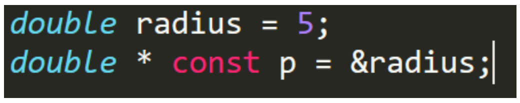
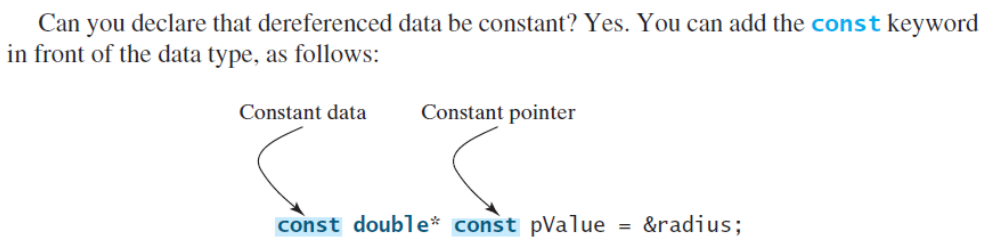
OOP复习

- 匿名对象
- 内联函数（空间换时间）
    - 如果一个函数在类定义内实现，则自动成为内联函数
- 实例成员，静态成员
    - 静态函数不能访问类的实例成员
    - 只有实例成员函数可以被定义为只读函数
- 常量指针



```cpp
double radius = 5;
double * const p = &radius;
```

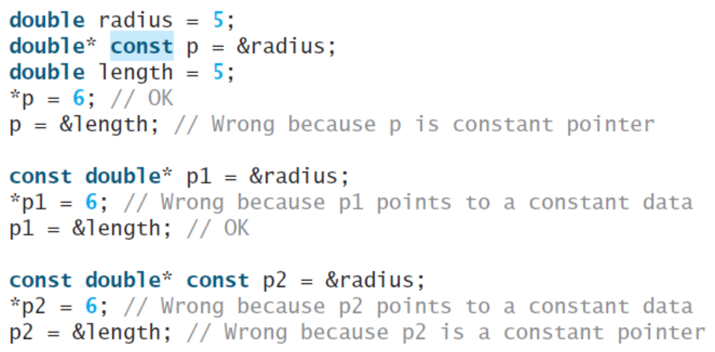
- 声明和初始化必须在同一条语句中，后面的程序不能为其赋予新的地址,
- 但p指向的数据不是常量，可以更改
    - *p = 10;



Can you declare that dereferenced data be constant? Yes. You can add the `const` keyword in front of the data type, as follows:

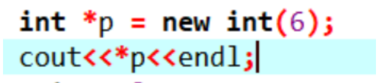
```cpp
const double* const pValue = &radius;
```

- 指针时常量，指向的数据也是常量
- const double* p = &radius;
    - 指针不是常量，但指向的数据是常量



```cpp
double radius = 5;
double* const p = &radius;
double length = 5;
*p = 6; // OK
p = &length; // Wrong because p is constant pointer

const double* p1 = &radius;
*p1 = 6; // Wrong because p1 points to a constant data
p1 = &length; // OK

const double* const p2 = &radius;
*p2 = 6; // Wrong because p2 points to a constant data
p2 = &length; // Wrong because p2 is a constant pointer
```

- 动态持久内存分配



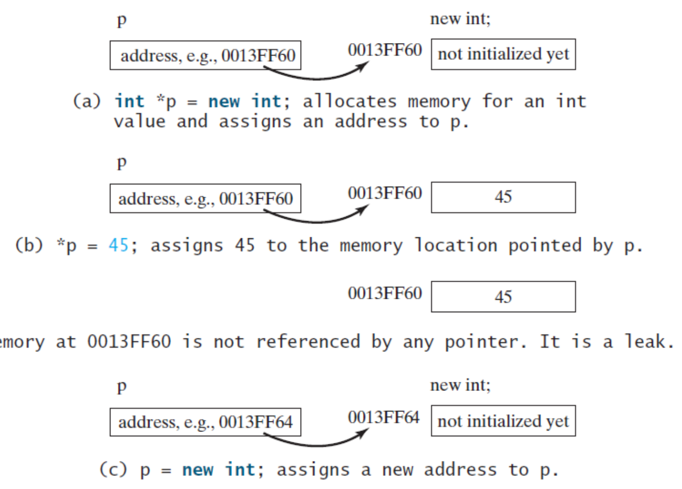
```cpp
int *p = new int(6);
cout<<*p<<endl;
```

- new int告诉计算机在运行时为一个int变量分配内存空间，并初始化为4，故输出为4
- 动态数组：
    - int *list = new int[4]；
    - 长度为4的数组

- 内存泄漏

```
1  int* p = new int;
2  *p = 45;
3  p = new int;
```

Line 1 declares a pointer assigned with a memory address for an `int` value, as shown in Figure 11.4a. Line 2 assigns `45` to the variable pointed by `p`, as shown in Figure 11.4b. Line 3 assigns a new memory address to `p`, as shown in Figure 11.4c. The original memory space that holds value `45` is not accessible, because it is not pointed to by any pointer. This memory cannot be accessed and cannot be deleted. This is a *memory leak*.



p
new int;
address, e.g., 0013FF60
0013FF60
not initialized yet
(a) int *p = new int; allocates memory for an int value and assigns an address to p.
p
address, e.g., 0013FF60
0013FF60
45
(b) *p = 45; assigns 45 to the memory location pointed by p.
0013FF60
45
Memory at 0013FF60 is not referenced by any pointer. It is a leak.
p
new int;
address, e.g., 0013FF64
0013FF64
not initialized yet
(c) p = new int; assigns a new address to p.

FIGURE 11.4 Unreferenced memory space causes memory leak.

- p1 and p2 point to the same data, but it was destroyed using delete p1. So you cannot access it later using *p2.

What is wrong in the following code:

```
double* p1 = new double;
double* p2 = p1;
*p2 = 5.4;
delete p1;
cout << *p2 << endl;
```

- 创建及访问动态对象
  - ClassName* pObject = new ClassName() 或 ClassName *pObject = new ClassName
- 拷贝构造函数
  - 函数签名
    - ClassName(const ClassName&)
  - shallow copy
    - *Yes. They are the same, but s1 = s2 is better than s1 = string(s2), because the latter creates a temporary string object for s2, and then copied to s1.*
    - 缺省拷贝构造函数和赋值运算符（=）执行的是浅拷贝
  - deep copy
  - 拷贝构造函数和赋值函数非常容易混淆，常导致错写、错用。拷贝构造函数是在对象被创建时调用的，而赋值函数只能被已经存在了的对象调用。（U3P54
- 模板类
  - 在模板类中可以定义静态成员。每个模板特例化都拥有独有的静态数据域拷贝。

- 默认类型只能用于模板类，不能用于模板函数
- Function templates cannot have default values.

What is wrong in the following code?

```
template<typename T = int>
void printArray(const T list[], int arraySize)
{
  for (int i = 0; i < arraySize; i++)
  {
    cout << list[i] << " ";
  }
  cout << endl;
}
```

- 运算符重载
  - bool Rational::operator<(const Rational& secondRational) const

```
bool operator<(const Rational& secondRational) const
```

defines the < operator function that returns true if this Rational object is less than secondRational. You can invoke the function using

```
r1.operator<(r2)
```

or simply

```
r1 < r2
```

  - 不可重载运算符

Table 14.2 Operators That Cannot Be Overloaded

?:	.	.*	::

  - 重载[]运算符
    - 返回一个引用
      - int& operator[](int index);
    - 需要返回引用的左值运算符
      - += -= *= /= %=
      - 重载+=

```
Rational& Rational::operator+=(const Rational& secondRational)
{
  *this = add(secondRational);
  return *this;
}
```

  - 重载++ --
    - 若定义后缀形式，使用一个特殊的int型伪参数来表示，而前缀形式不需要任何参数
    - 前缀++、--是左值运算符，而后缀++、--不是
      - 前缀返回引用

```
// Prefix increment
Rational& Rational::operator++()
{
  numerator += denominator;
  return *this;
}

// Postfix increment
Rational Rational::operator++(int dummy)
{
  Rational temp(numerator, denominator);
  numerator += denominator;
  return temp;
}
```

- 友元函数和友元类
  - < < > >必须以非成员的友元函数形式实现
  - 重载< <

```
friend ostream& operator<<(ostream& out, const Rational& rational);
```

```
ostream& operator<<(ostream& out, const Rational& rational)
{
  out << rational.numerator << "/" << rational.denominator;
  return out;
}
```

  - 重载> >

```
friend istream& operator>>(istream& in, Rational& rational);
```

```
istream& operator>>(istream& in, Rational& rational)
{
  cout << "Enter numerator: ";
  in >> rational.numerator;

  cout << "Enter denominator: ";
  in >> rational.denominator;
  return in;
}
```

- 自动类型转换
  - 转换为基本数据类型
    - 函数没有返回类型，函数名就是期望将对象转换到的目标类型

```
Rational::operator double()
{
  return doubleValue(); // doubleValue() already in Rational.h
}
```

Don’t forget that you have to add the member function header in the Rational.h header file.

```
operator double();
```

    - double d = re + 5.1;
      - r1先被转换为一个double型值0.25
  - 转换为对象类型
    - 定义构造函数

To achieve this, define the following constructor in the header file:

```
Rational(int numerator);
```

- 注意，+运算符也被重载了

```
Rational r1(2, 3);
Rational r = r1 + 4; // Automatically converting 4 to Rational
cout << r << endl;
```

- 当C++看到r1+4，会首先检查+运算符是否被重载用于Rational对象和整数的加法。如果没有这一定义，才会继续查找+运算符是否被用于连个Rational对象的加法操作。这里，4是个整数，C++使用构造函数基于该整数生成了一个Rational对象。（因为构造函数的存在，这一自动转换是可行的）
- 一个类不能既有对象->基本数据类型；又有构造函数基本数据类型->对象

A class can define the conversion function to convert an object to a primitive type value or define a conversion constructor to convert a primitive type value to an object, but not both simultaneously in the class. If both are defined, the compiler will report an ambiguity error.

- 定义重载运算符的非成员函数
  - 如果一个运算符能够被重载成非成员函数，将其定义为非成员函数可以实现隐式类型转换
  - 左边是+运算符的调用者，它必须是一个对象才可以

You can add a **Rational** object **r1** with an integer like this:

```
r1 + 4
```

Can you add an integer with a **Rational** object **r1** like this?

```
4 + r1
```

  - 1.定义构造函数 2.将+运算符定义为非成员函数

1. Define and implement the following constructor, as discussed in the preceding section.

   ```
   Rational(int numerator);
   ```

   This constructor enables the integer to be converted to a **Rational** object.

2. Define the + operator as a nonmember function in the Rational.h header file as follows:

   ```
   Rational operator+(const Rational& r1, const Rational& r2)
   ```

   Implement the function in Rational.cpp as follows:

   ```
   Rational operator+(const Rational& r1, const Rational& r2)
   {
     return r1.add(r2);
   }
   ```

  - const用在非成员函数上
- 重载赋值运算符
  - const Course& operator=(const Course course);

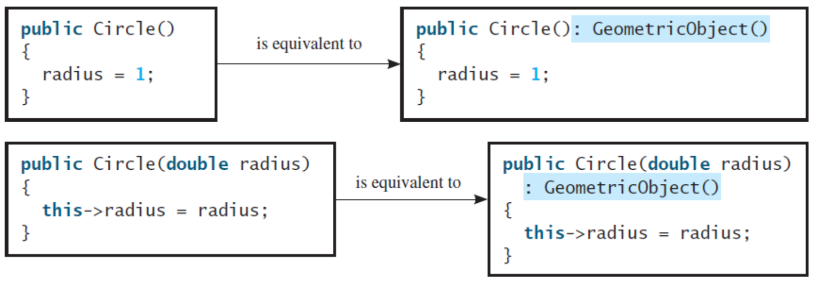
- 继承与多态
- 不会自动继承的函数
  - constructor
  - destuctor
  - overloading operator new
  - assignment operation =
  - friend relation
- 泛型程序设计
  - 如果一个函数的参数是基类，可以向它传递任何派生类的对象
- 构造函数和析构函数
  - 派生类不继承基类的构造函数，派生类仅仅在自己的构造函数中调用基类的构造函数

```
DerivedClass(parameterList): BaseClass()
{
  // Perform initialization
}
```

or

```
DerivedClass(parameterList): BaseClass(argumentList)
{
  // Perform initialization
}
```

  - 派生类的构造函数总是显式或隐式地调用基类的构造函数。如果基类中的构造函数没有被显式调用，基类的无参构造函数会被默认调用。



```
public Circle()
{
  radius = 1;
}
```

```
public Circle(): GeometricObject()
{
  radius = 1;
}
```

```
public Circle(double radius)
{
  this->radius = radius;
}
```

```
public Circle(double radius)
  : GeometricObject()
{
  this->radius = radius;
}
```

  - Apple类没有显式的构造函数，因此其默认的无参构造函数会被隐式声明。由于Apple是Fruit的派生类，Apple的缺省无参构造函数会自动调用Fruit的无参构造函数，而Fruit没有无参构造函数

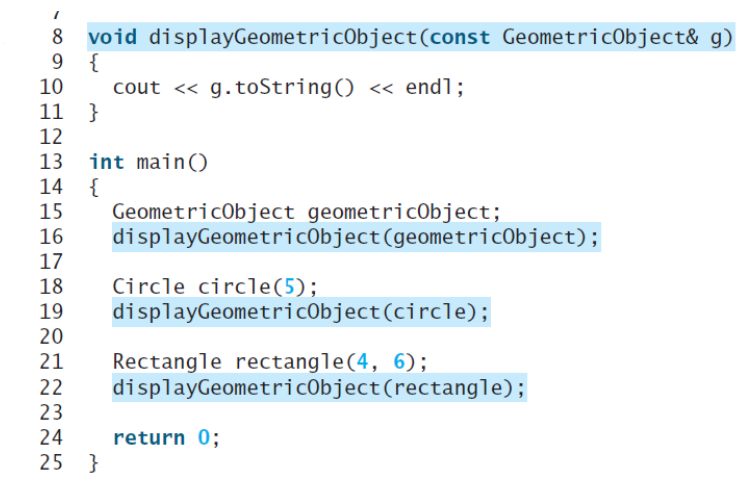
```
class Fruit
{
public:
  Fruit(int id)
  {
  }
};

class Apple: public Fruit
{
public:
  Apple()
  {
  }
};
```

  - 如果基类有一个自定义的拷贝构造函数和赋值操作，应该在派生类中自定义这些来保证基类中的数据域被正确拷贝

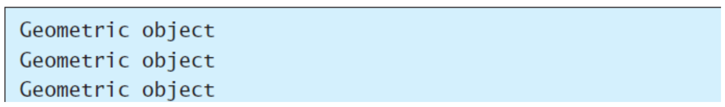
the copy constructor in Child would typically look like this:

```
Child::Child(const Child& object): Parent(object)
{
  // Write the code for copying data fields in Child
}
```

- 函数重定义
    - 在基类中定义的函数能够在派生类中被重新定义
    - 可以重定义基类中的静态成员函数
- 多态
    - 多态意味着一个超类型的变量可引用一个子类型的对象。在任何其基类对象使用的地方，派生类对象也能够被使用

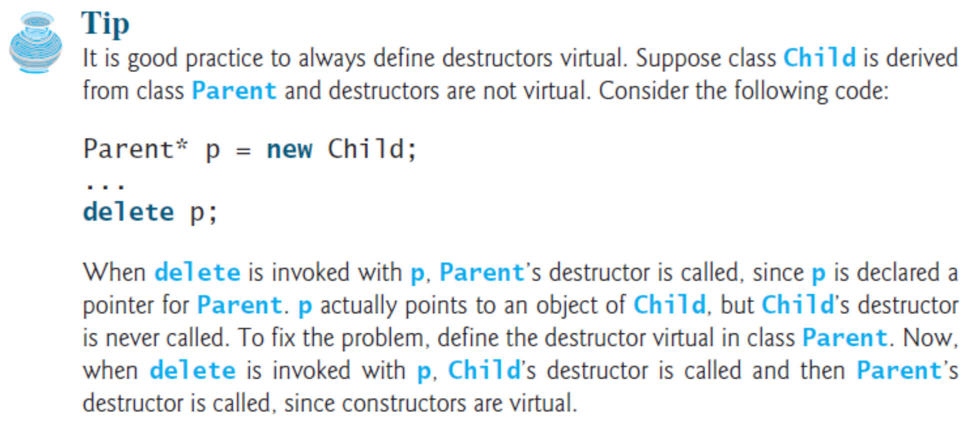


```
void displayGeometricObject(const GeometricObject& g)
{
  cout << g.toString() << endl;
}

int main()
{
  GeometricObject geometricObject;
  displayGeometricObject(geometricObject);

  Circle circle(5);
  displayGeometricObject(circle);

  Rectangle rectangle(4, 6);
  displayGeometricObject(rectangle);

  return 0;
}
```



```
Geometric object
Geometric object
Geometric object
```

- 虚函数
    - 虚函数使得系统能够基于对象的实际类型决定在运行时调用哪个函数
    - In C++, redefining a virtual function in a derived class is called overriding a function.
    - 在虚函数中，引用对象的变量必须以**引用**或者**指针**的形式传递



Tip

It is good practice to always define destructors virtual. Suppose class **Child** is derived from class **Parent** and destructors are not virtual. Consider the following code:

```
Parent* p = new Child;
...
delete p;
```

When **delete** is invoked with **p**, **Parent**'s destructor is called, since **p** is declared a pointer for **Parent**. **p** actually points to an object of **Child**, but **Child**'s destructor is never called. To fix the problem, define the destructor virtual in class **Parent**. Now, when **delete** is invoked with **p**, **Child**'s destructor is called and then **Parent**'s destructor is called, since constructors are virtual.

- 纯虚函数、抽象类
    - 析构函数不能是纯虚函数
    - 抽象类
        - 至少含有一个纯虚函数的类__抽象类。
        - 抽象类只能用作其它类的基类。不能创建实例，不能用作函数参数、函数返回类型以及显式转换的类型。
        - 可以声明抽象类的指针和引用，此指针指向派生类对象，实现动态束定。

- 纯虚函数和空函数不一样。
- 如果派生类中没有给出基类的全部纯虚函数的实现代码，它继承基类的这些没有实现的纯虚函数，则派生类还是一个抽象类。