Table of Contents

صریحا می‌توانید اجازه ندهید از توابعی که کامپایلر تولید می‌کند استفاده شود

فرض کنید که یک سیستم مدیریت املاک داریم که کارش فروختن خانه هست، و نرم افزاری که چنین چیزی را مدیریت می‌کند طبیعتا دارای یک کلاس هست که خانه‌های برای فروش را ارایه می‌کند:

```
class HomeForSale
{
};
```

همچنان که هر مشاور املاکی در این سیستم به سادگی قابل دسترسی است، هر ویژگی نیز یکتا هست؟ نه ممکنه چند ملک در چند املاکی به ثبت رسیده باشه. این موردی است که، ایده‌ی کپی کردن **HomeForSale** منطقی به نظر میرسد. چطور می‌تونیم چیزی رو کپی کنیم که از یک کلاس یونیک ارث بری کرده؟ بنابراین احتمالا شما دوست خواهید داشت که چنین کدهایی برایتان کامپایل نشوند.

```
HomeForSale h1;
HomeForSale h2;
HomeForSale h3(h1);  // we want to not compile
h1=h2;   //we want to not compile
```

در هر صورت، جلوگیری از کامپایل این کدها خیلی هم آسان نیست. معمولا، وقتی شما می‌خواهید که یک کلاس خاص از یک کاربرد خاص پشتیبانی نداشته باشه، به صورت ساده اون رو کلا تعریف نمی‌کنید ولی همونطور که در آیتم ۵ دیدیم، این استراتژی برای **کپی سازنده** و **اپراتور انتساب** عمل نمی‌کند. چون اگه اون‌ها رو تعریف نکنیم کامپایلر در صورت لزوم ورژن دلخواه خودش از این‌هارا برایمان می‌سازد.

همچین چیزی شمارو توی تنگنا قرار میده، چرا که اگر کپی سازنده و یا اپراتور انتساب رو اعلان نکرده باشید، کامپایلر خودش این توابع رو برای شما خواهد ساخت. بنابراین کلاس، قابلیت کپی کردن رو پیدا می‌کنه. و اگر خودتون این توابع رو اعلان کرده باشید در این صورت هم دوباره کلاس از این توابع پشتیبانی داره!! . ولی هدف ما در این بخش این هست که از انجام چنین کاری ممانعت به عمل بیاوریم.

و اما راه حل چیست؟ کامپایلر این توابع رو به صورت **public** تولید می‌کنه و برای این که جلوی تولید کردن دوباره این توابع رو بگیریم، مجبوریم اون‌هارو خودمون تعریف کنیم، اما نیازی نیست که این توابع رو به صورت عمومی تعریف کنیم. بنابراین کپی سازنده و اپراتور انتساب رو به صورت خصوصی تعریف می‌کنیم. با اعلان کردن یک تابع عضو از این که کامپایلر دوباره کاری کنه و اون توابع رو بسازه جلوگیری کردیم، و با خصوصی اعلان کردنش می‌تونیم مطمعن بشیم که کسی نمی‌تونه بیرون از کلاس این توابع رو صدا بزنه.

این روش خیلی روش مطمعن و جامعی نیست(بعدها در C++ مدرن روش کامل‌تر را خواهیم دید). چرا این روش کاملی نیست؟ چرا که توابع عضو و توابع دوست همچنان می‌توننند توابع خصوصی رو صدا بزنند. مگراین کـه شـما بـه انـدازه‌ی کافی باهوش باشید که اون‌ها رو کلا تعریف نکنید. حال اگر کسی کد شمارو داشته باشه و بخواهد که این چنین چیزی رو صدا بزنه، در موقع **link-time** به ارور خواهد خورد. این کلک که توابع عضو رو به صـورت خصوصـی تعریـف کـنیم و حواسمون جمع باشه که از اون‌ها خودمون استفاده نکنیم، در یـک کتابخـانه‌ی خیلی معـروف مثلا کتابخـانه **iostream** انجام شده.

به عنوان مثال می‌تونید یک نگاهی به نحوه‌ی پیاده‌سازی **ios_base, basic_ios** و **sentry** تـوی پیاده‌سـازی **standat library** داشته باشید. وقتی که این پیاده‌سـازی‌هارو نگـاه کنیـد متوجـه می‌شـوید کـه هم **copy constructor** و هم **copy assignment operator** به صورت خصوصی اعلان شده و هیچوقت تعریف یا **define** نشده‌اند. اعمال این کلک روی کلاس خودمون خیلی ساده است ببینید:

```
class HomeForSale
{
public:
   HomeForSale();    //declare constructor
private:
   HomeForSale(const HomeForSale&);          //declare but not defined
   HomeForSale& operator=(const HomeForSale&);  //declare but not defined
};
HomeForSale::HomeForSale()   //define constructor
{
}
```

شاید در کدهای بالا این نظرتون رو به خودش جلب کرده باشه که چرا نام پارامترهای توابع رو اصلا نیاورده‌ایم!!. در واقع نیازی به ذکر کردن این اسامی نیز نیست، این یک توافق عمومی توی زبان هست. چون این توابع اصلا قـرار نیسـت کـه پیاده‌سازی بشن، پس چه نیازی به ذکر کردن اسامی پارامترها هست؟

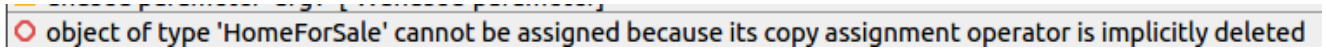
با پیاده سازی بالا، کامپایلر هیچ دسترسی‌ای به مشتری نمیده که بتونه شیء **HomeForSale** رو کپی کنه، و حتی اگه شما بخواین توی تابع عضو و یا یک تابع دوست از این توابع ممنوعه استفاده کنید، **linker** بهتون ارور میـده. در خـاطر داشته باشید که ما می‌تونیم خطایی که موقع **link-time** میگیریم رو بـه **compile time** بـبریم(انجـام این کـار خیلی توصیه میشه، چون این که ارور رو زودتر بگیریم بهتر از این هست که بعدا بخواهیم خطارو ببینیم)، این کار رو می‌تونیم با اعلان کپی سازنده و اپراتور انتساب به صورت خصوصی انجام بدیم، اما نـه در خـود کلاس بلکـه در کلاس **base** بایـد این کار انجام بشه، فرض کنید کلاس **base** ما یک کلاس به صورت زیر باشه:

```
class unCopyable
{
protected:
   unCopyable() {}
   ~unCopyable() {}
private:
   unCopyable(const unCopyable&);
   unCopyable& operator=(const unCopyable&);
};
```

در این صورت کلاس **HomeForSale** رو از این کلاس ارث بری می‌کنیم.

```
class HomeForSale :private unCopyable
{
};
```

در این صورت اگه بخواهیم، مثلا اپراتور انتساب رو استفاده کنیم به همچین اروری برخورد خواهیم کرد.



چرا این کد درست کار می‌کنه؟ چرا که کامپایلر تلاش می‌کنه تا کپی سازنده و اپراتور انتساب رو برای هر کسی که تلاش می‌کنه تا شیء **HomeForSale** رو کپی کنه، بسازه، حتی اگه تابع عضو و یا تابع دوست باشه. همانطور که در آیتم ۱۲ خواهیم دید، تابعی که کامپایلر برای این‌ها تولید می‌کنه، سعی می‌کنه تا همتای این توابع رو از کلاس **base** صدا بزنه، و چنین فراخوانی **reject** میشه، چرا که این توابع در کلاس **base** به صورت خصوصی تعریف شده.

پیاده‌سازی و استفاده از **Uncopyable** ظرافت خاص خودش رو داره، مثل این مورد که ارث‌بری از **Uncopyable** نیازی نیست به صورت **public** باشه(آیتم ۳۲ و ۳۹ رو ببینید)، و این که مخرب **Uncopyable** نیازی نیست به صورت **virtual** باشه(آیتم ۷ رو ببینید). چرا که **Uncopyable** هیچ نوع دیتایی نداره، در این صورت مستعد بهینه‌سازی کلاس خالی **base** هست که در آیتم ۳۹ خواهیم دید، استفاده از این تکنیک ممکنه منجر به ارث‌بری چندگانه بشه(آیتم چهلم رو ببینید). ارث‌بری چندگانه، در عوض، ممکنه بهینه‌سازی کلاس خالی رو کنسل کنه(آیتم ۳۹ رو ببینید). در حالت کلی، شما می‌تونید این ظرافت‌های طراحی رو نادیده بگیرید و همانطوری که **Uncopyable** رو دیدیم ازش استفاده کنید. همچنین شما می‌تونید از ورژنی که توی **Boost** هست استفاده کنید (آیتم ۵۵ رو ببینید). اسم اون کلاس **noncopyable** . این کلاس، کلاس مناسبیه فقط اسمش یه خورده غیر عادی بود که من عوضش کردم.