其他成员省略
    static double ms_payRate;//小时工资
};
double PartTimeWorker::ms_payRate = 7.53;
```

定义静态成员：在类内部提供初始值

```
class PartTimeWorker {//其它成员与前面一致
    static const int ms_maxHourWeek = 20; //每周最长工作时间
};
const int PartTimeWorker::ms_maxHourWeek;//在类内部已经有了初始值，则在类外部不可以
 再提供初始值
```

6.4.2 使用静态成员

定义静态成员：在类外部提供初始值

```
class PartTimeWorker {//其他成员省略
    static double ms_payRate;//小时工资
};
double PartTimeWorker::ms_payRate = 7.53;
```

定义静态成员：在类内部提供初始值

```
class PartTimeWorker {//其它成员与前面一致
    static const int ms_maxHourWeek = 20; //每周最长工作时间
};
const int PartTimeWorker::ms_maxHourWeek;//在类内部已经有了初始值，则在类外部不可以
 再提供初始值
```

定义静态成员：可以在类内部也可以在类外部定义静态成员函数

静态成员函数 initRate 在类外的定义：

```
void PartTimeWorker::initRate(double rate) {
    ms_maxHourWeek = rate;
}
```

6.4.2 静态成员 — 使用静态成员

使用静态成员：在类外可以通过类名直接访问公有的静态成员

```
cout << PartTimeWorker::rate() << endl; //通过类名访问静态成员
```

6.4.2 静态成员 — 使用静态成员

使用静态成员：在类外可以通过类名直接访问公有的静态成员

```
cout << PartTimeWorker::rate() << endl; //通过类名访问静态成员
```

使用静态成员：通过类对象来访问静态成员

```
PartTimeWorker kevin;
cout << kevin.rate() << endl; //通过类对象访问静态成员函数
```

6.4.2 静态成员 — 使用静态成员

使用静态成员：在类外可以通过类名直接访问公有的静态成员

```
cout << PartTimeWorker::rate() << endl; //通过类名访问静态成员
```

使用静态成员：通过类对象来访问静态成员

```
PartTimeWorker kevin;
cout << kevin.rate() << endl; //通过类对象访问静态成员函数
```

使用静态成员：成员函数内部可以直接访问静态成员

```
double PartTimeWorker::salary() {
    return ms_payRate * m_hours; //类内使用静态数据成员
}
```

6.5 类成员指针

类成员指针（pointer to member）

指向类的非静态成员的指针。

6.5.1 数据成员指针

数据成员指针

- 需要指明类成员的类型；
- 需要显式指明成员所属的类。

6.5.1 数据成员指针

数据成员指针

- 需要指明类成员的类型；
- 需要显式指明成员所属的类。

示例：类成员指针的定义

定义一个指向 PartTimeWorker 类数据成员 m_name 的指针：

```
string PartTimeWorker::*p1 = &PartTimeWorker::m_name;
auto p2 = &PartTimeWorker::m_name;//利用 auto 简化指针的定义
```

6.5.1 数据成员指针

数据成员指针

- 需要指明类成员的类型；
- 需要显式指明成员所属的类。

示例：类成员指针的定义

定义一个指向 PartTimeWorker 类数据成员 m_name 的指针：

```
string PartTimeWorker::*p1 = &PartTimeWorker::m_name;
auto p2 = &PartTimeWorker::m_name;//利用 auto 简化指针的定义
```

示例：类成员指针的使用

数据成员指针没有和类对象关联，可以将指针成员作用于类的对象来访问类的成员。

```
PartTimeWorker w1, *w2 = &w1;
cout << w1.m_name << endl; //普通访问方式
cout << w1.*p1 << endl; //数据成员指针访问方式，等价于 w1.m_name
cout << w2->*p1 << endl; //数据成员指针访问方式，等价于 w2->m_name
```

6.5.1 数据成员指针

注意

类的访问控制规则同样适用于成员指针：
m_name 为 PartTimeWorker 类的私有数据成员，因此上面的代码必须位于类的成员函数或类的友元函数中；否则，会出现编译错误。

6.5.2 成员函数指针

成员函数指针

- 需要指明类成员函数的类型；
- 需要显式指明成员函数所属的类。

6.5.2 成员函数指针

成员函数指针

- 需要指明类成员函数的类型；
- 需要显式指明成员函数所属的类。

示例：成员函数指针的定义

定义一个指向 PartTimeWorker 类的成员函数指针 pf，pf 可以指向 PartTimeWorker 类中返回类型为 double 且无参的成员函数。

6.5.2 成员函数指针

成员函数指针

- 需要指明类成员函数的类型；
- 需要显式指明成员函数所属的类。

示例：成员函数指针的定义

定义一个指向 PartTimeWorker 类的成员函数指针 pf，pf 可以指向 PartTimeWorker 类中返回类型为 double 且无参的成员函数。

```
double (PartTimeWorker::*pf)();
pf = &PartTimeWorker::salary;
```

6.5.2 成员函数指针

成员函数指针

- 需要指明类成员函数的类型；
- 需要显式指明成员函数所属的类。

示例：成员函数指针的定义

定义一个指向 PartTimeWorker 类的成员函数指针 pf，pf 可以指向 PartTimeWorker 类中返回类型为 double 且无参的成员函数。

```
double (PartTimeWorker::*pf)();
pf = &PartTimeWorker::salary;

auto pf2 = &PartTimeWorker::salary;
```

6.5.2 成员函数指针

成员函数指针

- 需要指明类成员函数的类型；
- 需要显式指明成员函数所属的类。

示例：成员函数指针的定义

定义一个指向 PartTimeWorker 类的成员函数指针 pf，pf 可以指向 PartTimeWorker 类中返回类型为 double 且无参的成员函数。

```
double (PartTimeWorker::*pf)();
pf = &PartTimeWorker::salary;

auto pf2 = &PartTimeWorker::salary;


using PTWS = double (PartTimeWorker::*)();
PTWS pf3 = &PartTimeWorker::salary;
```

6.5.2 成员函数指针

示例：类成员指针的使用

需要使用 .* 或者 ->* 运算符作用于指向成员函数的指针，来调用类成员函数。

```
PartTimeWorker w;
cout << w.salary() << endl; //普通调用方式
cout << (w.*pf)() << endl; //成员函数指针调用方式
```

6.5.2 成员函数指针

示例：类成员指针的使用

需要使用 .* 或者 ->* 运算符作用于指向成员函数的指针，来调用类成员函数。

```
PartTimeWorker w;
cout << w.salary() << endl; //普通调用方式
cout << (w.*pf)() << endl; //成员函数指针调用方式
```

(w.*pf) 的圆括号能否省略？

6.5.2 成员函数指针

示例：类成员指针的使用

需要使用 .* 或者 ->* 运算符作用于指向成员函数的指针，来调用类成员函数。

```
PartTimeWorker w;
cout << w.salary() << endl; //普通调用方式
cout << (w.*pf)() << endl; //成员函数指针调用方式
```

(w.*pf) 的圆括号能否省略？

- (w.*pf) 的圆括号不能缺少，原因是函数调用运算符的优先级高于指向成员指针运算符的优先级。

本章结束