运算符重载

友元

一般来说，类的私有成员只能在类的内部访问，类之外是不能访问它们的。但如果将其他类或函数设置为类的友元（friend），就可以访问了。用这个比喻形容友元可能比较恰当：将类比作一个家庭，类的`private`成员相当于家庭的秘密，一般的外人是不允许探听这些秘密的，只有`friend`（朋友）才有能力探听这些秘密。

友元的形式可以分为友元函数和友元类。

```
class 类名 {
    //...
    friend 函数原型;
    friend class 类名;
    //...
}
```

目前为止，我们学过的函数形式有两种：全局函数（自由函数/普通函数）和成员函数。接下来，我们分别进行讨论。现在我们有一个类Point，表示一个二维的点:

```
class Point {
public:
    Point(int ix = 0, int iy = 0)
    : _ix(ix)
    , _iy(iy)
    {}

    void print() const
    {
        cout << "(" << _ix
             << "," << _iy
             << ")";
    }
    friend float distance(const Point & lhs, const Point & rhs);
    friend float Line::distance(const Point & lhs, const Point & rhs);
private:
    int _ix;
    int _iy;
};
```

友元函数之全局函数

现在我们有一个全局函数`distance`，通过它计算两个点之间的距离，如果直接访问肯定是不行的，编译会报错。当我们在类Point中将其声明为友元之后，就可以了。

```
float distance(const Point & lhs, const Point & rhs)
{
    return sqrt((lhs._ix - rhs._ix) * (lhs._ix - rhs._ix) +
                (lhs._iy - rhs._iy) * (lhs._iy - rhs._iy));
}
```

友元函数之成员函数

假设类A有一个成员函数，该成员函数想去访问另一个类B类中的私有成员变量。这时候则可以在第二个类B中，声明第一个类A的那个成员函数为类B的友元函数，这样第一个类A的某个成员函数就可以访问第二个类B的私有成员变量了。同样还是求取两个点之间的距离，现在我们再定义一个类`Line`，由`Line`中的成员函数`distance`完成：

```
class Line {
public:
    float distance(const Point & lhs, const Point & rhs)
    {
          return sqrt((lhs._ix - rhs._ix) * (lhs._ix - rhs._ix) +
                      (lhs._iy - rhs._iy) * (lhs._iy - rhs._iy));
    }
};
```

友元之友元类

如上的例子，假设类`Line`中不止有一个`distance`成员函数，还有其他成员函数，它们都需要访问`Point`的私有成员，如果还像上面的方式一个一个设置友元，就比较繁琐了，可以直接将`Line`类设置为`Point`的友元。

```
class Point {
    //...
    friend class Line;
    //...
};
```

不可否认，友元在一定程度上将类的私有成员暴露出来，破坏了信息隐藏机制，似乎是种“副作用很大的药”，但俗话说“良药苦口”，好工具总是要付出点代价的，拿把锋利的刀砍瓜切菜，总是要注意不要割到手指的。
友元的存在，使得类的接口扩展更为灵活，使用友元进行运算符重载从概念上也更容易理解一些，而且，`C++`规则已经极力地将友元的使用限制在了一定范围内，它是单向的、不具备传递性、不能被继承，所以，应尽力合理使用友元。

注意：友元的声明是不受`public/protected/private`关键字限制的。

运算符重载

为什么需要对运算符进行重载?

`C++`预定义中的运算符的操作对象只局限于基本的内置数据类型，但是对于我们自定义的类型是没有办法操作的。但是大多时候我们需要对我们定义的类型进行类似的运算，这个时候就需要我们对这么运算符进行重新定义，赋予其新的功能，以满足自身的需求。比如:

```
class Complex {
public:
    Complex(double real = 0, double image = 0)
    : _real(real)
    , _image(image)
    {}
private:
    double _real;
```

```
    double _image;
};

void test()
{
    Complex c1(1, 2), c2(3, 4);
    Complex c3 = c1 + c2;//编译出错
}
```

为了使对用户自定义数据类型的数据的操作与内置的数据类型的数据的操作形式一致，C++ 提供了运算符的重载,通过把 C++ 中预定义的运算符重载为类的成员函数或者友元函数，使得对用户的自定义数据类型的数据(对象)的操作形式与 C++ 内部定义的类型的数据一致。

运算符重载的实质就是**函数重载**或**函数多态**。运算符重载是一种形式的 C++ 多态。目的在于让人能够用同名的函数来完成不同的基本操作。要重载运算符，需要使用被称为运算符函数的特殊函数形式，运算符函数形式：

```
返回类型 operator 运算符(参数表)
{
    //...
}
```

运算符重载的规则

运算符是一种通俗、直观的函数，比如：`int x = 2 + 3;`语句中的“+”操作符，系统本身就提供了很多个重载版本：

```
int operator+(int,int);
double operator+(double,double);
```

但并不是所有的运算符都可以重载。可以重载的运算符有：

+	-	*	/	%	^
&	|	~	!	=	<
>	+=	-=	*=	/=	%=
^=	&=	|=	>>	<<	>>=
<<=	==	!=	>=	<=	&&
||	++	--	->*	->	,
[]	()	new	delete	new[]	delete[]

不可以重载的运算符：`.` `.*` `?:` `::` `sizeof`

运算符重载还具有以下规则：

- 为了防止用户对标准类型进行运算符重载，C++ 规定重载的运算符的操作对象必须至少有一个是自定义类型或枚举类型
- 重载运算符之后，其优先级和结合性还是固定不变的。
- 重载不会改变运算符的用法，原来有几个操作数、操作数在左边还是在右边，这些都不会改变。
- 重载运算符函数不能有默认参数，否则就改变了运算符操作数的个数。

- 重载逻辑运算符（`&&`,`||`）后，不再具备短路求值特性。
- 不能臆造一个并不存在的运算符，如@、$等

运算符重载的形式

运算符重载的形式有三种：

- 采用普通函数的重载形式
- 采用成员函数的重载形式
- 采用友元函数的重载形式

以普通函数形式重载

在上面的例子中，`Complex`对象无法执行加法操作，接下来我们重载`+`运算符。由于之前的定义中`Complex`的成员都设置成了`private`成员，所以不能访问，我们需要在类中添加2个`getter`函数，获取其值。

```
class Complex {
public:
    //...
    double getReal() const {    return _real;   }
    double getImage() const {   return _image;  }
    //...
};

Complex operator+(const Complex & lhs, const Complex & rhs)
{
    return Complex(lhs.getReal() + rhs.getReal(),lhs.getImage() +
rhs.getImage());
}

void test0()
{
    Complex c1(1, 2), c2(3, 4);
    Complex c3 = c1 + c2;//编译通过
}
```

以成员函数形式重载

成员函数形式的运算符声明和实现与成员函数类似，首先应当在类定义中声明该运算符，声明的具体形式为：

```
返回类型  operator 运算符（参数列表）;
```

既可以在类定义的同时，定义运算符函数使其成为`inline`型，也可以在类定义之外定义运算符函数，但要使用作用域限定符“`::`”，类外定义的基本格式为：

```
返回类型  类名::operator 运算符（参数列表）
{
    //...
}
```

注意：用成员函数重载双目运算符时，左操作数无须用参数输入，而是通过隐含的`this`指针传入。

回到`Complex`的例子，如果以成员函数形式进行重载，则不需要定义`getter`函数：

```
class Complex{
public:
    //...
    Complex operator+(const Complex & rhs)
    {
        return Complex(_real + rhs._real, _image + rhs._image);
    }
};
```

以友元函数形式重载

如果以友元函数形式进行重载，同样不需要定义`get`函数：

```
class Complex{
    //...
    friend Complex operator+(const Complex & lhs, const Complex & rhs);
};

Complex operator+(const Complex & lhs, const Complex & rhs)
{
    return Complex(lhs._real + rhs._real, lhs._image + rhs._image);
}
```

运算符重载可以改变运算符内置的语义，如以友元函数形式定义的加操作符：

```
Complex operator+(const Complex& lhs,const Complex& rhs)
{
    return complex(lhs._real - rhs._real, lhs._image - rhs._image);
}
```

明明是加操作符，但函数内却进行的是减法运算，这是合乎语法规则的，不过却有悖于人们的直觉思维，会引起不必要的混乱。因此，除非有特别的理由，**尽量使重载的运算符与其内置的、广为接受的语义保持一致**。

特殊运算符的重载

自增自减运算符

自增运算符“`++`”和自减运算符“`--`”推荐以成员函数形式重载，分别包含两个版本，即运算符前置形式(如`++x`)和运算符后置形式(如`x++`)，这两者进行的操作是不一样的。因此，当我们在对这两个运算符进行重载时，就必须区分前置和后置形式。

C++根据参数的个数来区分前置和后置形式。如果按照通常的方法（成员函数不带参数）来重载`++/--`运算符，那么重载的就是前置版本。要对后置形式进行重载，就必须为重载函数再增加一个`int`类型的参数，该参数仅仅用来告诉编译器这是一个运算符后置形式，在实际调用时不需要传递实参。

```
class Complex {
public:
    //...
    //前置形式
    Complex & operator++()
    {
        ++_real;
```

```
        ++_image;
        return *this;
    }
    //后置形式
    Complex operator++(int) //int作为标记，并不传递参数
    {
        Complex tmp(*this);
        ++_real;
        ++_image;
        return tmp;
    }
};
```

赋值运算符

对于赋值运算符“=”，只能以成员函数形式进行重载，我们已经在类和对象中讲过了，就不再赘述，大家可以翻看前面的内容。

函数调用运算符

我们知道，普通函数执行时，有一个特点就是无记忆性，一个普通函数执行完毕，它所在的函数栈空间就会被销毁，所以普通函数执行时的状态信息，是无法保存下来的，这就让它无法应用在那些需要对每次的执行状态信息进行维护的场景。大家知道，我们学习了成员函数以后，有了对象的存在，对象执行某些操作之后，只要对象没有销毁，其状态就是可以保留下来的，但在函数作为参数传递时，会有障碍。为了解决这个问题，C++引入了函数调用运算符。函数调用运算符的重载形式只能是**成员函数**形式，其形式为:

```
返回类型 类名::operator()(参数列表)
{
    //...
}
```

在定义"()"运算符的语句中，第一对小括号总是空的，因为它代表着我们定义的运算符名，第二对小括号就是函数参数列表了，它与普通函数的参数列表完全相同。对于其他能够重载的运算符而言，操作数个数都是固定的，但函数调用运算符不同，它的参数是根据需要来确定的， 并不固定。

接下来，我们来看一个例子:

```
class Foo
{
public:
    Foo(int count = 0): _count(count){}

    void operator()(int x)
    {
        cout << " x = " << x << endl;
    }

    int operator()(int x, int y)
    {
        ++_count;
       return x + y;
    }

    int _count;
};
```

```
void test()
{
    Foo foo;
    int a = 3, b = 4;
    foo(a);
    cout << foo(a, b) << endl;
}
```

从例子可以看出，一个类如果重载了函数调用operator()，就可以将该类对象作为一个函数使用。对于这种重载了函数调用运算符的类创建的对象，我们称为函数对象（Function Object）。函数也是一种对象，这是泛型思考问题的方式。

下标访问运算符

下标访问运算符[]通常用于访问数组元素，它是一个二元运算符，如 arr[idx] 可以理解成 arr 是左操作数，idx 是右操作数。对下标访问运算符进行重载时，只能以成员函数形式进行，如果从函数的观点来看，语句 arr[idx]; 可以解释为 arr.operator[](idx);，因此下标访问运算符的重载形式如下：

```
返回类型 & 类名::operator[](参数类型);
返回类型 & 类名::operator[](参数类型) const;
```

下标运算符的重载函数只能有一个参数，不过该参数并没有类型限制，任何类型都可以。如果类中未重载下标访问运算符，编译器将会给出其缺省定义，在其表达对象数组时使用。

```
class CharArray
{
public:
    CharArray(int size)
    : _size(size)
    , _array(new char[_size]())
    {}

    char & operator[](int idx)
    {
        if(idx >=0 && idx < _size) {
            return _array[idx];
        } else {
            static char nullchar = '\0';
            cerr << "下标越界！\n";
            return nullchar;
        }
    }

    const char & operator[](int idx) const
    {
        return _array[idx];
    }

    ~CharArray()
    {
        delete [] _array;
    }
private:
    int _size;
    char * _array;
```

```
};
```

我们之前使用过的 `std::string` 同样也重载了下标访问运算符，这也是为什么它能像数组一样去访问元素的原因。

成员访问运算符

成员访问运算符包括箭头访问运算符 `->` 和解引用运算符 `*`，我们先来看箭头运算符 `->`.

箭头运算符只能以成员函数的形式重载，其返回值必须是一个指针或者重载了箭头运算符的对象。来看下例子：

```
class Data {
public:
    int getData() { return _data;   }
private:
    int _data;
};

class MiddleLayer
{
public:
    MiddleLayer(Data * pdata)
    : _pdata(pdata)
    {}

    //返回值是一个指针
    Data * operator->()
    {   return _pdata;  }

    Data & operator*()
    {   return *_pdata; }

    ~MiddleLayer()
    {   delete _data;   }
private:
    Data * _pdata;
};

class ThirdLayer
{
public:
    ThirdLayer(MiddleLayer * ml)
    : _ml(ml)
    {}
    //返回一个重载了箭头运算符的对象
    MiddleLayer & operator->()
    {   return *_ml;    }

    ~ThirdLayer()
    {   delete _ml; }

private:
    MiddleLayer * _ml;
};

void test()
```

```
{
    MiddleLayer ml(new Data());
    cout << ml->getData() << endl;
    cout << (ml.operator->())->getData() << endl;

    cout << (*ml).getData() << endl;

    ThirdLayer tl(new MiddleLayer(new Data()));
    cout << tl->getData() << endl;
    cout << ((tl.operator->()).operator->())->getData() << endl;
}
```

输入输出流运算符

在之前的例子中，我们如果想打印一个对象时，常用的方法是通过定义一个`print`成员函数来完成，但使用起来不太方便。我们希望打印一个对象，与打印一个整型数据在形式上没有差别(如下例子)，那就必须要重载`<<`运算符。

```
void test()
{
    int a = 1, b = 2;
    cout << a << b << endl;
    Point pt1(1, 2), pt2(3, 4);
    cout << pt1 << pt2 << endl;
}
```

从上面的形式能看出，`cout`是左操作数，`a`或者`pt1`是右操作数，那输入输出流能重载为成员函数形式吗？我们假设是可以的，由于非静态成员函数的第一个参数是隐含的this指针，代表当前对象本身，这与其要求是冲突的，因此`>>`和`<<`不能重载为成员函数，只能是非成员函数，如果涉及到要对类中私有成员进行访问，还得将非成员函数设置为类的友元函数。

```
class Point {
public:
    //...
    friend std::ostream & operator<<(std::ostream & os, const Point & rhs);
    friend std::istream & operator>>(std::istream & is, Point & rhs);
private:
    int _x;
    int _y;
};

std::ostream & operator<<(std::ostream & os, const Point & rhs)
{
    os << "(" << rhs._x
       << "," << rhs._y
       << ")";
    return os;
}

istd::stream & operator>>(std::istream & is, Point & rhs)
{
    is >> rhs._x;
    is >> rhs._y;
    return is;
```

```
}
```

通常来说，重载输出流运算符用得更多一些。同样的，输入流运算符也可以进行重载，如上。

总结：对于运算符重载时采用的形式的建议：

- 所有的一元运算符，建议以成员函数重载
- 运算符 = () [] -> ->* ，必须以成员函数重载
- 运算符 += -= /= *= %= ^= &= != >>= <<= 建议以成员函数形式重载
- 其它二元运算符，建议以非成员函数重载

类型转换

前面介绍过对普通变量的类型转换，比如说`int`型转换为`long`型，`double`型转换为`int`型，接下来我们要讨论下类对象与其他类型的转换。转换的方向有:

- 由其他类型向自定义类型转换
- 由自定义类型向其他类型转换

由其他类型向自定义类型转换

由其他类型向定义类型转换是由构造函数来实现的，只有当类中定义了合适的构造函数时，转换才能通过。这种转换，一般称为**隐式转换**。下面，我们通过一个例子进行说明:

```
class Point {
public:
    Point(int x = 0, int y = 0)
    : _x(x)
    , _y(y)
    {}

    //...
    friend std::ostream & operator<<(std::ostream & os, const Point & rhs);
private:
    int _x;
    int _y;
};

std::ostream & operator<<(std::ostream & os, const Point & rhs)
{
    os << "(" << rhs._x
       << "," << rhs._y
       << ")";
    return os;
}

void test()
{
    Point pt = 1;//隐式转换
    cout << "pt = " << pt << endl;
}
```

这种隐式转换有时候用起来是挺好的，比如，我们以前学过的`std::string`，当执行

```
std::string s1 = "hello,world";
```

该语句时，这里其实是有隐式转换的，但该隐式转换的执行很自然，很和谐。而上面把一个 `int` 型数据直接赋值给一个 `Point` 对象，看起来就是比较诡异的，难以接受，所以这里我们是不希望发生这样的隐式转换的。那怎么禁止隐式转换呢，比较简单，只需要在相应构造函数前面加上 `explicit` 关键字就能解决。

由自定义类型向其他类型转换

由自定义类型向其他类型的转换是由**类型转换函数**完成的，这是一个特殊的成员函数。它的形式如下：

```
operator 目标类型()
{
    //...
}
```

类型转换函数具有以下的特征:

- 必须是成员函数;
- 参数列表中没有参数;
- 没有返回值，但在函数体内必须以 `return` 语句返回一个目标类型的变量。

我们来看一个例子:

```
class Fraction
{
public:
    Fraction(double numerator, double denominator)
    : _numerator(numerator)
    , _denominator(denominator)
    {}

    operator double()
    {   return _numerator / _denominator;   }

    operator Point()
    {   return Point(_numerator, _denominator); }
private:
    double _numerator;
    double _denominator;
};

void test()
{
    Fraction f(2, 4);
    cout << "f = " << f << endl;
    double x = f + 1.11;
    cout << "x = " << x << endl;

    double y = f;
}
```

类作用域

作用域可以分为**类作用域**、**类名的作用域**以及**对象的作用域**几部分内容。在类中定义的成员变量和成员函数的作用域是整个类，这些名称只有在类中（包含类的定义部分和类外函数实现部分）是可见的，在类外是不可见的，因此，可以在不同类中使用相同的成员名。另外，类作用域意味着不能从外部直接访问类的任何成员，即使该成员的访问权限是`public`，也要通过对象名来调用，对于`static`成员函数，要指定类名来调用。

如果发生“屏蔽”现象，类成员的可见域将小于作用域，但此时可借助this指针或“类名::”形式指明所访问的是类成员，这有些类似于使用“`::`”访问全局变量。例如:

```
#include <iostream>

int num = 1;
int x = 2;
class Example {
public:
    void print(int num)
    {
        cout << "形参num = " << num << endl;
        cout << "数据成员num = " << this->num << endl;
        cout << "数据成员num = " << Example::num << endl;
        cout << "全局变量num = " << ::num << endl;
    }
private:
    int num;
};

```

和函数一样，类的定义没有生存期的概念，但类定义有作用域和可见域。使用类名创建对象时，首要的前提是类名可见，类名是否可见取决于类定义的可见域，该可见域同样包含在其作用域中，类本身可被定义在3种作用域内，这也是类定义的作用域。

全局作用域

在函数和其他类定义的外部定义的类称为全局类，绝大多数的`C++`类是定义在该作用域中，我们在前面定义的所有类都是在全局作用域中，全局类具有全局作用域。

类作用域

一个类可以定义在另一类的定义中，这是所谓**嵌套类**或者**内部类**，举例来说，如果类`A`定义在类`B`中，如果`A`的访问权限是`public`，则`A`的作用域可认为和`B`的作用域相同，不同之处在于必须使用`B::A`的形式访问`A`的类名。当然，如果`A`的访问权限是`private`，则只能在类内使用类名创建该类的对象，无法在外部创建`A`类的对象。

```
class Line {
public:
    Line(int x1, int y1, int x2, int y2);
    void printLine() const;
private:
    class Point {
    public:
        Point(int x = 0, int y = 0)
        : _x(x), _y(y)
        {}

        void print() const;
    private:
        int _x;
        int _y;
    };

    Point _pt1;
    Point _pt2;
};

Line::Line(int x1, int y1, int x2, int y2)
: _pt1(x1, y1), _pt2(x2, y2)
{}

void Line::printLine() const
{
    _pt1.print();
    cout << " ---> ";
    _pt2.print();
    cout << endl;
}

void Line::Point::print() const
{
    cout << "(" << _x
```

```
        << "," << _y
        << ")";
}

```

设计模式之Pimpl

`PIMPL`（`Private Implementation`或`Pointer to Implementation`）是通过一个私有的成员指针，将指针所指向的类的内部实现数据进行隐藏。`PIMPL`又称作“编译防火墙”，它的实现中就用到了嵌套类。`PIMPL`设计模式有如下优点：

1. 提高编译速度；
2. 实现信息隐藏；
3. 减小编译依赖，可以用最小的代价平滑的升级库文件；
4. 接口与实现进行解耦；
5. 移动语义友好。

```
//Line.h
#pragma once
class Line
{
public:
    Line(int,int,int,int);
    ~Line();
    void printLine() const;
private:
    class LineImpl;//类的前向声明
    LineImple * _pimpl;
};

//Line.cc
#include "Line.h"

class Line::LineImpl{
public:
    LineImpl(int x1, int y1, int x2, int y2);
    void printLine() const;
private:
    class Point {
    public:
        Point(int x = 0, int y = 0)
        : _x(x), _y(y)
        {}

```

```
        void print() const;
    private:
        int _x;
        int _y;
    };

    Point _pt1;
    Point _pt2;
};

Line::LineImpl::LineImpl(int x1, int y1, int x2, int y2)
: _pt1(x1, y1)
, _pt2(x2, y2)
{}

void Line::LineImpl::printLine() const
{
    _pt1.print();
    cout << " ---> ";
    _pt2.print();
    cout << endl;
}

Line::Line(int x1, int y1, int x2, int y2)
: _pimpl(new LineImpl(x1, y1, x2, y2))
{}

Line::~Line() { delete _pimpl;  }

void Line::printLine() const
{
    _pimpl->printLine();
}
```

单例模式的自动释放

在类和对象那一章，我们讲过单例模式，其中对象是由`_pInstance`指针来保存的，而在使用单例设计模式的过程中，也难免会遇到内存泄漏的问题。那么是否有一个方法，可以让对象自动释放，而不需要程序员自己手动去释放呢？在学习了嵌套类之后，我们就可以完美的解决这一问题。

在涉及到自动的问题时，我们很自然的可以想到：当对象被销毁时，会自动调用其析构函数。利用这一特性，我们可以解决这一问题。

```
class Singleton {
public:
    static Singleton * getInstance()
    {
        if(_pInstance == nullptr) {
            _pInstance = new Singleton();
        }
        return _pInstance;
    }

private:
    class AutoRelease {
    public:
        AutoRelease() {}

        ~AutoReleas() {
            if(_pInstance) {
                delete _pInstance;
            }
        }
    };
private:
    Singleton() {}
    ~Singleton() {}

private:
    static Singleton * _pInstance;
    static Singleton _auto;
};
```

块作用域

类的定义在代码块中，这是所谓**局部类**，该类完全被块包含，其作用域仅仅限于定义所在块，不能在块外使用类名声明该类的对象。

```
void test() {
    class Point {
    public:
        Point(int x, int y)
        : _x(x), _y(y)
        {}
        void print() const
```

```
8           {
9               cout << "(" << _x
10                   << "," << _y
11                   << ")" << endl;
12          }
13      private:
14          int _x;
15          int _y;
16      };
17
18      Point pt(1, 2);
19      pt.print();
20  }
```

标准库中string的底层实现方式

我们都知道， std::string的一些基本功能和用法了，但它底层到底是如何实现的呢? 其实在std::string的历史中，出现过几种不同的方式。下面我们来一一揭晓。

我们可以从一个简单的问题来探索，一个std::string对象占据的内存空间有多大，即sizeof(std::string)的值为多大？如果我们在不同的编译器（VC++, GNU, Clang++）上去测试，可能会发现其值并不相同；即使是GNU，不同的版本，获取的值也是不同的。

虽然历史上的实现有多种，但基本上有三种方式：

- Eager Copy(深拷贝)
- COW（Copy-On-Write 写时复制）
- SSO(Short String Optimization-短字符串优化)

每种实现，std::string都包含了下面的信息：

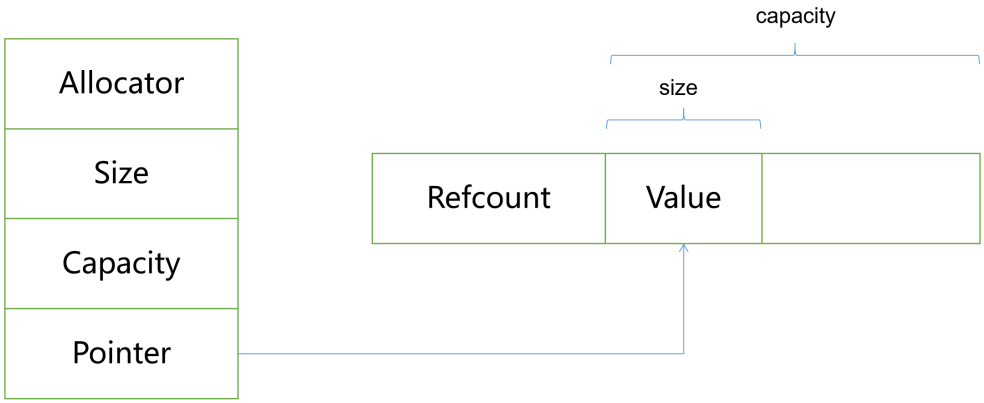
- 字符串的大小
- 能够容纳的字符数量
- 字符串内容本身

Eager Copy

最简单的就是深拷贝了。无论什么情况，都是采用拷贝字符串内容的方式解决，这也是我们之前已经实现过的方式。这种实现方式，在需要对字符串进行频繁复制而又并不改变字符串内容时，效率比较低下。所以需要对其实现进行优化，之后便出现了下面的COW的实现方式。

COW(Copy-On-Write)

当两个std::string发生复制构造或者赋值时，不会复制字符串内容，而是增加一个引用计数，然后字符串指针进行浅拷贝，其执行效率为O(1)。只有当需要修改其中一个字符串内容时，才执行真正的复制。其实现的示意图，有下面两种形式：

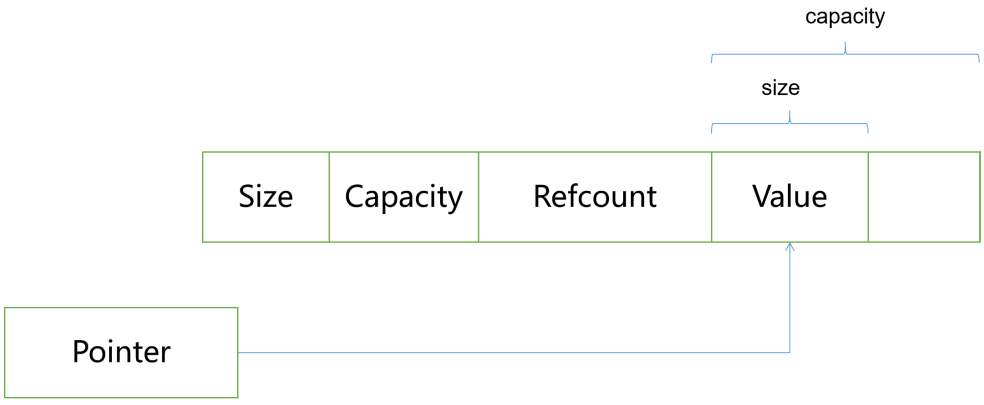


实现1

std::string的数据成员就是:

```
class string {
private:
    Allocator _allocator;
    size_t size;
    size_t capacity;
    char * pointer;
};
```

第二种形式为:



实现2

std::string的数据成员就只有一个了:

```
class string {
private:
    char * _pointer;
};
```

为了实现的简单，在GNU4.8.4的中，采用的是实现2的形式。从上面的实现，我们看到引用计数并没有与std::string的数据成员放在一起，为什么呢？大家可以思考一下。

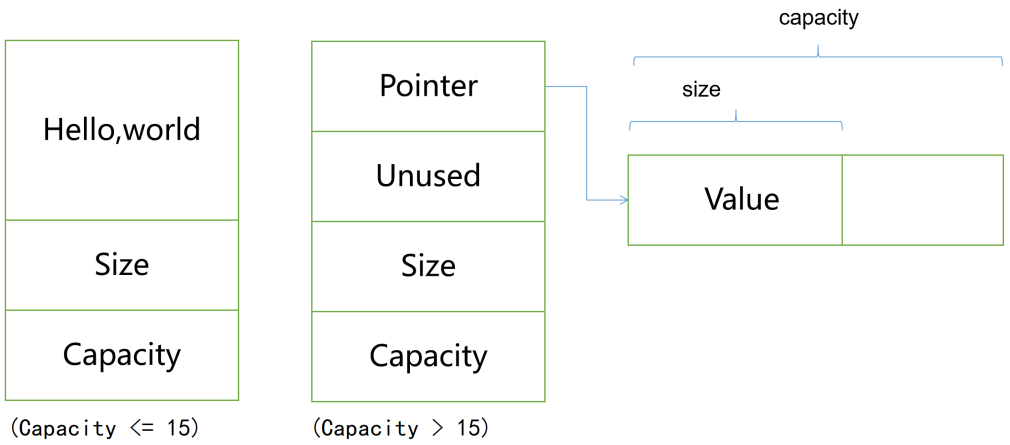
当执行复制构造或赋值时，引用计数加1，std::string对象共享字符串内容；当std::string对象销毁时，并不直接释放字符串所在的空间，而是先将引用计数减1，直到引用计数为0时，则真正释放字符串内容所在的空间。根据这个思路，大家可以自己动手实现一下。

大家再思考一下，既然涉及到了引用计数，那么在多线程环境下，涉及到修改引用计数的操作，是否是线程安全的呢？为了解决这个问题，GNU4.8.4的实现中，采用了原子操作。

SS0（Short String Optimization）

目前，在VC++、GNU5.x.x以上、Clang++上，std::string实现均采用了SSO的实现。

通常来说，一个程序里用到的字符串大部分都很短小，而在64位机器上，一个char*指针就占用了8个字节，所以SSO就出现了，其核心思想是：发生拷贝时要复制一个指针，对小字符串来说，为啥不直接复制整个字符串呢，说不定还没有复制一个指针的代价大。其实现示意图如下：



实现3

std::string的数据成员就是：

```
class string {
    union Buffer{
        char * _pointer;
        char _local[16];
    };

    Buffer _buffer;
    size_t _size;
    size_t _capacity;
};
```

当字符串的长度小于等于15个字节时，buffer直接存放整个字符串；当字符串大于15个字节时，buffer存放的就是一个指针，指向堆空间的区域。这样做的好处是，当字符串较小时，直接拷贝字符串，放在string内部，不用获取堆空间，开销小。

最佳策略

以上三种方式，都不能解决所有可能遇到的字符串的情况，各有所长，又各有缺陷。综合考虑所有情况之后，facebook开源的folly库中，实现了一个fbstring, 它根据字符串的不同长度使用不同的拷贝策略， 最终每个fbstring对象占据的空间大小都是24字节。

1. 很短的（0~22）字符串用SSO，23字节表示字符串（包括'\0'）,1字节表示长度
2. 中等长度的（23~255）字符串用eager copy，8字节字符串指针，8字节size，8字节capacity.
3. 很长的(大于255)字符串用COW, 8字节指针（字符串和引用计数），8字节size，8字节capacity.

线程安全性

两个线程同时对同一个字符串进行操作的话, 是不可能线程安全的, 出于性能考虑, C++并没有为string实现线程安全, 毕竟不是所有程序都要用到多线程。

但是两个线程同时对独立的两个string操作时, 必须是安全的. COW技术实现这一点是通过原子的对引用计数进行+1或-1操作。

CPU的原子操作虽然比mutex锁好多了, 但是仍然会带来性能损失, 原因如下:

- 阻止了CPU的乱性执行.
- 两个CPU对同一个地址进行原子操作, 会导致cache失效, 从而重新从内存中读数据.
- 系统通常会lock住比目标地址更大的一片区域，影响逻辑上不相关的地址访问

这也是在多核时代，各大编译器厂商都选择了SS0实现的原因吧。

面向对象之继承

在学习类和对象时，我们知道对象是基本，我们从对象上抽象出类。但是，世界可并不是一层对象一层类那么简单，对象抽象出类，在类的基础上可以再进行抽象，抽象出更高层次的类。所以经过抽象的对象论世界，形成了一个树状结构。(动物分类图 金融类 家用电器分类图 家族图谱 --- 泛化)

对象论的世界观认为，世界的基本元素是对象，我们将抽象思维作用于对象，形成了类的概念，而抽象的层次性形成了**抽象层次树**的概念。而抽象层次树为我们衍生出继承的概念，提供了依据。

我们需要继承这个概念，本质上是因为对象论中世界的运作往往是在某一抽象层次上进行的，而不是在最低的基本对象层次上。举个例子，某人发烧了，对其他人说：“我生病了，要去医院看医生。”这句简短的话中有一个代词“我”和三个名词“**病**”、“**医院**”、“**医生**”。这四个具有名词性的词语中，除了“我”是运作在世界的最底层——基本对象层外，其他三个都运作在抽象层次，在这个语境中，“**病**”、“**医院**”、“**医生**”都是抽象的。但是，本质上他确实是生了一个具体的病，要去一个具体的医院看一个具体的医生，那么在哲学上要如何映射这种抽象和具体呢？就是靠继承， 拿医生来说吧，所有继承自“医生”类的类所指的所有具体对象都可以替换掉这里具体的医生，这都不影响这句话语义的正确性。

回到编程语言层面，在`C`语言中重用代码的方式就是拷贝代码、修改代码。 C++ 中代码重用的方式之一就是采用继承。继承是面向对象程序设计中重要的特征，可以说，不掌握继承就等于没有掌握面向对象的精华。**通过继承，我们可以用原有类型来定义一个新类型，定义的新类型既包含了原有类型的成员，也能自己添加新的成员，而不用将原有类的内容重新书写一遍。**原有类型称为“基类”或“父类”，在它的基础上建立的类称为“派生类”或“子类”。

继承的定义

当一个派生类继承一个基类时，需要在派生类的**类派生列表**中明确的指出它是从哪个基类继承而来的。类派生列表的形式是在类名之后，大括号之前用冒号分隔，后面紧跟以逗号分隔的基类列表，其中每个基类前面可以有访问修饰限定符，其形式如下:

```
class 派生类
: public/protected/private 基类
{};
```

派生类的生成过程包含3个步骤:

1. 吸收基类的成员
2. 改造基类的成员
3. 添加自己新的成员

```
class Point3D
: public Point
{
public:
    Point3D(int x, int y, int z)
    : Point(x, y)
    , _z(z)
    {
        cout << "Point3D(int,int,int)" << endl;
    }

    void display() const
    {
        print();
        cout << _z << endl;
    }
private:
    int _z;
};
```

继承的局限

不论何种继承方式，下面这些基类的特征是不能从基类继承下来的：

- 构造函数
- 析构函数
- 用户重载的operator new/delete运算符
- 用户重载的operator=运算符
- 友元关系

派生方式对基类成员的访问权限

派生类继承了基类的全部成员变量和成员方法（除了构造和析构之外的成员方法），但是这些成员的访问属性，在派生过程中是可以调整的。

派生（继承）方式有3种，分别是

- public（公有）继承
- protected（保护型）继承
- private（私有）继承

继承方式	基类成员访问权限	在派生类中访问权限	派生类对象访问
公有继承	public protected private	public protected 不可直接访问	可直接访问 不可直接访问 不可直接访问
保护继承	public protected private	protected protected 不可直接访问	不可直接访问
私有继承	public protected private	private private 不可直接访问	不可直接访问

通过继承，除了基类私有成员以外的其它所有数据成员和成员函数，派生类中可以直接访问。 private成员是私有成员，只能被本类的成员函数所访问，派生类和类外都不能访问。 public成员是公有成员，在本类、派生类和外部都可访问。 protected成员是保护成员，只能在本类和派生类中访问，是一种区分血缘关系内外有别的成员。

总结：派生类的访问权限规则如下： 1．不管是什么继承方式，派生类中都不能访问基类的私有成员。 2．派生类内部除了父类的私有成员不可以访问外，其他的都可以访问。 3．**派生类对象**除了公有继承基类中的公有成员可以访问外，其他的一律不能访问 。

4.

派生类对象的构造

我们知道，构造函数和析构函数是不能继承的，为了对数据成员进行初始化，**派生类必须重新定义构造函数和析构函数。**由于派生类对象通过继承而包含了基类数据成员，因此，创建派生类对象时，系统首先通过派生类的构造函数来调用基类的构造函数，完成基类成员的初始化，而后对派生类中新增的成员进行初始化。

派生类构造函数的一般格式为：

```
派生类名(总参数表): 基类构造函数(参数表)
{
    //函数体
};
```

对于派生类对象的构造，我们分下面4种情况进行讨论：

1. 如果派生类有显式定义构造函数，而基类没有，则创建派生类的对象时，派生类相应的构造函数会被自动调用，此时都自动调用了基类缺省的无参构造函数。

```
class Base {
pulbic:
    Base(){ cout << "Base()" << endl;   }
};

class Derived
: public Base {
public:
    Derived(long derived)
    : _derived(derived)
    {   cout << "Derived(long)" << endl;    }

    long _derived;
};

void test() {
    Derived d(1);
}
```

2. 如果派生类没有显式定义构造函数而基类有，则基类必须拥有默认构造函数。

```
class Base {
pulbic:
    Base(long base){    cout << "Base(long)" << endl;   }
private:
    long _base;
};

class Derived
: public Base {
public:
};

void test() {
```

```
    Derived d;//error
}
```

3. 如果派生类有构造函数，基类有默认构造函数，则创建派生类的对象时，基类的默认构造函数会自动调用，如果你想调用基类的有参构造函数，必须要在派生类构造函数的初始化列表中显示调用基类的有参构造函数。
4. 如果派生类和基类都有构造函数，但基类没有默认的无参构造函数，即基类的构造函数均带有参数，则派生类的每一个构造函数必须在其初始化列表中显示的去调用基类的某个带参的构造函数。如果派生类的初始化列表中没有显示调用则会出错，因为基类中没有默认的构造函数。

```
class Base {
pulbic:
    Base(long base) {   cout << "Base(long)" << endl;   }
private:
    long _base;
};

class Derived
: public Base {
public:
    Derived(long base, long derived)
    : Base(base)
    , _derived(derived)
    {   cout << "Derived(long,long)" << endl;   }

    long _derived;
};

void test() {
    Derived d(1, 2);
}
```

虽然上面细分了四种情况进行讨论，但不管如何，谨记一条: 必须将基类构造函数放在派生类构造函数的初试化列表中，以调用基类构造函数完成基类数据成员的初始化。派生类构造函数实现的功能，或者说调用顺序为：

1. 完成对象所占整块内存的开辟，由系统在调用构造函数时自动完成。
2. 调用基类的构造函数完成基类成员的初始化。
3. 若派生类中含对象成员、`const`成员或引用成员，则必须在初始化表中完成其初始化。
4. 派生类构造函数体执行。

派生类对象的销毁

当派生类对象被删除时，派生类的析构函数被执行。析构函数同样不能继承，因此，在执行派生类析构函数时，基类析构函数会被自动调用。执行顺序是先执行派生类的析构函数，再执行基类的析构函数，这和执行构造函数时的顺序正好相反。当考虑对象成员时，继承机制下析构函数的调用顺序：

1. 先调用派生类的析构函数
2. 再调用派生类中成员对象的析构函数
3. 最后调用普通基类的析构函数

多基继承（多基派生）

C++除了支持单根继承外，还支持多重继承。那为什么要引入多重继承呢？其实是因为在客观现实世界中，我们经常碰到一个人身兼数职的情况，如在学校里，一个同学可能既是一个班的班长，又是学生会中某个部门的部长；在创业公司中，某人既是软件研发部的CTO，又是财务部的CFO；一个人既是程序员，又是段子手。诸如此类的情况出现时，单一继承解决不了问题，就可以采用多基继承了。

多重继承的定义形式如下：

```
class 派生类
: public/protected/private 基类1
, ...
, public/protected/private 基类N
{};
```

下面我们举一个例子：

```
class Fairy {
public:
    void fly() {    cout << " can fly.\n";}
};

class Monstor {
public:
    void attack() { cout << " take an attack.\n";   }
};

class Monkey
: public Fairy
, public Monstor {
public:
    Monkey(const string & name)
    : _name(name)
    {}
```

```
18
19      void print() {
20          cout << _name << " ";
21      }
22  private:
23      string _name;
24  };
25
26  void test() {
27      Monkey sunWukong("Sun Wukong");
28      sunWukong.print();
29      sunWukong.fly();
30      sunWukong.attack();
31  }
```

多基继承的派生类对象的构造和销毁

多基派生时，派生类的构造函数格式如（假设有N个基类）：

```
派生类名(总参数表)
: 基类名1(参数表1)
, 基类名2(参数表2)
, ...
, 基类名N(参数表N)
{
    //函数体
}
```

和前面所讲的单基派生类似，总参数表中包含了后面各个基类构造函数需要的参数。

多基继承和单基继承的派生类构造函数完成的任务和执行顺序并没有本质不同，唯一一点区别在于：首先要执行所有基类的构造函数，再执行派生类构造函数中初始化表达式的其他内容和构造函数体。各基类构造函数的执行顺序与其在初始化表中的顺序无关，而是由定义派生类时指定的基类顺序决定的。

析构函数的执行顺序同样是与构造函数的执行顺序相反。

但在使用多基继承过程中，会产生两种二义性。

成员名冲突的二义性

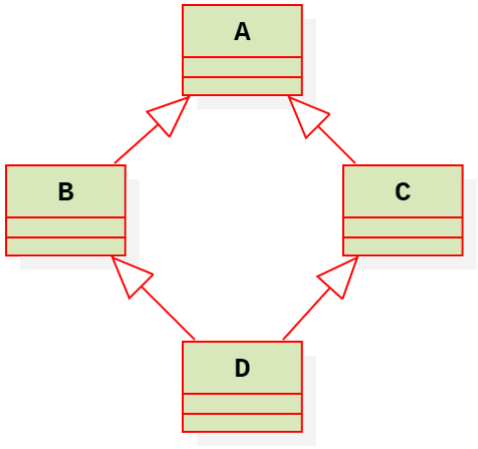
一般来说，在派生类中对基类成员的访问应当具有唯一性，但在多基继承时，如果多个基类中存在同名成员的情况，造成编译器无从判断具体要访问的哪个基类中的成员，则称为对基类成员访问的二义性问题。如下面的例子，我们先定义3个不同的类A、B、C，这3个类中都有一个同名成员函数print，然后让类D继承自A、B、C，则当创建D的对象d，用d调用成员函数print时，发生编译错误。

```
class A {
public:
    void print() {  cout << "A::print()" << endl;   }
};

class B {
public:
    void print() {  cout << "B::print()" << endl;   }
};

class C {
public:
    void print() {  cout << "C::print()" << endl;   }
};

class D
: public A
, public B
, public C
{};

void test() {
    D d;
    d.print();//error
    d.A::print();//ok
    d.B::print();//ok
    d.C::print();//ok
}
```

解决该问题的方式比较简单，只需要在调用时，指明要调用的是某个基类的成员函数即可，即使用作用域限定符就可以解决该问题。

菱形继承的二义性问题

而另外一个就是菱形继承的问题了。多基派生中，如果在多条继承路径上有一个共同的基类，如下图所示，不难看出，在D类对象中，会有来自两条不同路径的共同基类（类A）的双重拷贝。



出现这种问题时，我们的解决方案是采用虚拟继承。中间的类B、C虚拟继承自A，就可以解决了。至于背后到底发生了什么，待我们学了多态的知识后一起做讲解。

```
class A {
public:
    void setNumber(long number)
    {   _number = number;   }

private:
    long _number;
};

class B
: virtual public A
{};

class C
: virtual public A
{};

class D
: public B
, public C
{};

int main(void)
{
    D d;
    d.setNumber(10);
    return 0;
}
```

基类与派生类间的相互转换

“类型适应”是指两种类型之间的关系，说A类适应B类是指A类的对象能直接用于B类对象所能应用的场合，从这种意义上讲，**派生类适应于基类**，派生类的对象适应于基类对象，派生类对象的指针和引用也适应于基类对象的指针和引用。

- 可以把派生类的对象赋值给基类的对象
- 可以把派生类的对象赋值给基类的引用
- 可以声明基类的指针指向派生类的对象 (向上转型)

也就是说如果函数的形参是基类对象或者基类对象的引用或者基类对象的指针类型，在进行函数调用时，相应的实参可以是派生类对象。

```
class Base {
public:
    Base(long base) {   cout << "Base(long)" << endl;   }
private:
    long _base;
};

class Derived
: public Base {
public:
    Derived(long base, long derived)
    : Base(base)
    , _derived(derived)
    {   cout << "Derived(long,long)" << endl;   }
private:
    long _derived;
};

void test() {
    Base base(1);

    Derived derived(10, 11);

    base = derived;//ok

    Base & refBase = derived;//ok

    Base * pBase = &derived;//ok

    derived = base;//error

    Derived & refDerived = base;//error
```

```
33
34      Derived * pDerived = &base;//error
35  }
```

派生类对象间的复制控制

从前面的知识，我们知道，基类的**拷贝构造函数**和**operator=运算符函数**不能被派生类继承，那么在执行派生类对象间的复制操作时，就需要注意以下几种情况:

1. 如果用户定义了基类的拷贝构造函数，而没有定义派生类的拷贝构造函数，那么在用一个派生类对象初始化新的派生类对象时，两对象间的派生类部分执行缺省的行为，而两对象间的基类部分执行用户定义的基类拷贝构造函数。
2. 如果用户重载了基类的赋值运算符函数，而没有重载派生类的赋值运算符函数，那么在用一个派生类对象给另一个已经存在的派生类对象赋值时，两对象间的派生类部分执行缺省的赋值行为，而两对象间的基类部分执行用户定义的重载赋值函数。
3. 如果用户定义了派生类的拷贝构造函数或者重载了派生类的对象赋值运算符=，则在用已有派生类对象初始化新的派生类对象时，或者在派生类对象间赋值时，将会执行用户定义的派生类的拷贝构造函数或者重载赋值函数，而不会再自动调用基类的拷贝构造函数和基类的重载对象赋值运算符，这时，通常需要用户在派生类的拷贝构造函数或者派生类的赋值函数中显式调用基类的拷贝构造或赋值运算符函数。

```
1   class Base {
2   public:
3       Base(const char * data)
4       : _data(new char[strlen(data) + 1]())
5       {
6           cout << "Base(const char * data)" << endl;
7           strcpy(_data, data);
8       }
9
10      Base(const Base & rhs)
11      : _data(new char[strlen(rhs._data) + 1]())
12      {
13          strcpy(_data, rhs._data);
14          cout << "Base(const Base & )" << endl;
15      }
16
17      Base & operator=(const Base & rhs)
18      {
19          cout << "Base & operator=(const Base &)" << endl;
```

```
20          if(this != &rhs) {
21              delete [] _data;
22
23              _data = new char[strlen(rhs._data) + 1]();
24              strcpy(_data, rhs._data);
25          }
26          return *this;
27      }
28
29      ~Base()
30      {
31          cout << "~Base()" << endl;
32          delete [] _data;
33      }
34
35      const char * data() const
36      {   return _data;   }
37  private:
38      char * _data;
39  };
40
41  class Derived
42  : public Base
43  {
44  public:
45      Derived(const char * data, const char * data2)
46      : Base(data)
47      , _data2(new char[strlen(data2) + 1]())
48      {
49          cout << "Derived(const char *, const char *)" <<
    endl;
50          strcpy(_data2, data2);
51      }
52
53      Derived(const Derived & rhs)
54      : Base(rhs)
55      , _data2(new char[strlen(rhs._data2) + 1]())
56      {
57          cout << "Derived(const Derived &)" << endl;
58          strcpy(_data2, rhs._data2);
59      }
60
61      Derived & operator=(const Derived & rhs)
62      {
```

```
63          cout << "Derived & operator=(const Derived&)" <<
endl;
64          if(this != & rhs) {
65              Base::operator=(rhs);//显式调用赋值运算符函数
66              delete [] _data2;
67
68              _data2 = new char[strlen(rhs._data2) + 1]();
69              strcpy(_data2, rhs._data2);
70          }
71          return *this;
72      }
73
74      ~Derived()
75      {   cout << "~Derived()" << endl;   }
76
77      friend std::ostream & operator<<(std::ostream & os,
const Derived & rhs);
78  private:
79      char * _data2;
80  };
81
82  std::ostream & operator<<(std::ostream & os, const Derived &
rhs)
83  {
84      os << rhs.data() << "," << rhs._data2;
85      return os;
86  }
87
88  int main(void)
89  {
90      Derived d1("hello", "world");
91      cout << "d1 = " << d1 << endl;
92
93      Derived d2 = d1;
94      cout << "d2 = " << d2 << endl;
95
96      Derived d3("guangdong", "shenzhen");
97      cout << "d3 = " << d3 << endl;
98
99      d3 = d1;
100     cout << "d3 = " << d3 << endl;
101
102     return 0;
103 }
```

多态

多态性（`polymorphism`）是面向对象设计语言的基本特征之一。仅仅是将数据和函数捆绑在一起，进行类的封装，使用一些简单的继承，还不能算是真正应用了面向对象的设计思想。多态性是面向对象的精髓。多态性可以简单地概括为“一个接口，多种方法”。

多态按字面意思就是多种形态。比如说：警车鸣笛，普通人反应一般，但逃犯听见会大惊失色，拔腿就跑。又比如说，....。

通常是指对于同一个消息、同一种调用，在不同的场合，不同的情况下，执行不同的行为 。

为什么要用多态?

我们知道，封装可以隐藏实现细节，使得代码模块化；继承可以扩展已存在的代码模块（类）。它们的目的都是为了代码重用。而多态除了代码的复用性外，还可以解决项目中紧偶合的问题，提高程序的可扩展性。

如果项目耦合度很高的情况下，维护代码时修改一个地方会牵连到很多地方，会无休止的增加开发成本。而降低耦合度，可以保证程序的扩展性。而多态对代码具有很好的可扩充性。增加新的子类不影响已存在类的多态性、继承性，以及其他特性的运行和操作。实际上新加子类更容易获得多态功能。例如，在实现了圆锥、半圆锥以及半球体的多态基础上，很容易增添球体类的多态性。

C++支持两种多态性：编译时多态和运行时多态。

编译时多态：也称为静态多态，我们之前学习过的**函数重载、运算符重载**就是采用的静态多态，C++编译器根据传递给函数的参数和函数名决定具体要使用哪一个函数，又称为先期联编（early binding）。

运行时多态：在一些场合下，编译器无法在编译过程中完成联编，必须在程序运行时完成选择，因此编译器必须提供这么一套称为“动态联编”（dynamic binding）的机制，也叫晚期联编（late binding）。C++通过虚函数来实现动态联编。

接下来，我们提到的多态，不做特殊说明，指的就是动态多态。

虚函数的定义

什么是虚函数呢？虚函数就是在基类中被声明为virtual，并在一个或多个派生类中被重新定义的成员函数。其形式如下:

```
// 类内部
class 类名 {
    virtual 返回类型 函数名(参数表)
    {
        //...
    }
};

//类之外
virtual 返回类型 类名::函数名(参数表)
{
    //...
}
```

如果一个基类的成员函数定义为虚函数，那么它在所有派生类中也保持为虚函数，即使在派生类中省略了virtual关键字，也仍然是虚函数。派生类要对虚函数进行中可根据需重定义，重定义的格式有一定的要求：

- 与基类的虚函数有相同的参数个数；
- 与基类的虚函数有相同的参数类型；
- 与基类的虚函数有相同的返回类型。

```
class Base {
public:
    virtual void display()
    {   cout << "Base::display()" << endl;  }

    virtual void print()
    {   cout << "Base::print()" << endl;    }
};

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

void test(Base * pbase)
{
    pbase->display();
}

int main()
{
```

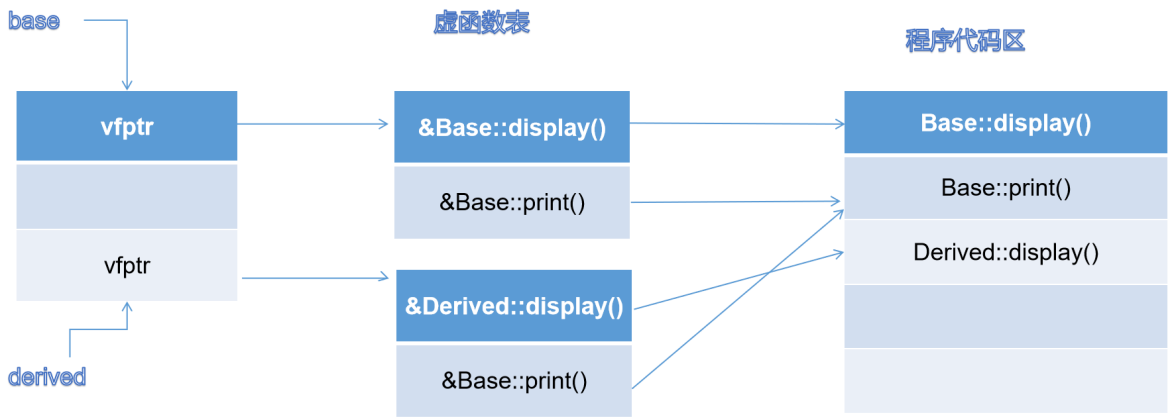
```
    Base base;
    Derived derived;
    test(&base);
    test(&derived);
    return 0;
}
```

上面的例子中，对于`test()`函数，如果不管测试的结果，从其实现来看，通过类`Base`的指针`pbase`只能调用到`Base`类型的`display`函数；但最终的结果是24行的`test`调用，最终会调用到`Derived`类的`display`函数，这里就体现出虚函数的作用了，这是怎么做到的呢，或者说虚函数底层是的怎么实现的呢？

虚函数的实现机制

虚函数的实现是怎样的呢？简单来说，就是通过一张**虚函数表**（`Virtual Fucntion Table`）实现的。具体地讲，当类中定义了一个虚函数后，会在该类创建的对象的存储布局的开始位置多一个虚函数指针（`vfptr`），该虚函数指针指向了一张虚函数表，而该虚函数表就像一个数组，表中存放的就是各虚函数的入口地址。如下图

当一个基类中设有虚函数，而一个派生类继承了该基类，并对虚函数进行了重定义，我们称之为**覆盖**(`override`). 这里的覆盖指的是派生类的虚函数表中相应虚函数的入口地址被覆盖。



虚函数机制是如何被激活的呢，或者说**动态多态是怎么表现出来的呢**？从上面的例子，可以得出结论：

1. 基类定义虚函数，派生类重定义（覆盖）虚函数
2. 创建派生类对象
3. 基类的指针指向派生类对象
4. 基类指针调用虚函数

哪些函数不能被设置为虚函数?

1. 普通函数（非成员函数）：定义虚函数的主要目的是为了重写达到多态，所以普通函数声明为虚函数没有意义，因此编译器在编译时就绑定了它。
2. 静态成员函数：静态成员函数对于每个类都只有一份代码，所有对象都可以共享这份代码，他不归某一个对象所有，所以它也没有动态绑定的必要。
3. 内联成员函数：内联函数本就是为了减少函数调用的代价，所以在代码中直接展开。但虚函数一定要创建虚函数表，这两者不可能统一。另外，内联函数在编译时被展开，而虚函数在运行时才动态绑定。
4. 构造函数：这个原因很简单，主要从语义上考虑。因为构造函数本来是为了初始化对象成员才产生的，然而虚函数的目的是为了在完全不了解细节的情况下也能正确处理对象，两者根本不能“ 好好相处 ”。因为虚函数要对不同类型的对象产生不同的动作，如果将构造函数定义成虚函数，那么对象都没有产生，怎么完成想要的动作呢
5. 友元函数：当我们把一个函数声明为一个类的友元函数时，它只是一个可以访问类内成员的普通函数，并不是这个类的成员函数，自然也不能在自己的类内将它声明为虚函数。

虚函数的访问

指针访问

使用指针访问非虚函数时，编译器根据指针本身的类型决定要调用哪个函数，而不是根据指针指向的对象类型； 使用指针访问虚函数时，编译器根据指针所指对象的类型决定要调用哪个函数(**动态联编**)，而与指针本身的类型无关。

引用访问

使用引用访问虚函数，与使用指针访问虚函数类似，**表现出动态多态**特性。不同的是，引用一经声明后，引用变量本身无论如何改变，其调用的函数就不会再改变，始终指向其开始定义时的函数。因此在使用上有一定限制，但这在一定程度上提高了代码的安全性，特别体现在函数参数传递等场合中，可以将引用理解成一种“受限制的指针”。

对象访问

和普通函数一样，虚函数一样可以通过对象名来调用，此时编译器采用的是**静态联编**。通过对象名访问虚函数时，调用哪个类的函数取决于定义对象名的类型。对象类型是基类时，就调用基类的函数；对象类型是子类时，就调用子类的函数。

成员函数中访问

在类内的成员函数中访问该类层次中的虚函数，采用**动态联编**，要使用`this`指针。

构造函数和析构函数中访问

构造函数和析构函数是特殊的成员函数，在其中访问虚函数时，C++采用**静态联编**，即在构造函数或析构函数内，即使是使用“this->虚函数名”的形式来调用，编译器仍将其解释为静态联编的“本类名::虚函数名”。即它们所调用的虚函数是自己类中定义的函数，如果在自己的类中没有实现该函数，则调用的是基类中的虚函数。但绝不会调用任何在派生类中重定义的虚函数。

```
class Grandpa {
public:
    Grandpa() { cout << "Grandpa()" << endl;    }
    ~Grandpa() {    cout << "~Grandpa()" << endl;   }

    virtual
    void func1() {  cout << "Grandpa::func1()" << endl; }
    virtual
    void func2() {  cout << "Grandpa::func2()" << endl; }
};

class Father
: public Grandpa {
public:
    Father() {
        cout << "Father()" << endl;
        func1();
    }
    ~Father() {
        cout << "~Father()" << endl;
        func2();
    }

    virtual
    void func1() {  cout << "Father::func1()" << endl;  }
    virtual
    void func2() {  cout << "Father::func2()" << endl;  }
};

class Son
: public Father {
public:
    Son() { cout << "Son()" << endl;    }
    ~Son() {    cout << "~Son()" << endl;   }

    virtual void func1() {  cout << "Son::func1()" << endl; }
```

```
    virtual void func2() {  cout << "Son::func2()" << endl; }
};
```

接下来，我们看一个练习，大家思考一下，会输出什么呢?

```
class A {
public:
    virtual
    void func(int val = 1) {    cout << "A->" << val << endl;}

    virtual void test() {   func();     }
private:
    long _a;
};

class B
: public A {
public:
    virtual
    void func(int val = 10) { cout << "B->" << val << endl; }
private:
    long _b;
};

int main(void)
{
    B b;
    A * p1 = (A*)&b;
    B * p2 = &b;
    p1->func();
    p2->func();
    return 0;
}
```

纯虚函数

纯虚函数是一种特殊的虚函数，在许多情况下，在基类中不能对虚函数给出有意义的实现，而把它声明为纯虚函数，**它的实现留给该基类的派生类去做**。这就是纯虚函数的作用。纯虚函数的格式如下：

```
class 类名 {
public:
    virtual 返回类型 函数名(参数包) = 0;
};
```

设置纯虚函数的意义，就是让所有的类对象（主要是派生类对象）都可以执行纯虚函数的动作，但类无法为纯虚函数提供一个合理的缺省实现。所以类纯虚函数的声明就是在告诉子类的设计者，“**你必须提供一个纯虚函数的实现，但我不知道你会怎样实现它**”。

```
class Base {
public:
    virtual void display()=0;
};

class Child
: public Base {
public:
    virtual void display()
    {   cout << "Child::display()" << endl; }
};
```

声明纯虚函数的目的在于，提供一个与派生类一致的接口。

```
class Figure {
public:
    void display() const =0;
    double area() const =0;
};

class Circle
: public Figure {
public:
    explicit Circle(double radius)
    : _radius(radius) {}

    void display() const {  cout << "Circle";   }

    double area() const
    {   return 3.14159 * _radius * _radius; }
private:
    double _radius;
};

```

```
class Rectangle
: public Figure {
public:
    Rectangle(double length, double width)
    : _length(length)
    , _width(width)
    {}

    void display() const {  cout << "Rectangle";    }

    double area() const {   return _length * _width;    }
private:
    double _length;
    double _width;
};

class Triangle
: public Figure {
public:
    Triangle(double a, double b, double c)
    : _a(a)
    , _b(b)
    , _c(c)
    {}

    void display() const {  cout << "Triangle"; }

    double area() const {
        double p = (_a + _b + _c) / 2;
        return  sqrt(p * (p - _a) * (p - _b) * (p - _c));
    }
private:
    double _a;
    double _b;
    double _c;
};
```

抽象类

一个类可以包含多个纯虚函数。只要类中含有一个纯虚函数，该类便为抽象类。一个抽象类只能作为基类来派生新类，不能创建抽象类的对象。

和普通的虚函数不同，在派生类中一般要对基类中纯虚函数进行重定义。如果该派生类没有对所有的纯虚函数进行重定义，则该派生类也会成为抽象类。这说明只有在派生类中给出了基类中所有纯虚函数的实现时，该派生类便不再是抽象类。

除此以外，还有另外一种形式的抽象类。对一个类来说，如果只定义了protected型的构造函数而没有提供public构造函数，无论是在外部还是在派生类中作为其对象成员都不能创建该类的对象，但可以由其派生出新的类，这种能派生新类，却不能创建自己对象的类是另一种形式的抽象类。

```
1  class Base {
2  protected:
3      Base(int base): _base(base) {   cout << "Base()" << endl;
   }
4  private:
5      int _base;
6  };
7
8  class Derived
9  : public Base {
10 public:
11     Derived(int base, int derived)
12     : Base(base)
13     , _derived(derived)
14     {   cout << "Derived(int,int)" << endl; }
15
16     void print() const
17     {
18         cout << "_base:" << _base
19              << ", _derived:" << _derived << endl;
20     }
21 private:
22     int _derived;
23 };
24
25 void test()
26 {
27     Base base(1);//error
28     Derived derived(1, 2);
29 }
```

虚析构函数

虽然构造函数不能被定义成虚函数，但析构函数可以定义为虚函数，一般来说，如果类中定义了虚函数，析构函数也应被定义为虚析构函数，尤其是类内有申请的动态内存，需要清理和释放的时候。

```
class Base {
public:
    Base(const char * pstr)
    : _pstr(new char[strlen(pstr) + 1]())
    {   cout << "Base(const char *)" << endl;   }

    ~Base() {
        delete [] _pstr;
        cout << "~Base()" << endl;
    }

private:
    char * _pstr;
};

class Derived
: public Base {
public:
    Derived(const char * pstr, const char * pstr2)
    : Base(pstr)
    , _pstr2(new char[strlen(pstr2) + 1]())
    {   cout << "Derived(const char*, const char*)" << endl;
}

    ~Derived() {
        delete [] _pstr2;
        cout << "~Derived()" << endl;
    }
};

void test()
{
    Base * pbase = new Derived("hello", "wuhan");
    delete pbase;
}
```

如上，在例子中，如果基类 Base 的析构函数没有设置为虚函数，则在执行 delete pbase;语句时，不会调用派生类 Derived 的析构函数，这样就会造成内存泄漏。此时，将基类Base的析构函数设置为虚函数，就可以解决该问题。

如果有一个基类的指针指向派生类的对象，并且想通过该指针`delete`派生类对象，系统将只会执行基类的析构函数，而不会执行派生类的析构函数。为避免这种情况的发生，往往把基类的析构函数声明为虚的，此时，系统将先执行派生类对象的析构函数，然后再执行基类的析构函数。

如果基类的析构函数声明为虚的，派生类的析构函数也将自动成为虚析构函数，无论派生类析构函数声明中是否加virtual关键字。

重载、隐藏、覆盖

重载：发生在同一个类中，函数名称相同，但参数的类型、个数、顺序不同。

覆盖：发生在父子类中，同名虚函数，参数亦完全相同。（同名数据成员也有隐藏）

隐藏：发生在父子类中，指的是在某些情况下，派生类中的函数屏蔽了基类中的同名函数。

```
//针对于隐藏的例子
class Base {
public:
    Base(int m)
    : _member(m)
    {   cout << "Base(int)" << endl;    }

    void func(int x)
    {   cout << "Base::func(int)" << endl;  }

    ~Base() {   cout << "~Base()" << endl;  }
protected:
    int _member;
};

class Derived
: public Base {
public:
    Derived(int m1, int m2)
    : Base(m1)
    , _memeber(m2)
    {   cout << "Derived(int, int)" << endl;    }

    void func(int *)
    {
        cout << "_member: " << _member << endl;
        cout << "Derived::func(int*)" << endl;
```

```
    }

    ~Derived() {    cout << "~Derived()" << endl;   }
private:
    int _member;
};
```

测试虚表的存在

从前面的知识讲解，我们已经知道虚表的存在，但之前都是理论的说法，我们是否可以通过程序来验证呢？答案是肯定的。接下来我们看看下面的例子：

```
class Base {
public:
    Base(long data1): _data1(data1) {}

    virtual
    void func1() {  cout << "Base::func1()" << endl;    }
    virtual
    void func2() {  cout << "Base::func2()" << endl;    }
    virtual
    void func3() {  cout << "Base::func3()" << endl;    }
private:
    long _data1;
};

class Derived
: public Base {
public:
    Derived(long data1, long data2): _data1(data1),
_data2(data2) {}

    virtual void func1() {  cout << "Derived::func1()" <<
endl; }
    virtual void func2() {  cout << "Derived::func2()" <<
endl; }
private:
    long _data2;
};

void test()
{
    Derived derived(10, 100);
    long ** pVtable = (long **)&derived;
```

```
30
31      typedef void(* Function)();
32      for(int idx = 0; idx < 3; ++idx) {
33          Function f = (Function)pVtable[0][idx];
34          f();
35      }
36  }
37
```

以上例子充分说明了虚表的存在。那虚表到底存在什么位置呢？大家可以思考一下。

之前我们的例子都比较简单，接下来我们看看相对复杂一些的例子

带虚函数的多基派生

```
1   class Base1 {
2   public:
3       Base1() : _iBase1(10) {}
4
5       virtual void f() {  cout << "Base1::f()" << endl;   }
6
7       virtual void g() {  cout << "Base1::g()" << endl;   }
8
9       virtual void h() {  cout << "Base1::h()" << endl;   }
10  private:
11      int _iBase1;
12  };
13
14  class Base2 {
15  public:
16      Base2() : _iBase2(100) {}
17
18      virtual void f() {  cout << "Base2::f()" << endl;   }
19
20      virtual void g() {  cout << "Base2::g()" << endl;   }
21
22      virtual void h() {  cout << "Base2::h()" << endl;   }
23  private:
24      int _iBase2;
25  };
26
27  class Base3 {
28  public:
29      Base3() : _iBase3(1000) {}
```

```
30
31      virtual void f() {  cout << "Base3::f()" << endl;   }
32
33      virtual void g() {  cout << "Base3::g()" << endl;   }
34
35      virtual void h() {  cout << "Base3::h()" << endl;   }
36  private:
37      int _iBase3;
38  };
39
40  class Derived
41  : public Base1
42  , public Base2
43  , public Base3 {
44  public:
45      Derived() : _iDerived(10000) {}
46
47      void f() {  cout << "Derived::f()" << endl; }
48
49      virtual void g1() { cout << "Derived::g1()" << endl;    }
50  private:
51      int _iDerived;
52  };
53
54  void test() {
55      Derived d;
56      Base2 * pBase2 = &d;
57      Base3 * pBase3 = &d;
58      Derived * pDerived = &d;
59
60      pBase2->f();
61      cout << "sizeof(d) = " << sizeof(d) << endl;
62
63      cout << "&Derived = " << &d << endl;
64      cout << "pBase2 = " << pBase2 << endl;
65      cout << "pBase3 = " << pBase3 << endl;
66  }
67
68  //结论：多重继承（带虚函数）
69  //1. 每个基类都有自己的虚函数表
70  //2. 派生类如果有自己的虚函数，会被加入到第一个虚函数表之中
71  //3. 内存布局中，其基类的布局按照基类被声明时的顺序进行排列
72  //4. 派生类会覆盖基类的虚函数，只有第一个虚函数表中存放的是
73  //    真实的被覆盖的函数的地址；其它的虚函数表中存放的并不是真实的
74  //    对应的虚函数的地址，而只是一条跳转指令
```

多基派生的二义性

```
//练习：
class A {
public:
    virtual void a() {  cout << "a() in A" << endl; }
    virtual void b() {  cout << "b() in A" << endl; }
    virtual void c() {  cout << "c() in A" << endl; }
};

class B {
public:
    virtual void a() {  cout << "a() in B" << endl; }
    virtual void b() {  cout << "b() in B" << endl; }
    void c() {  cout << "c() in B" << endl; }
    void d() {  cout << "d() in B" << endl; }
};

class C
: public A
, public B {
public:
    virtual void a() {  cout << "a() in C" << endl; }
    void c() {  cout << "c() in C" << endl; }
    void d() {  cout << "d() in C" << endl; }
};

void test() {
    C c;
    c.b();
    cout << endl;

    A* pA = &c;
    pA->a();
    pA->b();
    pA->c();
    cout << endl;

    B* pB = &c;
    pB->a();
    pB->b();
    pB->c();
    pB->d();
    cout << endl;
```

```
43
44      C *pC = &c;
45      pC->a();
46      pC->b();
47      pC->c();
48      pC->d();
49      return 0;
50  }
51
```

虚拟继承

在讲虚函数之前，我们首先问大家一个问题：从语义上来讲，为什么**动态多态**和**虚继承**都使用`virtual`关键字来表示?

virtual在词典中的解释有两种：

1. `Existing or resulting in essence or effect though not in actual fact, form, or name.`实质上的，实际上的：虽然没有实际的事实、形式或名义，但在实际上或效果上存在或产生的；
2. `Existing in the mind, especially as a product of the imagination. Used in literary criticism of text.`虚的，内心的：在头脑中存在的，尤指意想的产物，用于文学批评中。

`C++`中的`virtual`关键字采用第一个定义，即被`virtual`所修饰的事物或现象在本质上是存在的，但是没有直观的形式表现，无法直接描述或定义，需要通过其他的间接方式或手段才能够体现出其实际上的效果。关键就在于**存在**、**间接**和**共享**这三种特征：

- 虚函数是存在的
- 虚函数必须要通过一种间接的运行时（而不是编译时）机制才能够激活（调用）的函数
- 共享性表现在基类会共享被派生类重定义后的虚函数

那**虚拟继承**又是如何表现这三种特征的呢?

- 存在即表示虚继承体系和虚基类确实存在
- 间接性表现在当访问虚基类的成员时同样也必须通过某种间接机制来完成（通过**虚基表**来完成）
- 共享性表现在虚基类会在虚继承体系中被共享，而不会出现多份拷贝

虚拟继承是指在继承定义中包含了`virtual`关键字的继承关系。**虚基类**是指在虚继承体系中的通过`virtual`继承而来的基类。语法格式如下：

```
class Baseclass;
class Subclass
: public/private/protected virtual Baseclass
{
public:    //...
private:   //...
protected: //...
};
//其中Baseclass称之为Subclass的虚基类，而不是说Baseclass就是虚基类
```

接下来，我们看看例子：

```
#pragma vtordisp(off)
#include <iostream>
using std::cout;
using std::endl;

class A {
public:
    A() : _ia(10) {}

    virtual
    void f() {  cout << "A::f()" << endl;   }
private:
    int _ia;
};

class B
: virtual public A {
public:
    B() : _ib(20) {}

    void fb() { cout << "A::fb()" << endl;  }

    /*virtual*/ void f() {  cout << "B::f()" << endl;   }

    virtual void fb2() {    cout << "B::fb2()" << endl; }
private:
    int _ib;
};

void test(void) {
    cout << sizeof(A) << endl;
    cout << sizeof(B) << endl;
    B b;
```

```
    return 0;
}

// 结论一：单个虚继承，不带虚函数
//  虚继承与继承的区别
//  1. 多了一个虚基指针
//  2. 虚基类位于派生类存储空间的最末尾

// 结论二：单个虚继承，带虚函数
//  1. 如果派生类没有自己的虚函数，此时派生类对象不会产生虚函数指针
//  2. 如果派生类拥有自己的虚函数，此时派生类对象就会产生自己本身的虚函数指针，
//      并且该虚函数指针位于派生类对象存储空间的开始位置
```

虚拟继承时派生类对象的构造和析构

在普通的继承体系中，比如A派生出B，B派生出C，则创建C对象时，在C类构造函数的初始化列表中调用B类构造函数，然后在B类构造函数初始化列表中调用A类的构造函数，即可完成对象的创建操作。但在虚拟继承中，则有所不同。

```
class A {
public:
    A() {   cout << "A()" << endl;  }
    A(int ia): _ia(ia) {    cout << "A(int)" << endl;   }

    void f() {  cout << "A::f()" << endl;   }
protected:
    int _ia;
};

class B
: virtual public A {
public:
    B() {   cout << "B()" << endl;  }
    B(int ia, int ib)
    : A(ia), _ib(ib)
    {   cout << "B(int,int)" << endl;   }

protected:
    int _ib;
};

class C
: public B {
```

```
public:
    C(int ia, int ib, int ic)
    : B(ia, ib)
    , _ic(ic)
    {   cout << "C(int,int,int)" << endl;   }

    void show() const {
        cout << "_ia: " << _ia << endl
             << "_ib: " << _ib << endl
             << "_ic: " << _ic << endl;
    }
private:
    int _ic;
};

void test() {
    C c(10, 20, 30);
    c.show();
}

```

从最终的打印结果来看，在创建对象c的过程中，我们看到C带三个参数的构造函数执行了，同时B带两个参数的构造函数也执行了，但A带一个参数的构造函数没有执行，而是执行了A的默认构造函数。这与我们的预期是有差别的。如果想要得到预期的结果，我们还需要在C的构造函数初始化列表最后，显式调用A的相应构造函数。那为什么需要这样做呢?

在 C++ 中，如果继承链上存在虚继承的基类，则最底层的子类要负责完成该虚基类部分成员的构造。即我们需要显式调用虚基类的构造函数来完成初始化，如果**不显式调用**，则编译器会调用虚基类的**缺省构造函数**，不管初始化列表中次序如何，对虚基类构造函数的调用总是先于普通基类的构造函数。如果虚基类中**没有定义**的缺省构造函数，则会**编译错误。因为如果不这样做，虚基类部分会在存在的多个继承链上被多次初始化。**很多时候，对于继承链上的中间类，我们也会在其构造函数中显式调用虚基类的构造函数，因为一旦有人要创建这些中间类的对象，我们要保证它们能够得到正确的初始化。这种情况在菱形继承中非常明显， 我们接下来看看这种情况

菱形继承

```
#include <iostream>
using std::cout;
using std::endl;

class B {
```

```
public:
    B() : _ib(10), _cb('B') {   cout << "B()" << endl;  }
    B(int ib, char cb)
    : _ib(ib), _cb(cb){ cout << "B(int,char)" << endl;  }

    //virtual
    void f() {  cout << "B::f()" << endl;   }
    //virtual
    void Bf() { cout << "B::Bf()" << endl;  }
private:
    int _ib;
    char _cb;
};

class B1 : virtual public B {
public:
    B1() : _ib1(100), _cb1('1') {}
    B1(int ib, char ic, int ib1, char cb1)
    : B(ib, ic)
    , _ib1(ib1)
    , _cb1(cb1)
    {   cout << "B1(int,char,int,char)" << endl;    }

    //virtual
    void f() {  cout << "B1::f()" << endl;  }
    //virtual
    void f1() { cout << "B1::f1()" << endl; }
    //virtual
    void Bf1() {    cout << "B1::Bf1()" << endl;    }
private:
    int _ib1;
    char _cb1;
};

class B2 : virtual public B {
public:
    B2() : _ib2(1000), _cb2('2') {}
    B2(int ib, char ic, int ib2, char cb2)
    : B(ib, ic)
    , _ib2(ib2)
    , _cb2(cb2)
    {   cout << "B2(int,char,int,char)" << endl;    }

    //virtual
    void f() {  cout << "B2::f()" << endl;  }
```

```
    //virtual
    void f2() { cout << "B2::f2()" << endl; }
    //virtual
    void Bf2() {    cout << "B2::Bf2()" << endl;    }
private:
    int _ib2;
    char _cb2;
};

class D
: public B1
, public B2 {
public:
    D() : _id(10000), _cd('3') {}
    D(int ib1, char ib1,
      int ib2, char cb2,
      int id, char cd)
    : B1(ib1, ib1)
    , B2(ib2, cb2)
    , _id(id)
    , _cd(cd)
    {   cout << "D(...)" << endl;   }

    //virtual
    void f() {  cout << "D::f()" << endl;   }
    //virtual
    void f1() { cout << "D::f1()" << endl;  }
    //virtual
    void f2() { cout << "D::f2()" << endl;  }
    //virtual
    void Df() { cout << "D::Df()" << endl;  }
private:
    int _id;
    char _cd;
};

void test(void) {
    D d;
    cout << sizeof(d) << endl;
    return 0;
}
//结论：虚基指针所指向的虚基表的内容
//  1. 虚基指针的第一条内容表示的是该虚基指针距离所在的子对象的首地址的偏移
```

```
94  //  2. 虚基指针的第二条内容表示的是该虚基指针距离虚基类子对象的首地址的偏
    移
95
```

如果是在若干类层次中，从虚基类直接或间接派生出来的派生类的构造函数初始化列表均有对该虚基类构造函数的调用，那么创建一个派生类对象的时候**只有该派生类列出的虚基类的构造函数被调用**，**其他类列出的将被忽略**，这样就保证虚基类的唯一副本只被初始化一次。即虚基类的构造函数只被执行一次。

对于虚继承的派生类对象的析构，析构函数的调用顺序为：

- 先调用派生类的析构函数；
- 然后调用派生类中成员对象的析构函数；
- 再调用普通基类的析构函数；
- 最后调用虚基类 的析构函数。

效率分析

通过以上的学习我们可以知道，多重继承和虚拟继承对象模型较单一继承复杂的对象模型，造成了成员访问低效率，表现在两个方面：对象构造时`vptr`的多次设定，以及`this`指针的调整。对于多种继承情况的效率比较如下：

继承情况	vptr是否设定	数据成员访问	虚函数访问	效率
单一继承	无	指针/对象/引用访问效率相同	直接访问	效率最高
单一继承	一次	指针/对象/引用访问效率相同	通过vptr和vtable访问	多态的引入，带来了设定`vptr`和间接访问虚函数等效率的降低
多重继承	多次	指针/对象/引用访问效率相同	通过vptr和vtable访问;通过第二或后继Base类指针访问，需要调整this指针	除了单一继承效率降低的情形，调整this指针也带来了效率的降低
虚拟继承	多次	指针/对象/引用访问效率降低	通过vptr和vtable访问;访问虚基类需要调整this指针	除了单一继承效率降低的情形，调整this指针也带来了效率的降低

模板

现在的C++编译器实现了一项新的特性：模板（`Template`），简单地说，模板是一种通用的描述机制，也就是说，使用模板允许使用通用类型来定义函数或类等。在使用时，通用类型可被具体的类型，如`int`、`double`甚至是用户自定义的类型来代替。模板引入一种全新的编程思维方式，称为“泛型编程”或“通用编程”。

为什么要定义模板呢?

强类型程序设计中，参与运算的所有对象的类型在编译时即确定下来，并且编译程序将进行严格的类型检查。为了解决强类型的严格性和灵活性的冲突，有以下3中方式解决：

- 带参数宏定义（原样替换）
- 重载函数（函数名相同，函数参数不同）
- 模板（将数据类型作为参数）

形象地说，把函数比喻为一个游戏过程，函数的流程就相当于游戏规则。在以往的函数定义中，总是指明参数是`int`型还是`double`型等等，这就像是为张三（好比`int`型）和李四（好比`double`型）比赛制定规则。可如果王五（`char*`型）和赵六（`bool`型）要比赛，还得提供一套函数的定义，这相当于又制定了一次规则，显然这是很麻烦的。

模板的的引入解决了这一问题，不管是谁和谁比赛，都把他们定义成`A`与`B`比赛，制定好了`A`与`B`比赛的规则（定义了关于`A`和`B`的函数）后，比赛时只要把`A`替换成张三，把`B`替换成李四就可以了，大大简化了程序代码量，维持了结构的清晰，大大提高了程序设计的效率。该过程称为“**类型参数化**”。

在讲类型参数化之前，先来看一个示例：

```
int add(int x, int y)   //定义两个int类型相加的函数
{
    return x + y;
}

double add(double x, double y) //重载double类型
{
    return x + y;
}

```

```
char* add(char* px, char* py)//重载字符数组
{
    return strcat(px, py);  //调用库函数strcat
}

```

模板的定义

模板就是实现代码重用机制的一种工具，它可以实现类型参数化，即把类型定义为参数， 从而实现了真正的代码可重用性。模板可以分为两类，一个是**函数模版**，另外一个是**类模板**。通过参数实例化构造出具体的函数或类，称为**模板函数**或**模板类**。`C++`中通过下述形式定义一个模板:

```
template <class/typename T, ...>
```

对于一些更早接触`C++`的朋友，可能知道，在`C++`标准还未统一时，很多旧的编译器只支持`class`，因为那时`C++`并没有`typename`关键字。相比`class`，`typename`更容易体现“类型”的观点，虽然两个关键字在模板定义时是等价的，但从代码兼容的角度讲，使用`class`较好一些。

关键字`template`放在模板的定义与声明的最前面，其后是用逗号分隔的**模板参数表**，用尖括号（`<>`）括起来。模板参数表不能为空，**模板参数**有两种类型：

- `class`或`typename`修饰的类型参数，代表一种类型；
- 非类型参数表达式，必须是整型类型，使用已知类型符，代表一个常量

函数模板

函数模板的定义形式如下：

```
template <模板参数表>
返回类型 函数名(参数列表)
{
    //函数体
}
```

有了函数模板之后，我们就可以对之前的add函数进行定义了：

```
template <class T>
T add(T t1, T t2)
{
    return t1 + t2;
}
```

函数模板实际上不是个完整的函数定义，因为其中的类型参数还不确定，只是定义了某些类型的角色（或变量）在函数中的操作形式。因此，必须将**模板参数实例化**才能使用函数，用模板实例化后的函数也称为模板函数，分为**隐式实例化**和**显式实例化**。

```
void test0()
{
    int a1 = 1, a2 = 2;
    float b1 = 1.1, b2 = 2.2;
    double c1 = 1.123, c2 = 2.123;
    cout << add(a1, a1) << endl;
    cout << add(b1, b2) << endl;
    cout << add<double>(c1, c2) << endl;
}
```

再谈函数重载

有了函数模板后，如果定义的函数模板与普通函数同名，会发生什么事情呢？同样会进行重载，而且普通函数会优先于函数模板的执行。除此以外，函数模板与函数模板之间也会进行重载。

非类型参数

之前我们说过，模板参数除了是类型参数外，还可以设置非类型参数，下面我们来看一个示例：

```
template <class T, int kMax=100>
T multiply(T x, T y) {
    return kMax * x * y;
}

void test() {
    int x = 2, y = 3;
    cout << multiply<int, 10>(x, y) << endl;
}
```

非类型参数必须是整型类型，使用已知类型符，代表一个常量，而且还可以设置默认值。

模板特化

在函数模板add中，我们可以发现，如果传递的参数是`char *`指针时，程序就会出现问题。对于这种情况，我们有两种解决方案：一是直接重载一个版本，二是针对于这种情况设置模板特化版本。

特化版本的函数模板格式如下：

1. template后直接跟<>，里面不写类型
2. 在函数名后跟<>，其中写要特化的类型

```
template <>
返回类型 函数名<特化类型>（特化类型 参数1，特化类型 参数2，...）；
```

针对于函数模板add的特化版本如下：

```
template <>
const char * add<const char *>(const char * p1, const char *
p2)
{
    char * tmp = new char[strlen(p1) + strlen(p2) + 1]();
    strcpy(tmp, p1);
    strcat(tmp, p2);
    return tmp;
}
```

注意:

- 使用模板特化时，必须要先有基础的函数模板
- 特化的函数的函数名，参数列表要和原基础的模板函数想相同，避免不必要的错误

普通类的成员函数模板

成员函数模板既可以定义在普通类中，也可以定义在类模板中。在普通类中，使用成员函数模板时，不用提供模板参数，函数可以根据使用的参数，自动推导模板实参(`template argument`)对应的模板形参(`template parameter`)。

```
class CharArray {
public:
    CharArray(size_t capacity)
    : _capacity(capacity)
    , _buff(new char[_capacity]())
    {}

    CharArray(const char * pstr)
    : _capacity(strlen(pstr) + 1)
    , _buff(new char[_capacity]())
    {
        strcpy(_buff, pstr);
    }
```

```
14
15      template <class Iterator>
16      void printElements(Iterator beg, Iterator end)
17      {
18          while(beg != end) {
19              cout << *beg << '\n';
20              ++beg;
21          }
22      }
23
24      ~CharArray()
25      {
26          delete [] _buff;
27          cout << "~CharArray()" << endl;
28      }
29
30      const char * c_str() const
31      {   return _buff;   }
32
33  private:
34      size_t _capacity;
35      char * _buff;
36  };
37
38  void test()
39  {
40      CharArray ca("hello,world");
41      ca.show(ca.c_str(), ca.c_str() + 3);
42  }
```

可变模板参数

可变模板参数(`variadic templates`)是`C++11`新增的最强大的特性之一，它对参数进行了高度泛化，它能表示0到任意个数、任意类型的参数。由于可变模版参数比较抽象，使用起来需要一定的技巧，所以它也是`C++11`中最难理解和掌握的特性之一。

可变参数模板和普通模板的语义是一样的，只是写法上稍有区别，声明可变参数模板时需要在`typename`或`class`后面带上省略号“`...`”。比如我们常常这样声明一个可变模版参数：

`template<typename...>`或`template<class...>`

一个典型的可变模版参数的定义是这样的：

```
template <class... Args>
void func(Args... args);
```

上面的可变模版参数的定义当中，`Args`标识符的左侧使用了省略号，在`C++11`中`Args`被称为**模板参数包**，表示可以接受任意多个参数作为模板参数，编译器将多个模板参数打包成“单个”的模板参数包；`args`被称为**函数参数包**，表示函数可以接受多个任意类型的参数。

在`C++11`标准中，要求**函数参数包**必须唯一，且是函数的最后一个参数；**模板参数包**则没有。当**声明**一个变量(或标识符)为可变参数时，省略号位于该变量的**左侧**；当**使用参数包**时，省略号位于参数名称的**右侧**，表示立即展开该参数，这个过程也被称为**解包**。

求取可变模板参数的长度

下面来看一个简单的可变模板参数函数：

```
//打印实际传递参数的个数
template <class... Args>
void printTotalSize(Args... args) {
    cout << sizeof...(Args) << endl;
    cout << sizeof...(args) << endl;
}

void test()
{
    printTotalSize();
    printTotalSize(3.14159);
    printTotalSize('a', "hello");
    printTotalSize(1, 1.2, "hello,world");
}
```

展开可变模版参数函数的方法一般有两种：一种是通过**递归函数**来展开参数包，另外一种是通过**逗号表达式**来展开参数包。

递归函数展开参数包

通过递归函数展开参数包，需要提供一个参数包展开的函数和一个递归终止函数。

```
template <class T>
void print(T t) {
    cout << t << '\n';
}

template <class T, class... Args>
```

```
void print(T t, Args... args) {
    cout << t << '\n';
    print(args...);
}

void test() {
    print();
    print(3.14159);
    print('a', "hello");
    print(1, 1.2, "hello,world");
}
```

逗号表达式展开参数包

递归函数展开参数包是一种标准做法，也比较好理解，但也有一个缺点,就是必须要一个重载的递归终止函数。其实还有一种方法可以不通过递归方式来展开参数包，这种方式需要借助逗号表达式和初始化列表。比如前面print的例子可以改成这样：

```
template <class T>
void print(T t) {
    cout << t << '\n';
}

template <class... Args>
void printAll(Args... args) {
    int arr[] = {(print(args), 0)...};
}

void test()
{
    printAll(1, 'a', 12.12, "hello");
}
```

包扩展表达式

假设`args`被声明为一个函数参数包，其扩展方式有

- `printArgs(args…)`相当于`printArgs(args1,args2,…,argsN)`
- `printArgs(args)…`相当于`printArgs(args1),…, printArgs(argsN)`
- `(printArgs(args),0)…`逗号表达式，相当于`(printArgs(args1),0),…(printArgs(argsN),0)`

包扩展表达式“`exp…`”相当于将省略号左侧的参数包exp视为一个整体来进行扩展。

类模板

一个类模板允许用户为类定义个一种模式，使得类中的某些数据成员、默认成员函数的参数，某些成员函数的返回值，能够取任意类型(包括系统预定义的和用户自定义的)。

如果一个类中的数据成员的数据类型不能确定，或者是某个成员函数的参数或返回值的类型不能确定，就必须将此类声明为模板，它的存在不是代表一个具体的、实际的类，而是代表一类类。

类模板的定义形式如下：

```
template <class/typename T, ...>
class类名{
    //类定义. . . . . .
};
```

模板类是类模板实例化后的一个产物。比如，我们把类模板比作是一个做蛋糕的模具，而模板类就是用这个模具做出来的蛋糕，至于蛋糕是什么味道的，就要看你自己在实例化时用的是什么材料了，可以做巧克力蛋糕，也可以做草莓蛋糕，这些蛋糕除了某些材料不一样之外，其它的东西都是一样的了。

```
template <class T, int kMax = 10>
class Stack {
public:
    Stack()
    : _top(-1)
    : _base(new T[kMax]())
    {}

    bool empty() const;
    bool full() const;
    void push(T t);
    void pop();
    T top();

    ~Stack() {
        if(_base) {
            delete [] _base;
        }
    }
private:
    int _top;
    T * _base;
```

```
};

template <class T, int kMax>
bool Stack<T, kMax>::empty() const {    return  _top == -1;
}

template <class T, int kMax>
bool Stack<T, kMax>::full() const { return _top == kMax - 1;
}

template <class T, int kMax>
void Stack<T, kMax>::push(T t) {
    if(!full()) {
        _base[++_top] = t;
    } else {
        cerr << "stack is full, cannot push any more
data!\n";
    }
}

template <class T, int kMax>
void Stack<T, kMax>::pop() {
    if(!empty()) {
        --_top;
    } else {
        cerr << "stack is empty, no more data!\n";
    }
}

template <class T, int kMax>
T Stack<T, kMax>::top() {
    return _base[_top];
}
```

当然，模板还有一些其他的用法，不过目前我们就不深究了。