نکته درمورد عملگر = و Copy Constructor:

فرض کنیم کلاس Student را داریم که شامل یک پوینتر int به اسم ptr می باشد: int *ptr;

قطعه کد زیر را داریم:

```
Student S1(127, "Ali");
Student S2 = S1;          (1)
Student S3;
S3 = S1;                  (2)
```

در دستور (1) برای اینکه پوینتر S1 و S2 به یک مکان در Heap اشاره نکنند به سراغ Copy Constructor می رود و در دستور (2) به سراغ overload کردن عملگر = . در صورتی که Copy Constructor باشد،

پوینتر هم در S2 و هم در S3 عوض خواهند شد. اما
اگر فقط عملگر مساوی = را overload کرده باشیم،
آدرس پوینتر در S2 مشابه با S1 خواهد شد ولی S3 تغییر
می‌کند. در نتیجه Copy Constructor برتری دارد نسبت به
overload = .

---

virtual چیست؟ فرض کنیم قطعه کد زیر را داریم:

```
class base
{ Public : int i ;
};
```

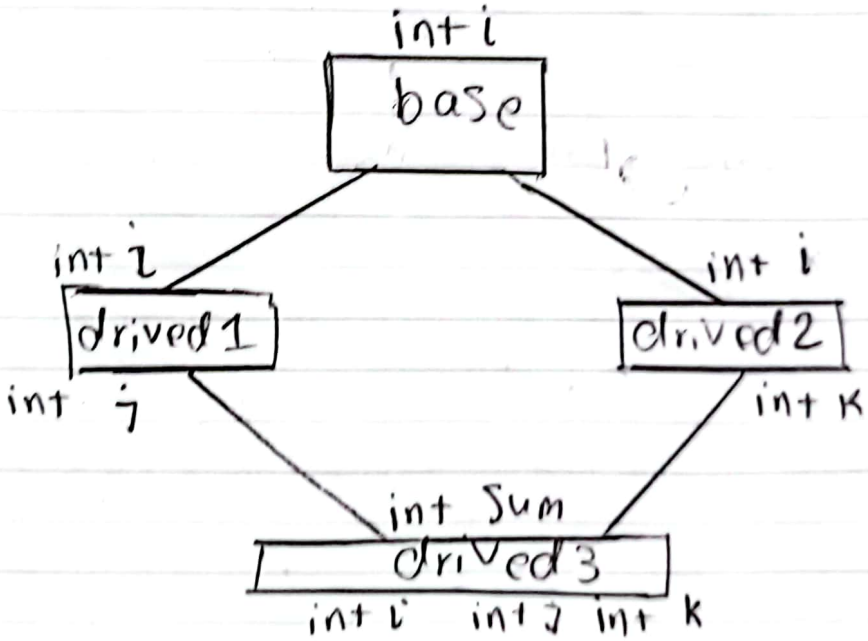


```
Class drived1 : Public base
{
    public : int j;
};

Class drived2 : Public : base
{
    public : int k;
};
```

```
class drived3 : public drived1,
 public drived2
{
  public: int Sum;
}
```

در اینجا کلاس drived3 شامل attrebiute های i, j, k, Sum است.



در استفاده از متغیر i برای اشیاء کلاس drived3، کامپایلر به مشکل برمی خورد که کدام یک از i ها را می خواهی استفاده کنی؟ زیرا یکی از drived1 است یا

drived 2 ؟ از

۲ راه وجود دارد:

۱- توسط اشاره گر شخص میسازیم:

drived 3 obj.

obj.drived1::i;

" drived2::i;

۲- از virtual استفاده میکنیم: در کلاس هایی که می دانیم از متغیرهای ارث بری شده شان بازهم ارث بری خواهد شد، از virtual استفاده می کنیم:

class drived 1: virtual public base

{

"

}

class drived 2: virtual public base

{

"

}

چطوره کلاس ۱ از کلاس پدر را درون کلاس ۱ از کلاس فرزند بریزیم؟

فرض کنیم کلاس پدر Person است و کلاس فرزند Student می‌باشد.

```
Student S1;
Person Pobj;
S1 = Pobj;   ⇝ Error
```

۲ راه برای این مشکل وجود دارد:

راه اول: می‌توانیم درون کلاس Person یک cast به Student بزنیم: `Student = (Student)Pobj;`

راه دوم: می‌توانیم درون کلاس Student یک overload = مساوی انجام دهیم:

```
// S1.operator =(Pobj)
```

در مورد توابع virtual : فرض کنید کلاس drived1 و drived2 از کلاس base ارث‌بری کرده‌اند. همچنین کلاس base دارای تابع virtual x می‌باشد. تابع x را drived1 دارد ولی drived2 ندارد.

مقدمه‌ی زیر را بنویسید: (مثال آشنا)

```
Shap * p;

Rectangle Rec;   p = & Rec;   1
```

drived 1 ←

```
Circle circ;   p = & circ;   2
```

drived 2 ←

حالت 1:

```
p -> x();
```

در این حالت پاسخ تابع x از drived 1 می‌رود.

حالت 2:

```
p -> x();
```

در این حالت هم پاسخ تابع x از drived 1 می‌رود.

نکته: در همین حالت، به جز کلاس base، به دنبال نزدیک‌ترین نسخه از تابع virtual، درون فرزندان و نوه‌ها می‌گردد که در اینجا x درون فرزند base، یعنی drived1 بود.

---

* فرض کنیم در مثال قبلی کلاس drived2 هم دارای تابع x باشد؛ در این صورت drived1 دیگر نیازی به ساختن تابع x() virtual نیست، چرا که کلاس پدر بزرگ (base) این کار را کرده و در کل فرزندان و فرزندان فرزندان این تابع virtual می‌باشد.

---

* مفهوم abstract class: کلاس‌هایی معین، کلاس shape که دارای مفاهیم انتزاعی هستند، مثل محاسبه area و perimeter هستند؛

(مفهوم انتزاعی: یعنی به طور خاص نمی‌دانیم برای چه shape ای داریم area و perimeter محاسبه

فی‌نفسه abstract class می‌گوییم، یعنی کلاسی که تنها مفهوم انتزاعی را پیاده‌سازی می‌کند.

این توابع به شکل Pure Virtual هستند، یعنی:

```
virtual int area() = 0;
virtual int Perimeter() = 0;
```

* اگر کلاسی abstract باشد، نمی‌توانیم از آن object

بسازیم چون‌که اگر object بسازیم متعاقبتاً دسته‌ای از توابع را هم خواهیم داشت و چون آن توابع مفاهیمی کلی هستند نمی‌توانیم آن‌ها را فراخوانی کنیم.

* درصورتی که کلاس فرزند از کلاسی base ای ارث‌بری شده که شامل pure virtual باشد ولی همه‌ی آن توابع را در کلاس خود فراخوانی نکند، آن هم به عنوان abstract کلاس شناخته می‌شود. و نمی‌توانیم از آن object بسازیم.

destructo in abstract class (*)

مطلعه کد زیر را داریم:

```
Rectangle rec;
Shap *P= & rec;
```

درصورتی که به سراغ تخریب P برویم، چون از جنس shape* است، فقط destructo کلاس Shape فراخوانی می‌شود و این مقصود ما نیست.

شاید ما نیاز داشته باشیم که ابتدا destructo کلاس Rectangle را صدا بزنیم و بعد destructo کلاس Shape فراخوانی شود.

صرف اینکه با فراخوانی destructo کلاس Shape، فقط ویژگی‌های مشترک بین کلاس پدر و فرزند هستند که تخریب می‌شوند.

راه حل چیست؟ باید destructo کلاس Shape را virtual تعریف کنیم:

```
virtual ~Shape() {}
```

اینکه در اینجا هم طبق ویژگی های inheritance دوباره مخرب کلاس shape() فراخوانی می شود؛ با این تفاوت که پس از آن مخرب Rectangle() فراخوانی شده است.

---

* در کلاس shape هم توابع تابع ~shape() را داشته باشیم و آن را پیاده سازی نکنیم. یک اشتباه است و آن این است که با وجود pure virtual، باز هم در بیرون از کلاس shape آن را پیاده سازی نکنیم.

```
virtual ~shape() = 0; // Pure virtual
};

shape::~shape( ){
//
}
```

نکته مهم: درون سمی تابع calculate()

به صورت زیر تعریف شده است:

```
Void Calculate( Shape * S )
{
  Cout << "area :" << S -> area() << endl;
  Cout << "perimeter :" << S -> Perimeter()
       << endl;
}
```

اما فرض کنید می‌خواهیم کلاس Rectangle مثلاً به عنوان یک ورودی دیگه به طور خاص انجام بده مثلاً یه کار خاص از Rectangle به اسم x را فراخوانی کن.
۲ راه حل داریم:

۱- استفاده از typeid:

```
if (typeid(*S) == typeid(Rectangle))
{
    Rectangle *r = (Rectangle*)S;
    r -> x();
}
```

۲- استفاده از dynamic_cast :

dynamic_cast فقط مخصوص Polymorphism است و باید به شکل زیر استفاده می‌شود :

```
Rectangle *r = dynamic_cast
<Rectangle*>(s);
```

در اینجا چک می‌کند که اگر تایپ s ، Rectangle بود بدون r s را برگردان اگر تایپ s چیزی دیگر بود بدون آن NULL برگردان ،

پس داریم : `if (r) { r->x(); }`