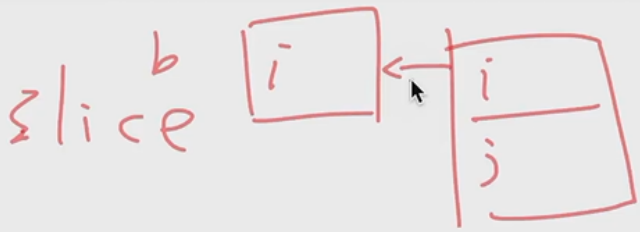
polymorphism

1 Slice and Up-casting

- **"slice(切片)"**：把**子类的对象**的值赋给**父类的对象**，但是父类的对象还是父类的对象，其中没有多出来它本来没有的变量(即子类才有的对象)；

```
D d;
d.prt();
B b;
b.prt();
b=d;
b.prt();
cout << sizeof(b) << endl;
```



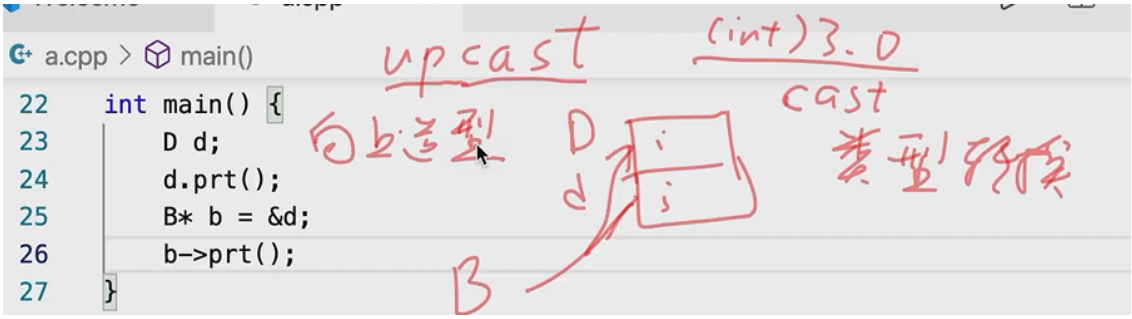
  - `eg1.` 将子类 `D` 的对象 `d` 赋值给父类 `B` 的对象赋值给 `b`：

```cpp
class B {
protected:
    int i = 0;
public:
    void prt() {
        cout << i << endl;
    }
};

class D : public B {
private:
    int j = 30;
public:
    D () {
        i = 10;
    }
};

int main() {
    D d;
    d.prt();    // Derived class's object
    B b;
    b.prt();    // Base class's object
    b = d;      // Assign a derived object to a base object
    b.prt();
    cout << "The size of b is: " << sizeof(b) << endl;
    return 0;
}
-----------------------------------------------------------------
10  // 子类对象的 i = 10
0   // 本来父类对象的 i = 0
10  // 将子类对象的 i 赋值给父类对象的 i，得到 10
The size of b is: 4 // 父类对象只有一个 int i，而没有子类对象的 j
```

- **"Up-casting"(向上造型)**：将一个子类对象的地址交给一个父类对象的指针(指着一个子类的对象说它就是父类的对象)，或者是将一个父类的引用指向一个子类的对象：



- Upcasting is the act of converting **from a Derived** reference or pointer **to a base class** reference or pointer.
  - `eg.` upcasting examples:

```
Manager pete( "Pete", "444-55-6666", "Bakery");
Employee* ep = &pete;   // Upcast
Employee& er = pete;    // Upcast
```

---

2 `virtual` keyword

- **Lose type information about the object**:

```
ep->print( cout );      // prints base class version
```

  - 假设父类 `Employee` 和子类 `Manager` 都有 `print` 函数的话，做了 **upcast** 之后，`ep->print` **是父类的** `print` **而不是原来子类对象** `&pete` **的** `print`.
- `virtual` **关键字**：当父类的某一个函数之前加上 `virtual` 后，当我们使用父类的指针去调用这个函数的时候，它能够根据这个指针所指向的那个对象到底是哪个类，调用那个对象所属的类的这个函数；
  - `eg.` 使用 `virtual` 关键字使得父类的函数成为虚函数：

```
class Point {};                 // Point class: denote a point
class Shape {                   // Shape class: denote a shape
private:
    Point center;
public:
    virtual void render() {
        cout << "shape::render" << endl;
    }
    void move() {}
};
class Rectangle:public Shape {  // Rectangle class: denote a rec
protected:
    int width;
    int height;
public:
    void render() {
        cout << "rec::render" << endl;
    }
};
class Square:public Rectangle { // Square class: denote a square
```

```
public:
    void render() {
        cout << "sqr::render" << endl;
    }
};
void render( Shape* s ) {
    s->render(); // calls correct render function for given shape
}
int main() {
    Square s;
    Rectangle r;
    s.render();     // static -- Square::render();
    r.render();     // static -- Rectangle::render();
    render( &s );   // dynamic -- Square::render();
    render( &r );   // dynamic -- Rectangle::render();
}
-----------------------------------------------------------------
sqr::render
rec::render
sqr::render
rec::render
```

- 虽然直接 `s.render()` 调用和 `(Shape*)s->render` 的结果是一样的，但是两者的调用过程是完全不一样的；

---

3 Polymorphism

```
Square s;
Rectangle r;
s.render();     // static -- Square::render();
r.render();     // static -- Rectangle::render();
render( &s );   // dynamic -- Square::render();
render( &r );   // dynamic -- Rectangle::render();
```

3.1 Two basis of Polymorphism

- Upcast: take an object of the derived class as an object of the base one.
  - 把一个子类的对象看成是父类的对象，从编程的角度：父类的指针指向子类对象的地址；
- Dynamic binding:
  - Binding: which function to be called:
    - **Static binding**: call the function as the code，编译时刻绑定，`virtual` 无所谓；
    - **Dynamic binding**: call the function of the object，编译时刻不知道的.

3.2 Principles of Polymorphism

☑ 任何一个编程语言要实现 Polymorphism 必须首先要实现 upcast；

☑ Virtual functions：

- Non-virtual functions：
  - **Compiler generates static**, or direct call to stated type，编译时刻静态绑定；
  - Faster to execute；

- Virtual functions：
  - Objects carry a pack of their virtual functions;
  - Can be **transparently overridden** in a derived class;
    - **overridden**：指的是父类和子类中**有返回类型和参数表特征完全相同**的函数，并且**父类的这个函数前面有** `virtual` **关键字**，那么它们构成 overridden 关系；
  - Compiler checks pack and dynamically calls the right function, If compiler knows the function at compile-time, it can generate a static call.

- [x] 多态变量：如果某一个**指针变量**它指向的那个类型里面是有 `virtual` 函数的，那么它就叫做多态变量，每一个多态变量都有两种类型：
  - 静态类型 / 声明类型：`eg.` `Shape *s`，被声明为 `Shape` 类型；
  - 动态类型 / 实时类型：根据我们传递给这个指针的对象的类型来实时确定；

---

4 Implementation of `virtual` function

```cpp
class Shape {
protected:
    int x = 10;
public:
    void render() { cout << "shape::render" << endl; }
    void move() {}
};

class Rectangle:public Shape {
public:
    void render() { cout << "rec::render" << endl; }
};

class Square:public Rectangle {
public:
    void render() { cout << "sqr::render" << endl; }
};

void render( Shape* s ) { s->render(); }

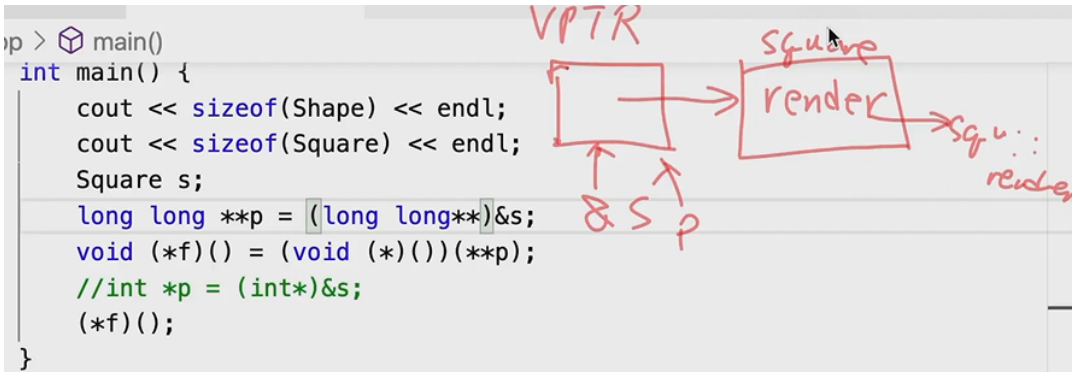
int main() {
    cout << sizeof(Shape) << endl;
    cout << sizeof(Square) << endl;
    Square s;
    int *p = (int*)&s;
    cout << *p << endl;
}
--------------------------------------------------------------------------------
/* There is no virtual function in the Shape */
4   // an int variable of Shape
4   // an int variable inherits from Shape
10  // it x = 10;

/*************************************************************************/
class Shape {
protected:
    // int x = 10;
```

```
public:
    virtual void render() { cout << "shape::render" << endl; }
    void move() {}
};
-------------------------------------------------------------------------
/* There is a virtual function render() in the Shape */
8           // a pointer's size
8           // a pointer's size
4935152     // 指向了一个 render 函数的地址
```

- 没有 `virtual` 函数的情况下：
  - `Shape` 类中有多少个变量，那么 `Shape` 类的大小就是这么些变量的总大小；
  - `Shape` 类中没有变量的时候，`Shape` 类的大小是 `1 Byte`，因为不能没有大小；
- 含有 `virtual` 函数的情况下：
  - 如果没有任何变量的情况下，`Shape` 类的大小是 `8 Byte`，是一个指针的大小；

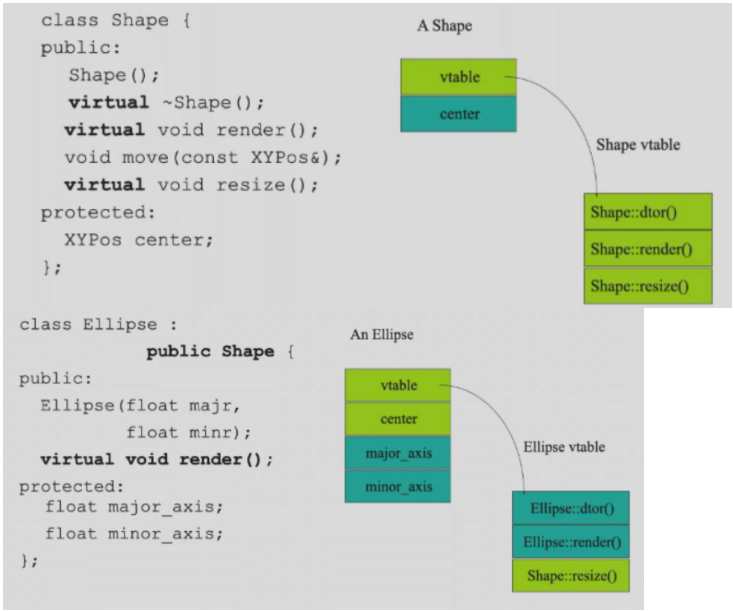
```
cout << sizeof(Shape) << endl;
cout << sizeof(Square) << endl;
Square s;
long long **p = (long long **)&s;
void (*f)() = (void (*)())(**p); // f is a function pointer(函数指针)
(*f)();
```



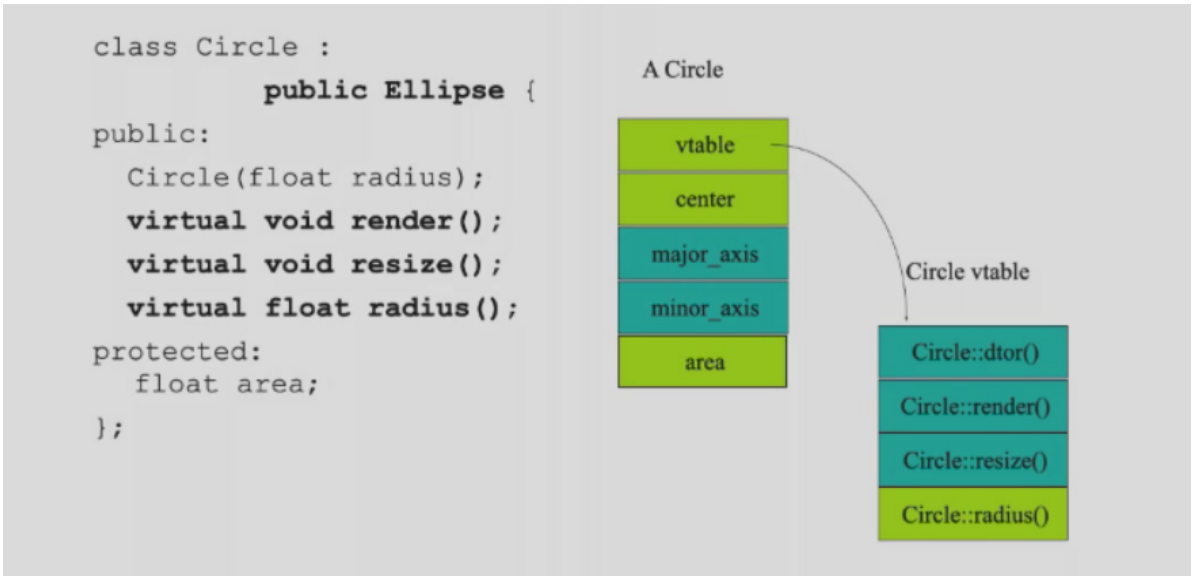
  - 这个 `s` 对象中只有一个指针 `VPTR`，它指向了一张表 `vtable`，这张表是它所属的 `Square` 类的表：
    - `vtable`：每一个类只有一张表，里面包含了该类的所有 `virtual` 函数的指针；
    - `VPTR`：每一个对象都有一个指向它所属的类的 `vtable` 的 `VPTR` 指针；

4.1 How `virtual` work in C++

如果一个类含有 `virtual` 函数的话，那么它的对象包含的第一个东西一定是一个 `VPTR` 指针，这个指针指向了这个类的 `vtable`，里面包含了这个类所有的 `virtual` 函数的函数指针。



`vtable` 其实是一张静态的表，因为在编译过程中这些 `virtual` 函数的地址都是确定的。



动态绑定：当我要用父类的指针调用子类对象的 `virtual` 函数的时候，它就会通过 `VPTR` 找到那个类的 `vtable` 找到对应的那一个函数指针，来进行调用：

- 所以动态绑定就是通过这一系列指针操作来实现的，而不是在程序执行的时候来拷问对象到底是什么类的，然后去找那个类的成员函数来实现。

4.2 What happens if

```
Ellipse elly(20F, 40F);
Circle circ(60F);
elly = circ;        // Slice，做的是赋值
```

- Area of `circ` is sliced off, only the part of `circ` that fits in `elly` gets copied;
- `vtable` from `circ` is ignored, the `vtable` in `elly` is the `Ellipse` `vatble`.
- ```
  shape *p = &elly;
  s->render();        // Ellipse::renser();
  ```
  ```

`VPTR` 的赋值只发生在构造函数里面，只有在进入构造函数的那一刻会执行 `VPTR` 的赋值操作。

4.3 Virtual destructors

- Make destructors `virtual` if they might be inherited:

```
Shape *p = new Ellipse(100.0F, 200.0F);
...
delete p;
```

- Want `Ellipse::~Ellipse()` to be called:
  - Must declare `Shape::~Shape()` `virtual`;
  - It will call `Shape::~Shape()` automatically.
- If `Shape:~Shape()` is not `virtual`, only `Shape::~Shape()` will be invoked!!!

如果我们希望一个类是会有子类的，那么就要把它的析构函数(destructor)变成 `virtual` 的，使得子类对象日后能正常调用它们自己的析构函数，而不是只能调用父类的析构。

4.4 Return types relaxation(current)

Suppose `D` is publicly derived from `B`, then `D::f()` can return a subclass of the return type defined in `B::f()`.

- It applies to pointer and reference types, `eg.` `D&, D*`;

Relaxation example

```
class Expr {
public:
   virtual Expr* newExpr();
   virtual Expr& clone();
   virtual Expr  self();
};

class BinaryExpr : public Expr {
public:
   virtual BinaryExpr* newExpr();     // Ok
   virtual BinaryExpr& clone();// Ok
   virtual BinaryExpr self();  // Error!
};
```

4.5 Overloading and `virtual`

Overloading adds multiple signatures:

```
class Base {
    public:
        virtual void func();
        virtual void func(int);
};
```

- If we override an overloaded function, you must override all of the variants!
    - Can not override just one;
    - If we don't override all, some will be hidden.