Table of Contents

استفاده از const و enum و inline ها نسبت به define کردن ارجحیت دارد

بهتر بود نامگذاری این قسمت به این صورت بود: ***ترجیح دادن کامپایلر به پیش پردازنده***، چرا چنین چیزی رو میگیم؟ چون ممکنه با #define به عنوان یک بخش از زبان برخورد نشه. این تنها یکی از مشکلات استفاده از دستورات پیش پردازنده هست. هنگامی که شما تکه کد زیر رو می‌نویسید:

```
#define ASPECT_RATIO 1.653
```

در این مورد اسمی که تحت عنوان ASPECT_RATIO مشخص کرده‌اید، ممکن است هیچگاه توسط کامپایلر دیده نشه، یعنی قبل از این که کد تحویل کامپایلر بشه، توسط پیش پردازنده این اسم حذف بشه. در این صورت، نام ASPECT_RATIO ممکنه هیچگاه وارد symbol table نشه.

نکته: **symbol table چیست؟** symbol table یک ساختمان داده بسیار مهم هست که توسط کامپایلر ساخته شده و نگه‌داری میشود، این امر به منظور پیگیری رفتار داده‌ها انجام میشود یعنی کامپایلر اطلاعاتی در مورد scope و اطلاعات نام‌ها، اطلاعاتی در مورد نمونه‌های موجود مثل متغیرها و نام کلاس‌ها و توابع و اشیاء و غیره را ذخیره می‌کند.

در این صورت، نام ASPECT_RATIO هیچگاه وارد symbolic table نخواهد شد. در این صورت اگر شما هنگام کامپایل کردن به اروری در مورد استفاده از مقدار constant برخورد کنید و اشاره به مقدار ۱.۶۵۳ کنه شما نخواهید توانست بفهمید منظور کامپایلر همون ASPECT_RATIO هستش. اگر ASPECT_RATIO در فایل هدر و توسط یک شخص دیگر نوشته شده باشد، در این صورت شما هرگز نمی‌توانید بفهمید که این مقدار ۱.۶۵۳ از کجا اومده، و این موضوع تنها اتلاف وقت خواهد بود. این موضوع حتی موقع استفاده از دیباگر نیز دردسر ساز خواهد شد، چرا که، ممکنه اسمی که شما دارید استفاده می‌کنید در symbol table نباشه. در این صورت راه حل استفاده از یک مقدار constant به جای استفاده از ماکرو خواهد بود.

```
const double AspectRatio=1.653;
```

این مقدار ثابت بوده و متعلق به زبان می‌باشد و همواره توسط compiler دیده می‌شود و قطعا وارد symbol table نیز می‌شود. همچنین در مورد متغیرهای ممیز شناور مثل همین مثال خودمون استفاده از constant موجب تولید کد کمتری نسبت به نمونه‌ی مشابه با #define میشود. این مورد به این دلیل رخ میدهد که وقتی از #define استفاده می‌کنیم، پیش پردازنده هر جا که از ASPECT_RATIO استفاده شده را با مقدار ۱.۶۵۳ جایگزین می‌کند و به همین دلیل چندین کپی از مقدار ۱.۶۵۳ در object code شما تولید میشه، اما اگر AspectRatio به صورت constant استفاده بشه در این صورت یک کپی تنها در برنامه خواهد بود.

وقتی ما #define هارو با مقادیر constant جایگزین می‌کنیم، دو نکته پیش میآید که باید گفته شود. اول تعریف مقادیر constant به صورت اشاره‌گری است. چرا که معمولا تعریف constant در فایل هدر انجام می‌پذیرد، این مهم است که پوینتری که استفاده می‌شود نیز به صورت const تعریف شود. برای تعریف یک مقدار constant از نوع *char در هدر فایل می‌بایست دوبار از const استفاده شود.

```
const char* const str="Saeed Masoomi";
```

برای بررسی معنا و مفهوم کامل استفاده از const ، به خصوص در مورد استفاده با اشاره‌گرها، باید تا آیتم بعدی منتظر بمانیم، اما، لازم به ذکر است که استفاده از اشیاء string نسبت به *char های ارجحیت دارند، در این صورت رشته‌ی پیشین که نوشتیم را می‌توانیم به سادگی به صورت زیر بنویسیم.

```
const std::string str("Saeed Masoomi");
```

دومین موردی که نیاز به بررسی دارد، استفاده از constantها در کلاس‌ها است. برای محدود کردن scope یک مقدار constant به یک کلاس، شما باید آن را به عنوان یک عضو کلاس تعریف کنید، و برای

این که اطمینان پیدا کنید که تنها یک کپی از آن مقدار constant وجود دارد، مجبورید آن عضو را به صورت static دربیاورید.

```
class GamePlayer{
 private:
   static const int numTurns =5;
   int scores[numTurns];
};
```

آنچه که شما در کد بالا میبینید یک declartion(اعلان) از NumTurns بوده، و نه یک definition . در واقع در declare کردن شما به کامپایلر در مورد size، type اطلاعات را میدهید ولی در declare کردن هیچ مقداری در حافظه رزرو نمی‌شود. اما در define کردن علاوه بر این موارد شما رزرو حافظه نیز دارید. به صورت معمول، در C++ شما برای هر چیزی که استفاده می‌کنید باید یک definition انجام بدید، اما در مورد constant هایی که مربوط به class هستند و به صورت static درآمده‌اند و از نوع integralای هستند(یعنی typeهایی مثل integer، char و bool و غیره) یک استثنا محسوب می‌شوند. تا وقتی که شما آدرسی برای آن‌ها تعیین نکرده باشید می‌توانید declare شون کنید و بدون این که definitionای براشون تعیین کرده باشید از آن‌ها استفاده بکنید. اگر آدرس یک مقدار constant مربوط به کلاس را نگیرید، یا این که compiler شما به صورت اشتباه اصرار بر definition داشته باشید (با این وجود که شما آدرس آن را نگرفته باشید) می‌توانید به صورت زیر یک تعریف مجزا برای آن فراهم کنید.

```
const int GamePlayer::numTurns;       //definition of numTurns
```

کد بالا که برای define کردن استفاده می‌شود، می‌بایست در فایل implementation باشد، نه در هدر فایل. برای این که مقدار اولیه‌ی ثابت کلاس وقتی فراهم می‌شود که مقدار ثابت declare می‌شود(یعنی numTurns مقدار ۵ را وقتی declare می‌شود میگیرد، و مقدار اولیه‌ای در هنگام define کردن نمی‌تواند بگیرد.

دقت کنید که، نمی‌توان constant مربوط به یک class را با استفاده از #define ایجاد کرد، چرا که #define اهمیتی به scopeهای ما نمیدهد. زمانی که یک ماکرو تعریف شود، برای همه‌ی قسمت‌های کامپایل آن ماکرو صحیح و معتبر است(مگر این که شما در جایی در بین خطوط کد آن را #undefed کنید). که این بدین معنی است که نه تنها نمی‌توان از #define برای مقادیر constant مربوط به classها استفاده کرد، همچنین نمی‌توان برای هیچگونه encapsulation از آن استفاده کرد، یعنی، وقتی ما از #define استفاده می‌کنیم دیگر چیزی به شکل private نداریم. در نظر داشته باشید که داده‌های const را می‌توان کپسوله کرد.

کامپایلرهای قدیمی‌تر ممکن است که شکل کد بالا را نپذیرند، چرا که فراهم کردن مقدار اولیه برای یک عضو static در هنگام declare کردن درست نیست. در نتیجه، کامپایلرهای قدیمی‌تر به شما اجازه

میدهند که مقداردهی اولیه درون کلاسی داشته باشید آن هم تنها برای نوع‌های integralای و تنها برای constant ها. در مواردی که syntax بالا نامعتبر باشد و نتوانیم از آن استفاده کنیم، شما باید مقداردهی اولیه را هنگامی که در حال define کردن هستید انجام دهید.

```
class CostEstimate
{
public:
   static const double FudgeFactor;
};
//in the implementation file
const double CostEstimate::FudgeFactor=1.35;
```

این تقریبا همه‌ی چیزی که شما نیاز دارید را رفع می‌کند. تنها یک استثناء وجود دارد و آن هم وقتی است که شما مقدار ثابت یک کلاس را در هنگام کامپایل تایم نیاز داشته باشید، مثل declartion یک آرایه که کامپایلر نیاز به دانستن سایز آرایه داشته باشد و کامپایلرهای قدیمی چنین چیزی را پشتیبانی نمی‌کردند. در این صورت راه حل پذیرفته شده استفاده از enum hack بود که به صورت زیر از آن استفاده می‌شد.

```
class Bunch {
 enum { size = 1000 };
 int i[size];
};
```

بهتره یک سری اطلاعات بیشتر در مورد enum hack رو با همدیگه ببینیم. اول این‌که، روش enum hack بیشتر از اون که شبیه const رفتار کند، رفتاری شبیه به #define دارد، و در برخی موارد شما چنین چیزی را می‌خواهید. به طور مثال، مشکلی وجود ندارد که ما آدرس یک const را بگیریم، اما ما نمی‌توانیم آدرس یک enum را بگیریم که در مورد #define نیز دقیقا به همین‌گونه می‌باشد. اگر شما بخواهید که دیگران نتوانند اشاره‌گر و یا رفرنسی رو بر روی مقدار ثابت integralای شما بگیرند، استفاده از enum همچین چیزی رو برای شما مرتفع می‌کند.(برای کسب اطلاعات بیشتر می‌توانید آیتم ۱۸ را ببینید). همچنین، کامپایلرهای خوب حافظه‌ی جداگانه‌ای برای اشیاء const کنار نمیگذارند(مگر این‌که شما یک اشاره‌گر و یا رفرنس به شیء بسازید)، اما کامپایلرهای متوسط ممکنه این کارو بکنند، و شما

قطعا دوست دارید که حافظه‌ی جداگانه‌ای برای چنین اشیایی ساخته نشـه. مثـل #define، اسـتفاده از enum ها از چنین هدر رفت حافظه اضافه جلوگیری می‌شود.

علت دوم بـرای آشـنایی بـا enum hack کـاملا یـک چـیز عملی اسـت. خیلی از کـدها از آن اسـتفاده می‌کنند،بنابراین لازمه که وقتی چنین چیزی رو دیدیم بشناسـیمش. در حقیقت، enum hack پـایه‌ی تکنیک template metaprogramming هست.

بگذارید به بحث اصلی این قسمت برگردیم، یعـنی دسـتورات پیش پردازنـده، یـک اسـتفاده نادرسـت از #define برای پیاده‌سازیه macro ها می‌باشد که شبیه توابع هستند اما سربار فراخـوانی تـابع را ندارنـد. در اینجا یک ماکرو را با هم می‌بینیم که یک تابع مثل f را فراخوانی می‌کند.

گویا در C++ جدید نمی‌توانیم از توابع ماکرو به همون صورتی که دیدیم استفاده کنیم.

ما از توابع ماکرو به دو دلیل استفاده می‌کردیم یکی این که هـر تـایپی را می‌پذیرفتنـد و دیگـر این کـه سربار فراخوانی نداشتند، خوشبختانه ما می‌توانیم همه‌ی این مزیت‌هـا را بعلاوه‌ی رفتارهـای قابـل پیش بینی و امنیت تایپ خروجی را با استفاده از توابـع عـادی داشـته باشـیم، همه‌ی چـیزی کـه نیـاز داریم استفاده از template و inline function می‌باشد.(در کد زیر f یک تابع دیگر می‌باشد).

```
template<typename T>
inline void callWithMax(const T&a,const T&b)
{
   f(a>b?a:b);
}
```

در کد بالا چون نوع متغیر T را نمیدانیم، آن را به صورت refrence به const پاس میدهیم.

با توجه به موجود بودن enum ،const و inline نیـاز شـما بـه دسـتورات پیش پردازنـده بـه خصـوص #define کاهش پیدا می‌کند، امـا واقعـا نمیتـوان دسـتورات پیش پردازنـده را حـذف کـرد. #Include کماکان ضروری است، و دسـتورات #ifdef و #ifndef کماکـان نقش مهمی در کنـترل کـردن کامپایـل دارند. هنوز نمی‌توانیم از دست دستورات پیش پردازنده خلاصی پیدا کنیم، اما شما باید از پیش پردازنده کمتر استفاده کنید.

نکات مبهم

۱- چطور نمی‌توانیم آدرس یک enum را بگیریم؟ مگر آن‌ها در حافظه نیستند؟

برای پاسخ به این سوال enum زیر را در نظر بگیرید:

```
enum example{
   first_value,
   second_value
};
```

در این حالت گرفتن آدرس first_value ممکن نیست چون در واقع first_value در حافظـه مقـداری را به خود اختصاص نداده است، بلکه صرفا یک مقدار ثابت در حافظه است، و یک نام دیگر بـرای حـافظه‌ی صفرم(جایی که nullptr ها به آن اشاره می‌کنند) که البته شما نمی‌توانید آدرس آن را بگیرید.

اما در حالتی که شما یک enum رو declare کرده باشید(یک نمونه از این enum ساخته باشـید) در این صورت می‌تونید حافظه آن را بگیرید مثال زیر می‌تواند یک مثال از enum بالا باشد.

```
   enum example ex;
   enum example *pointer=&ex;
```

۲- منظور از آدرس صفر چیست؟

آدرس صفر به جایی اشاره دارد که nullptr به آن اشاره دارد. مثال زیر را در این مورد ببینید.

```
   int* pointer=nullptr;
   cout<<pointer<<endl;
```

در مثال بالا خروجی pointer برابر با صفر است. در واقـع nullptr بـه این اشـاره دارد کـه پوینـتری کـه ساخته شده فعلا به شیء خاصی اشاره ندارد، مگر این که آدرس آن را عوض کنیم.

۳- symbol table چیست؟

symbol table یک ساختمان داده برای استفاده توسط کامپایلر می‌باشد، که در آن هر شناسه‌ی کد در سورس برنامه به همراه اطلاعات الحاق شده و همچـنین تعـاریف مربـوط ذخـیره می‌شـوند. در symbol table هر شناسه به همراه type، scope و خطی که آن اتفاق افتاده ذخیره می‌شود.

Symbol table می‌تواند توسط الگوریتم‌های LinkedList و HashTable و یا Tree ها پیاده‌سازی شود. معمولا از اپراتورهای زیر برای تعریف یک symbol table استفاده می‌شود.

Operation	Function
allocate	to allocate a new empty symbol table
free	to remove all entries and free storage of symbol table
lookup	to search for a name and return pointer to its entry
insert	to insert a name in a symbol table and return a pointer to its entry
set_attribute	to associate an attribute with a given entry
get_attribute	to get an attribute associated with a given entry

کد زیر را در C++ در نظر بگیرید.

```
// Define a global function
int add(int a, int b)
{
   int sum = 0;
   sum = a + b;
   return sum;
}
```

symbol table مربوطه برای کد بالا به صورت زیر خواهد بود.

NAME	TYPE	SCOPE
add	function	global
a	int	function parameter
b	int	function parameter
sum	int	local

آیتم ۳: هر جا موقعیتش وجود داشت از const استفاده کنید.

یک نکته‌ی جالب در مورد const این است که شما می‌توانید یک محدودیت معنایی به این صورت تعریف کنید که یک شیء خاص نمی‌تواند در طول برنامه تغییر کند- و کامپایلر این محدودیت را اعمال خواهد کرد. این موضوع به شما اجازه میدهد که هم به کامپایلر و هم به دیگر برنامه‌نویس‌ها بگویید که یک متغیر می‌بایست در طول برنامه بدون تغییر باقی بماند. هر موقع دوست داشتید که چنین اتفاقی

برای یک شیء بیفته، باید حتما از const استفاده کنید، چرا که تنها در این صورت می‌باشد که می‌توانید روی کمک کامپایلر حساب کنید تا چنین محدودیتی هیچگاه نقض نشه.

می‌توان گفت که کلمه‌ی کلیدی const یک کلمه‌ی کلیدی چند کاره است. در بیرون از کلاس‌ها، می‌توانید از const برای متغیرهای global و یا namaspaceها استفاده کنید، همچنین برای آبجکت‌هایی که به صورت static در function, file و یا scope block تعریف شده‌اند می‌توانید از آن استفاده کنید.در داخل کلاس‌ها، می‌توانید هم به صورت static و هم به صورت non-static از const استفاده کنید. برای اشاره‌گرها، می‌توانید در مورد const بودن دو چیز تصمیم بگیرید، یکی این که خود pointer به صورت const باشه، و یا این که داده‌هایی که به آن اشاره می‌شود به صورت const باشد.

```
char greeting[] = "hello";

char* p = greeting;                 //non-const pointer,
                                    //non-const data

const char *p = greeting;           //non-const pointer,
                                   //const data

char * const p = greeting;         //const pointer,
                                  //non-const data

const char * const p = greeting;   //const pointer,
                                    //const data
```

در نگاه اول syntax استفاده شده پیچیده به نظر میرسد. اگر const سمت چپ * دیده بشه در این صورت data ای که اشاره‌گر بهش اشاره میکنه به صورت const هستش، اما اگر const سمت راست * دیده بشه در این صورت خود اشاره‌گر به صورت ثابت تعریف می‌شود، و اگر const در هر دو سمت دیده شود در این صورت هر دو ثابت خواهند شد.

بعضی از برنامه‌نویس‌ها به خاطر قاعده‌ی بالا ممکن است که const رو قبل از type بیارن که تغییری در معنای کد نخواهد داشت. در این صورت هر دو تابع زیر ممکن است در دنیای واقعی وجود داشته باشند.

```
void f1(const int *pw);   // f1 takes a pointer to a constant int
```

```
void f2(int const *pw);   //f2 does so
```

از اونجایی که هر دو فرم در دنیای واقعی وجود دارند، باید با این فرمت‌ها آشنا بشید.

iteratorهای STL بر اساس اشاره‌گرها مدل شده‌اند، بنابراین یک iterator خیلی شبیه به T* رفتار می‌کند. تعریف یک iterator به عنوان const به مثابه‌ی تعریف یک pointer const هستش(یعنی، تعریف به صورت T* const): در این صورت iterator قادر نخواهد بود که به چیز دیگری اشاره کند و تغییر کند، اما چیزی که به آن اشاره می‌کند می‌تواند تغییر کند یعنی دیتا می‌تواند تغییر کند. اگر شما نیاز به یک iteratorای دارید که چیزی که بهش اشاره میشه به صورت const باشه و قابل تغییر نباشه (یعنی const T*)، در این صورت شما نیاز به یک **const_iterator** دارید:

```
 std::vector<int> vec;


const std::vector<int>::iterator iter=vec.begin();
*iter=10;  //OK, changes what iter points to
++iter;  //error,iter is const


std::vector<int>::const_iterator citer=vec.begin();
*citer=10;  //error! *citer is const
++citer;   //fine,changes citer
```

یکی از قویترین کاربردهای استفاده از const، کاربردی است که در تعریف توابع دارند. در تعریف تابع، const می‌تواند اشاره به متغیر برگردان شده از تابع، پارامترهای تکی و یا عضو تابع داشته باشند و یا به کل تابع اشاره داشته باشند.

این که تابعی داشته باشیم که یک مقدار ثابت را برگرداند این امکان را در اختیار برنامه‌نویس میگذارد که خطاهایی که سمت مشتری گرفته می‌شود را بدون این که امنیت و یا کارآیی کد کاهش پیدا کند، کاهش دهد. به طور مثال، operator (*) را در نظر داشته باشید که برای اعداد rational که در بخش ۲۴ بررسی خواهید کرد استفاده می‌شوند.

```
class Rational{...};
const Rational operator*(const Rational& lhs,const Rational& rhs);
```

بسیاری از برنامه‌نویس‌ها وقتی این کد رو میبینند دچار اشتباه می‌شوند(ترسناکه:)) . چرا باید خروجی operator* به صورت یک آبجکت const باشه؟ چون اگه همچین چیزی نبود ممکن بود کسی که از این کد استفاده می‌کنه دچار یک اشتباه بزرگی مثل این بشه:

```
Rational a,b,c;
(a*b)=c;
```

نمی‌دونم دقیقا چرا یک برنامه‌نویس دوست داره که حاصلضرب دو تا عدد را برابر با یک مقداری قرار بده، ولی می‌دونم برنامه‌نویس‌های زیادی وجود دارند که بدون این‌که بخواند، این اشتباه رو مرتکب می‌شوند. به عنوان یک مثال حالت زیر رو ببینید.

```
if(a*b=c)
```

مثلا در کد بالا برنامه‌نویس هدفش مقایسه دو مقدار بوده ولی اشتباها داره عمل انتساب رو انجام میده.چنین کدی در مورد متغیرهای built-in منجر به خطا می‌شود و کامپایل نمی‌شود. یکی از نشانه‌های این که یک متغیر که توسط user تعریف شده خوبه یا نه این هست که بتونن با متغیرهای built-in ترکیب شوند و همچنین عملکرد یکسانی داشته باشند.(برای اطلاعات بیشتر بخش ۱۸ رو ببینید)، این که اجازه بدیم که حاصلضرب دو تا مقدار برابر با یک عبارت قرار بگیره به اندازه‌ی کافی برای من عذاب آوره. تعریف اپراتور ضرب (*) به صورت const می‌تونه از چنین چیزی جلوگیری کنه، و دقیقا به همین دلیل هست که در اینجا از const استفاده کردیم.

در مورد پارامتر const چیز جدید وجود ندارد، اونا دقیقا مثل اشیاء const محلی رفتار می‌کنند.

توابع عضو const

هدف استفاده از const بر روی توابع عضو این است که مشخص کنیم چه توابعی بر روی اشیاء const باید صدا زده شوند. چنین توابعی به دو دلیل اهمیت دارند، **اول** این که، باعث میشند که ماهیت کلاس به راحتی فهمیده بشه، این خیلی مهمه که بدونیم چه تابعی ممکنه یک شیء رو تغییر بده و چه تابعی نمی‌تونه. **دوم** این که، اون‌ها به ما اجازه می‌دهند که با اشیاء const کار کنیم. این مورد خیلی در نوشتن کد سریع مهمه، در این مورد در آیتم ۲۰ بیشتر خواهیم دید، اما یکی از راه‌های ابتدایی برای بهبود بخشیدن به کارآیی برنامه پاس دادن آبجکت‌ها به صورت refrence-to-const هست.این تکنیک وقتی قابل دسترسی دارد که توابع عضو const وجود داشته باشند.

خیلی از مردم این حقیقت رو نادیده میگیرند که می‌تونند برای اشیاء const توابع overload مربوطه‌ش رو بنویسند، اما این یک ویژگی‌مهم در زبان C++ است.

یک کلاس را که به منظور مدیریت یک تکه متن نوشته شده را در نظر بگیرید:

```
class TextBlock{

public:
    TextBlock(const char* in){text=in;}

   const char& operator[](std::size_t position) const
   {return text[position];}

   char& operator[](std::size_t position)
   {return text[position];}

private:
   std::string text;
};
```

اپراتور [] مربوط به TextBlock رو می‌تونیم به صورت‌های زیر استفاده کنیم.

```
   TextBlock tb("Hello");
   std::cout<<tb[0]<<endl;

   const TextBlock ctb("World");
   std::cout<<ctb[0]<<endl;
```

در مورد کد بالا وقتی tb[0] رو صدا می‌زنیم تابع non-cost صدا زده میشه چون شیء tb به صورت const تعریف نشده ولی در مورد فراخوانی ctb[0] تابع const صدا زده میشه.

همچنین توجه داشته باشید که مقدار برگردان شده از اپراتور [] از شیء non-const ، یک رفرنس به char است-یعنی خود char برگردان نخواهد شد. اگر اپراتور[] یک char ساده را برگردان میکرد، عبارتی مثل حالت زیر کامپایل نمیشد.

```
tb[0]='x';
```

این به این دلیل است که نمی‌توان مقدار برگردان شده از یک تابع را در صورتی که type به صورت built-in باشد را تغییر داد. حتی اگر انجام چنین کاری انجام شدنی بود، این حقیقت که C++ اشیاء را by value برگردان می‌کند(آیتم شماره ۲۰ را برای این مورد ببینید)، این معنی را خواهد داشت که یک کپی از tb.text[0] تغییر پیدا خواهد کرد، نه خود tb.text[0]، و این رفتاری نیست که شما بخواهید.

اجازه بدهید یک نگاه مختصری به فلسفه این موضوع بپردازیم. این که تابع عضو به صورت const باشد چه معنی‌ای خواهد داشت؟ دو فلسفه در این مورد وجود دارد: bitwise constness (که همچنین به عنوان physical constness شناخته می‌شود) و همچنین logical constness.

فلسفه bitwise const باور دارد که تابع عضو به صورت const است اگر و تنها اگر، هیچ دیتای عضو کلاس را تغییر ندهد(حتی آن‌هایی که به صورت static هستند)، یعنی هیچ تغییری در آبجکت ندهد. یک چیز خوب در مورد bitwise constness این است که پیدا کردن violation در این فلسفه خیلی آسان است : در این حالت کامپایلر تنها به این نگاه می‌کند که assignmentای روی داده‌ی عضو کلاس رخ داده یا نه. در حقیقت، bitwise constness تعریف C++ از constness هست، و یک تابع عضو const اجازه‌ی تغییر دادن اعضای داده‌ای non-static را از شیءای که invoke شده ندارد.

متاسفانه، بسیاری از توابع عضو که این فلسفه const را تا حدودی رعایت می‌کنند، تست bitwise رو قبول میشند. به طور مشخص، یک تابع عضو که تنها پوینتری که به چیزی اشاره می‌کند را تغییر میدهد، مثل یک تابع عضو const عمل نمی‌کند. اما اگر تنها اشاره‌گر درون شیء باشد، تابع bitwise const بوده، و کامپایلر ایرادی به آن نمیگیرد. این موضوع می‌تواند باعث رخ دادن یک رویه غیرعادی در برنامه شود. به طور مثال فرض کنید که ما یک کلاس مثل TextBlock که قبلا دیدیم، در نظر بگیریم که در آن دیتا به صورت *char ذخیره شده(به صورت string نیست)، به این دلیل که در این مورد نیاز به ارتباط با C API داریم، چیزی در مورد اشیاء string نخواهد دانست.

```
class TextBlock{
public:
   TextBlock(char* in){pText=in;}

   char& operator[](std::size_t position) const
   {return pText[position];}

private:
   char *pText;
};
```

این کلاس به طور نامناسبی اپراتور [] را به عنوان یک تابع عضو const تعریف کرده است، این تعریف با توجه به این که خود تابع یک رفرنس به داده‌ی درونی شیء را برمیگرداند درست نیست(این موضوع به صورت گسترده تری در بخش ۲۸ مورد بررسی قرار خواهد گرفت). صرف نظر از این مشکل، اپراتور [] هیچگونه تغییری نمی‌تواند در pText اعمال کند. در نتیجه، کامپایلر بدون دردسر و با خوشحالی برای اپراتور [] کد رو تولید خواهد کرد، چون تمام چیزی که کامپایلر در این مورد بررسی می‌کند، این است که bitwise const درست باشه، اما بیایید نگاه کنیم ببینیم این کد می‌تونه باعث چه اتفاقی بشه:

اگر یک برنامه به صورت زیر بنویسیم ممکنه بتونیم مقدار char* pText رو تغییر بدیم.

```
const TextBlock cctb("Hello");
char *pc=& cctb[0];
*pc='r';
```

اما در خاطر داشته باشید که چون این const یک bitwise-const هست ممکنه بتونید مقدار رو تغییر بدید و ممکنه هم نتونید این کار رو انجام بدید. در واقع در این مورد خاص کامپایلر ایرادی به کد شما نمیگیرد و کد را برای شما کامپایل می‌کند ولی موقع اجرا ممکنه برنامه crash کند.

مطمعنا هر کسی می‌تونه به کد قبلی ایراد بگیره که چرا تنها تابع عضو const را برای آن ساخته‌ایم و در عین حال می‌خواهیم یک مقدار non-static رو تغییر بدهیم؟در واقع این مشکل در مورد استفاده از توابع const وجود داره، ما دوست داریم که توابع const رو داشته باشیم در عین حال متغیرهایی نیز جزو کلاس وجود داشته باشند که بتونیم اون‌هارو درون تابع const تغییر بدهیم.

این مشکل ما رو به سمت logical constness رهنمون می‌کنه. کد زیر را ببینید:

```
class TextBlock{
public:
   std::size_t length() const;
private:
   char *pText;
   std::size_t textLength;
   bool lengthIsValid;
};
std::size_t TextBlock::length() const
{
```

```
    if(!lengthIsValid)
    {
        textLength=strlen(pText);
        lengthIsValid=true;
    }
    return textLength;
}
```

در این کد ما در تابع length() قصد داریم که مشتری هر موقع درخواست داد مقدار txtLength برگردان بشه، که همون اندازه‌ی text ورودی است. چنین پیاده‌سازی‌ای قطعا نمی‌تواند یک bitwise const باشد، چرا که همانطور که در کد مشخص است هم txtLength و هم lengthIsValid ممکن است مقدارشان در طول برنامه تغییر کند. اما کامپایلرها بر روی bitwise constness پافشاری می‌کنند و اجازه نمیدهند درون یک تابع const شما یک دیتای عضو را تغییر بدهید. در این صورت شما چکار می‌کنید؟

راه حل خیلی ساده است: در این مورد از mutable استفاده خواهیم کرد. Mutable داده‌ی non-static عضو را از محدودیت bitwise آزاد می‌کنند. در این صورت کد به صورت زیر خواهد شد.

```
class TextBlock{
public:
    std::size_t length() const;
private:
    char *pText;
    mutable std::size_t textLength;
    mutable bool lengthIsValid;
};
std::size_t TextBlock::length() const
{
    if(!lengthIsValid)
    {
        textLength=strlen(pText);
```

```
        lengthIsValid=true;
    }
    return textLength;
}
```

جلوگیری از دوبار استفاده کردن تابع عضو const و non-const

استفاده از mutable یک راه حل خوب برای مساله‌ی bitwise-constness هست، اما mutable نمی‌تونه همه‌ی مشکلات مربوط به مشکلات مربوط به const رو حل و فصل کنه. به طور مثال، اپراتور [] را در کلاس TextBlock در نظر بگیرید که نه تنها مشکل برگرداندن رفرنس به character رو داره، بلکه مشکل bounds checking و یه سری مشکلات دیگه رو هم داره. قرار دادن همه‌ی این مشکلات در توابع اپراتور const و non-const ممکنه چنین مشکلاتی رو برای ما ایجاد بکنه:

```
class TextBlock{
public:
    const char& operator[](std::size_t position) const
    {
        //...
        //do bound checking, log access data
        // and verify data integrity
        //...
        return text[position];
    }
    char& operator[](std::size_t position)
    {
        //...
        //do bounds checking
        //log access data
        //verify data integrity
        //...
        return text[position];
```

```
    }
private:
    std::string text;
};
```

اوپس، می‌تونید ببینید که کد رو دوبار داریم تکرار می‌کنیم، که باعث میشه کامپایل تایم کد بالا بره، نگه‌داری کد سخت‌تر بشه، و همچنین موجب code-bloat بشه. قطعا میشه که یه چیزایی مثل bound checking و بقیه موارد رو به توابع دیگه انتقال دید(طبیعتا باید private باشه) که هر دو ورژن [] operator بتونن صداش بزنند، اما بازم شما دارید دوبار یک کد رو استفاده می‌کنید.

تمام اون چیزی که ما نیاز داریم این هست که اپراتور [] را یک بار تعریف کنیم و دو کاربرد برای آن داشته باشیم، در این صورت شما نیاز دارید که یک ورژن از اپراتور [] داشته باشید که دیگری رو صدا بزنه. و این مورد نیاز به این داره که constness رو کنار بگذاریم.

به عنوان یک قاعده کلی، cast کردن یک ایده بد به شمار میآید، در واقع ما یک آیتم برای همین مورد اختصاص دادیم که بگیم cast کردن ایده خوبی نیست که در آیتم ۲۷ خواهیم دید، اما این که کد رو دوبار بنویسیم از اون هم بدتره. در این مورد، ورژن const اپراتور [] دقیقا همان کاری را انجام می‌دهد که ورژن non-const انجام می‌دهد. کنار گذاشتن const بودن یک شیء در هنگام برگردان کردن یک کار امن به حساب میآید. در این مورد، چون کسی که اپراتور non-const رو صدا میزنه در ابتدا باید یک شیء به صورت non-const داشته باشه، در این صورت کنار گذاشتن const بودن مقدار برگردان شده امن هستش. در غیر این صورت نمی‌توان آن شیء را صدا زد. در این صورت داشتن یک اپرتور non-const، که ورژن const رو صدا میزنه یک راه مطمعن برای جلوگیری از دوباره نویسی کد هست، اگر چه در این مورد مجبوریم از cast استفاده کنیم. در اینجا کد این مورد را خواهیم دید ولی برای این که کد و چیزی که الان گفتیم براتون مشخص‌تر بشه بهتره که توضیحاتی که در ادامه میدیم رو هم با دقت بخونید.

```
class TextBlock{
public:
    const char& operator[](std::size_t position) const   //same as before
    {
        //...
        //...
        //...
```

```
        return text[position];
    }
    char& operator[](std::size_t position)
    {
        return
            const_cast<char&>(                          //  cast away const on [] operation
                static_cast<const TextBlock&>(*this)    //; add const to * this's type;
                [position]                              //  call const version of op[]
                );
    }
private:
    std::string text;
};
```

همانطور که می‌توانید ببینید، کد دو تا cast داره، نه یکی. ما می‌خواهیم که اپراتور non-const فرم const رو فراخوانی بکنه، اما اگه ما داخل اپراتور non-const، اپراتور[] رو صدا بزنیم، در این صورت بصورت برگشتی فقط خودمون رو صدا خواهیم زد و تو لوپ میفیتیم. برای جلوگیری از یک لوپ بینیهایت، ما باید مشخص کنیم که می‌خواهیم اپراتور[] را از نوع const صدا بزنیم، اما راه مستقیمی برای این کار وجود نداره. بجای این کار، ما تایپ طبیعیه *this را از TextBlock& به const TextBlock& تغییر میدهیم یا همان cast می‌کنیم. بله ما از cast برای اضافه کردن const استفاده کردیم! در این صورت ما دو تا cast خواهیم داشت: یکی برای اضافه کردن const به *this(برای این که بگیم اپراتور[] ورژن const رو صدا بزنه)، و دیگری برای حذف کردن const از اپراتور[] برای مقداری که برگردان شده.

آن castای که const بودن رو اضافه می‌کنه تنها برای این هست که تبدیل امنی ایجاد بشه(نه این که طوری بشه که یک شیء non-const به فرمت const ارسال بشه)، در این صورت ما از static_cast برای این مورد استفاده کردیم. و castای که const بودن رو حذف می‌کنه، با استفاده از const_cast قابل انجام هست، بنابراین در این مورد ما راه حل دیگری نداشتیم(به صورت تکنیکال، داریم، C-style cast می‌تونه در این مورد به ما کمک کنه، اما طبق توضیحاتی که در آیتم ۲۷ خواهیم داد، چنین castای در موارد خیلی کمی باید مورد استفاده قرار بگیرد، اگر شما در حال حاظر با static_cast و const_cast آشنایی دارید در این صورت آیتم ۲۷ یک مرور کلی بر دانسته‌های شما خواهد بود).

سوای هر چیز دیگری، ما در این مثال یک اپراتور را فراخوانی کرده‌ایم، بنابراین یه مقداری syntax استفاده شده عجیب شد. نتیجه ممکنه یک کد زیبا نباشه، اما نتیجه خوبی بر روی جلوگیری از دوباره نویسی کد داره. ما به نتیجه‌ای که می‌خواستیم رسیدیم، اما این که این کار ارزشش رو داره چیزی هست که شما می‌تونید خودتون تعیینش کنید، دوبار یک کد رو بنویسید و یا این که به این syntax ترسناک اکتفا کنید، اما این تکنیک ارزش گفتن رو داشت.

آیتم ۵: فانکشن‌های C++ چه چیزی را مینویسند و فراخوانی می‌کنند.

چه موقع یک کلاس خالی یک کلاس خالی نیست؟ وقتی که C++ وارد میدان می‌شود. وقتی شما چیزی رو تعریف نکنید، خود کامپایلر ورژن خودش رو از کپی سازنده، اپراتور انتساب، و destructor رو اعلان می‌کنه، اگر شما کلا هیچ constructorای رو اعلان نکرده باشید در این صورت کامپایلر یک سازنده پیش‌فرض برای شما ایجاد می‌کنه. همه‌ی این توابع به صورت public و inline خواهند بود. در نتیجه، اگر شما یک همچین چیزی رو بنویسید:

```
class empty
{


};
```

این در واقع مشابه این هست که شما کد رو به صورت زیر می‌نوشتید:

```
class Empty
{
public:
   Empty(){}                  //default constructor
   Empty(const Empty& rhs){}     //copy constructor
   ~Empty(){}                  //destructor
   Empty& operator =(const Empty& rhs){}  //copy assignment operator
};
```

این توابع در صورت نیاز بهشون تولید میشند، اما این که بهشون نیاز داشته باشیم خیلی ساده‌ست، کدهای زیر باعث تولید هر کدام از توابع بالا میشند:

```
Empty e1;  //default constructor
Empty e2(e1);  // copy constructor
e2=e1;  //copy assignment
```

این که اجازه بدهیم که کامپایلر توابع را برای ما بنویسد، در این صورت توابع دقیقا چه کاری برای ما انجام خواهند داد؟ سازنده پیش فرض و یک نابودگر اولیه در واقع توسط کامپایلر جاگذاری می‌شود. توجه داشته باشید که نابودکننده به صورت non-virtual می‌باشد(آیتم ۷ رو ببینید)، مگر این که این کلاس از یک کلاس دیگر ارث بری کرده باشد که در اون کلاس یک destructor به صورت virtual تعریف شده باشد.

همچنین برای copy constructor و اپراتور copy assignment ، کامپایلر تمام داده‌های عضو non-static رو در سورس مربوط به شیء نهایی کپی می‌کنه. به طور مثال، یک template به نام NamedObject رو در نظر بگیرید که به شما اجازه میده که با اشیاء نوع T کار کنید:

```
template<typename T>
class NamedObject{
  public:
    NamedObject(const char* name,const T& value);
    NamedObject(const std::string& name,const T& value);
private:
    std::string nameValue;
    T objectValue;
};
```

چون یک سازنده در namedObject اعلان شده، کامپایلر constructor پیش‌فرض رو در این مورد تولید نمی‌کنه. این خیلی مهمه. این یعنی که اگر شما با دقت بیشتری یک کلاس رو طراحی کنید، که در اون سازنده‌ای وجود داشته باشه که آرگومان بگیره، در این صورت نیاز نیست نگران این باشیم که کامپایلر دوباره بخواد سازنده کلاس رو دوباره بسازه، که حتی هیچ آرگومانی هم نپذیره.

NamedObject نه کپی سازنده و نه اپراتور انتساب رو اعلان کرده، در این صورت خود کامپایلر دست به کار میشه و این توابع رو تولید می‌کنه(البته یادتون باشه که در صورتی که نیاز باشه این کار رو انجام میده). به کد کپی سازنده زیر نگاه کنید. نحوه‌ی تعریف سازنده کلاس در پیاده سازی به صورت زیر خواهد بود.

```
template<typename T>
```

```
NamedObject<T>::NamedObject(const char *name, const T &value)
{
   nameValue=name;
   objectValue=value;
}
```

```
   NamedObject<int> no1("Smallest Prime number",2);
   NamedObject<int> no2(no1);   //calls copy constructor
```

کپی سازنده‌ای که توسط کامپایلر تولید میشود، می‌بایست no2.nameValue و no2.ObjectValue را توسط no1.nameValue و no1.ObjectValue آغازدهی کند. نوع nameValue رشته خواهد بود، و نوع رشته استاندارد کپی سازنده دارد، در این صورت no2.nameValue با فراخوانی کپی سازنده string آغازدهی یا initialize می‌شود، دقت کنید آرگومان کپی سازنده string همان no1.nameValue خواهد بود. از طرف دیگه، نوع NamedObject<int>::ObjectValue مشخصا int خواهد بود، و int یک نوع built-in هست، در این صورت no2.ObjectValue با کپی کردن بیتی no1.ObjectValue شروع به کار و یا initialize خواهد شد.

آیتم ۶: 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 می‌توننـد توابع خصوصی رو صدا بزنند. مگراین که شما به اندازه‌ی 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 صورت خصوصی تعریف کنیم و حواسمون جمع باشه که از اون‌ها خودمون استفاده نکنیم، در یک کتابخانه‌ی خیلی معروف مثلا کتابخانه iostream انجام شده.

به عنوان مثال می‌تونید یک نگاهی به نحوه‌ی پیاده‌سازی basic_ios ,ios_base و sentry توی پیاده‌سازی standat library داشته باشید. وقتی که این پیاده‌سازی‌هارو نگاه کنید متوجه می‌شوید که 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 قرار نیست که پیاده‌سازی بشن، پس چه نیازی به ذکر کردن اسامی پارامترها هست؟

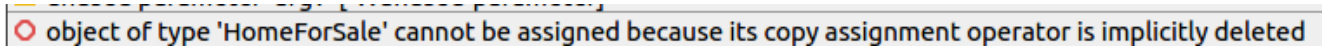
با پیاده سازی بالا، کامپایلر هیچ دسترسی‌ای به مشتری نمیده که بتونه شیء HomeForSale رو کپی کنه، و حتی اگه شما بخواین توی تابع عضو و یا یک تابع دوست از این توابع ممنوعه استفاده کنید، linker بهتون ارور میده. در خاطر داشته باشید که ما می‌تونیم خطایی که موقع link-time میگیریم رو به compile time ببریم(انجام این کار 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 نه در خود کلاس بلکه در کلاس base باید 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.

آیتم ۷: در کلاس‌های والد چندریختی ، مخرب را به صورت virtual تعریف کنید.

روش‌های زیادی وجود داره که بتونیم حساب کتاب زمان رو داشته باشیم، اما یک روش معقول این هست که یک کلاس base مثل TimeKeeper به همراه کلاس‌های مشتق شده ایجاد کنیم. در تکه کد زیر همچین موردی رو نوشتیم:

```
class TimeKeeper
```

```
{
public:
   TimeKeeper() {}
   ~TimeKeeper() {}
};
class AtmoicClock: public TimeKeeper{};
class WaterClock: public TimeKeeper{};
class WristWatch: public TimeKeeper{};
```

کد باید به صورتی نوشته بشه که مشتری‌ها هر وقت دوست داشتند به زمان دسترسی داشته باشند و نگران این نباشند که جزییات پیاده‌سازی به چه صورت است، در این صورت یک factory function (تابعی که اشاره‌گری از کلاس base به کلاس جدید مشتق شده رو برمیگردونه در واقع همون) می‌تونه برای برگرداندن یک اشاره‌گر به شیء timekeeping استفاده بشه(پس دقت کنید که getTimeKeeper یک factory function بود).

```
   TimeKeeper* getTimeKeeper();  // returns a pointer to a dynamically allocated
                                 // object of a class derived from TimeKeeper
```

همانطور که می‌دانید چون آبجکت به صورت داینامیک تعریف شده، شیءای که از getTimeKeeper برگشت داده شده روی heap بوده، بنابراین برای این که از نشت حافظه و هدر رفت سایر منابع جلوگیری کنیم، این خیلی مهمه که هر شیء که به این شکل هست delete بشه:

```
   TimeKeeper *ptk=getTimeKeeper();  //get dynamically allocated object from
TimeKeeper
                                     //hierachy
   //.... use it


   delete ptk;  //release it to avoid resource leak
```

آیتم ۱۳ نشان میدهد که انتظار delete کردن همچین چیزایی از مشتری خیلی خطرناکه:)، و آیتم ۱۸ نشان میدهد که چطور رابط factory function رو می‌توان تغییر داد تا از خطاهای رایجی که مشتری میگیرد جلوگیری شود، اما چنین چیزی الان اولویت نداره، توی این آیتم ما به دنبال پیدا کردن یک راه حل برای یک ضعف بنیادی از کد بالا هستیم: حتی اگه مشتری همه چیز رو درست انجام بده، هیچ تضمینی وجود نداره که بدونیم برنامه چطور کار خواهد کرد.

مشکل اینجاست که getTimeKeeper یک پوینتر به شیء کلاس مشتق شده برمیگرداند(مثلا AtmicClock)، که این شیء توسط اشاره‌گر کلاس base می‌تواند delete شود (یعنی اشاره‌گر **TimeKeeper**)، و کلاس base (یعنی **TimeKeeper**) مخرب non-virtual ندارد. این شرایط باعث بروز یک فاجعه خواهد شد، چون در C++ وقتی کلاس فرزند رو از طریق اشاره‌گر به کلاس والد delete می‌کنیم، و مخرب کلاس والد نیز به صورت non-virtual باشد، نتیجه اجرای کد نامشخص خواهد بود. عموما در صورت داشتن شرایط قبلی، در هنگام اجرای برنامه، قسمت مشتق شده از حافظه پاک نمیشه. اگر getTimeKeeper یک اشاره‌گر به شیء AtmoicClock برگردونه، قسمت AtmoicClock از شیء(یعنی، داده‌ی عضو که در کلاس AtmoicClock اعلان شده) احتمالا destroy نخواهد شد، یا اصلا مخرب AtmoicClock هرگز اجرا نخواهد شد. اگر چه، قسمت base class (یعنی قسمتی که به TimeKeeper مربوطه)معمولا destroy میشه، بنابراین منجر به حذف قسمتی از داده‌های یک شیء میشه. این یک روش خیلی عالی توی نشت منابع هست، از بین رفتن ساختمان داده، و در نتیجه کلی زمان برای دیباگ کردن کد از شما خواهد گرفت. راه حل رفع کردن این مشکل خیلی ساده است: توی کلاس base یک مخرب virtual اضافه کنیم. در این صورت حذف کردن شیء از کلاس مشتق شده دقیقا همان چیزی خواهد بود که شما می‌خواهید. در این صورت همه‌ی شیء حذف خواهد شد، که شامل همه‌ی قسمت‌های کلاس مشتق شده نیز خواهد شد.

```
class TimeKeeper
{
public:
   TimeKeeper() {}
   virtual ~TimeKeeper() {}
};



   TimeKeeper *ptk = getTimeKeeper();
   delete ptk; //now behaves correctly
```

کلاس‌های base ای مانند TimeKeeper عموما دارای توابع virtual دیگری غیر این مخربی که گفتیم هستند، چرا که هدف توابع virtual این هست که اجازه‌ی سفارشی سازی به کلاس‌های مشتق شده رو بدیم(برای جزییات بیشتر آیتم ۳۴ رو ببینید). به طور مثال، TimeKeeper ممکن است تابع virtual ای به نام getCurrentTime داشته باشد، که توی کلاس‌های مشتق شده ممکن است پیاده‌سازی‌های متفاوتی داشته باشد. هر کلاس با توابع virtual حتما باید مخرب virtual نیز داشته باشد.

اگر یک کلاس حاوی توابع virtual نباشد، معمولا نشانه‌ی این است که قرار نیست این کلاس به عنوان کلاس base استفاده شود. وقتی یک کلاس قرار نیست به عنوان کلاس base استفاده شود، این که مخرب رو به صورت virtual استفاده کنیم معمولا ایده‌ی بدی است. یک کلاس را در نظر بگیرید که برای بیان نقاط در دو بعد استفاده می‌شود:

```
class Point
{
public:
   Point(int xCoord,int yCoord);
   ~Point();

private:
   int x, y;
};
```

اگر int به اندازه‌ی 32bit حافظه اشغال کند، شیء Point می‌تواند عموما روی یک رجیستر ۶۴ بیتی جا بگیرد. علاوه بر این، چنین شیء می‌تواند به عنوان یک حافظه ۶۴ بیتی به توابع در سایز کلاس‌ها پاس داده شود، مثل C و FORTRAN . اگر مخرب Point به صورت Virtual بود، شرایط کاملا تغییر پیدا می‌کرد.

نحوه‌ی پیاده‌سازی توابع virtual باعث میشه که شیء نیازمند حمل اطلاعات در هنگام runtime بشه تا بشه تعیین کرد کدوم تابع virtual توی runtime قراره invoke بشه. این اطلاعات معمولا به فرم یک اشاره‌گر به نام vptr هستند(virtual table pointer). اشاره‌گر vptr به یک آرایه از فانکشن‌ها اشاره می‌کند که vtbl نامیده می‌شود(virtual table). هر کلاس با توابع virtual دارای vtbl همراه خواهد بود. وقتی یک تابع virtual بر روی یک شیء invoke میشه، تابعی که واقعا صدا زده میشه توسط دنبال کردن vptr به vtbl و سپس جستجوی اشاره‌گر تابع مناسب در vtbl پیدا میشه.

جزییات نحوه‌ی پیاده‌سازی توابع virtual واقعا مهم نیست. چیزی که مهمه این هست که اگه کلاس Point تو خودش توابع virtual داشته باشه، شیء از نوع Point افزایش سایز خواهد داشت. روی یک معماری ۳۲ بیتی، این اشیاء از ۶۴ بیت هم عبور خواهند کرد(۶۴ بیت به خاطر وجود دو تا int) و به ۹۶ بیت خواهند رسید(چون یک int یعنی vptr اضافه شده است)، اما بر روی یک معماری ۶۴ بیتی ممکن است که از ۶۴ بیت به ۱۲۸ بیت برسند، چون که اشاره‌گر روی چنین سیستم‌هایی ۶۴ بیتی است. با اضافه شدن vptr به Point اندازه‌ی شیء ۱۰۰ درصد اضافه شد. در این صورت دیگه یک Point نمی‌تونه روی یک رجیستر ۶۴ بیتی جا بگیره. علاوه بر این، دیگه شیء C++ شبیه ساختار structure اعلان شده

در یک زبان دیگر مثل C نخواهد بود، چون زبان‌های دیگر فاقد vptr هستند. در نتیجه، دیگه قادر نخواهیم بود که Point رو برای زبان‌های دیگه ارسال کنیم.

ذکر این نکته خالی از لطف نیست که اعلان همه‌ی مخرب‌گرها به صورت virtual به همان اندازه‌ی اعلان به صورت non-virtual اشتباهه. در حقیقت: " وقتی باید از virtual destructor استفاده شود که کلاس دارای حداقل یک تابع به صورت virtual باشد" .

این امکان وجود داره که مساله‌ی استفاده از non-virtual destructor حتی در صورتی که هیچ فانکشن virtual ای نداشته باشید هم گریبان‌تون رو بگیره. به طور مثال، string استاندارد هیچ تابع virtualای ندارد، ولی برخی برنامه‌نویسان گمراه شده ممکنه از این کلاس به عنوان کلاس base استفاده کنند.

```
class SpecialString: public std::string // bad idea!
                    //has a non-virtual destructor
{
};
```

در نگاه اول، کد بالا ممکنه هیچ مشکلی نداشته باشه، ولی اگر یک جایی از برنامه یک اشاره‌گر به SpecialString رو به اشاره‌گری به string تبدیل کنید و اشاره‌گر string رو delete کنید، معلوم نخواهد بود که کد چطور رفتار خواهد کرد.

```
SpecialString *pss=new SpecialString("Impending Doom");
std::string *ps;
ps=pss;                 //SpecialString* --> std::string*
delete ps;    //undefined! in practice,*ps's Special resources
          // will be leaked, becuase the SpeciaString destructor
          // won't be called.
```

همین تحلیل در مورد همه‌ی کلاس‌های که virtual destructor ندارند درست است، که شامل همه‌ی container های STL مانند vector,list,set,tr1::unordered_map نیز هست. هر گاه شما مشتاق شدید که از این نگه‌دارنده‌ها و یا هر کلاس دیگری ارث‌بری کنید در صورتی که non-virtual destructor داشت، در مقابل خواسته‌تون مقاومت کنید.(متاسفانه ، در C++ هیچگونه مکانیزمی برای مقابله با این چنین مشتقاتی وجود ندارد که در جاوا و C# وجود دارد).

در برخی مواقع، نیاز داریم تا به کلاس یک pure virtual destructor بدیم. به یاد بیاورید که pure virtual function در نتیجه‌ی کلاس‌های abstract بود(کلاس‌هایی که نمی‌توانند instnatiate بشن(یعنی نمی‌تونید ازشون شیء بسازید)). ولی در برخی مواقع، شما کلاسی دارید که دوست دارید که

abstract بشه، ولی شما هیچ تابع pure virtual ای ندارید. در این صورت چیکار می‌کنید؟ خب، از اونجایی که هدف کلاس abstract این بوده که به عنوان یک کلاس base استفاده بشه، و چون حتما یک کلاس base باید virtual destructor داشته باشه، و چون یک pure virtual function منجر به یک abstract class میشه راه حل ساده است: یک pure virtual destructor در کلاسی که می‌خواید abstract بشه اعلان کنید. در اینجا یک مثال در این مورد خواهیم دید.

```
class AWOV{    //AWOV="Abstract w/o virtuals
public:
   virtual ~AWOV()=0;   //declare pure virtual destructor
};
```

این کلاس یک pure virtual function داره، پس یک abstract می‌باشد، و یک virtual destructor هم داره، در این صورت نیازی نیست نگران مشکلات مربوط به non-virtual destructor باشیم. فقط یک نکته وجود داره، شما باید یک تعریف برای pure virtual destructor داشته باشید.

```
AWOV::~AWOV(){}   //definition of pure virtual
```

destructor به این صورت کار می‌کند که destructor آخرین کلاس مشتق شده اول فراخوانی می‌شود، و سپس destructor مربوط به کلاس‌های base فراخوانی می‌شود. کامپایلرها یک فراخوانی به AWOW~ از مخرب‌های کلاس‌های مشتق شده تولید می‌کنند، در این صورت باید اطمینان حاصل کنید که یک بدنه برای این تابع در نظر گرفته‌اید. اگر این کار رو انجام ندید، linker در این مورد خطا میده. قانونی که میگه باید برای کلاس‌های base یک virtual destructor بدید تنها در مورد کلاس‌های چندریختی base استفاده داره(کلاس‌های base ای که برای این طراحی شده‌اند که به عنوان رابط برای کلاس‌های دیگر استفاده شود). به طور مثال TimeKeeper یک کلاس چندریختی base هست، چرا که ما انتظار داریم که بتونیم اشیاء AtomicClock و WaterClock رو تغییر بدیم، چرا که ما تنها TimeKeeper رو برای اشاره به اون‌ها در اختیار داریم.

توجه شود که تمام کلاس‌های base برای این طراحی شده‌اند که به عنوان یک کلاس چندریختی استفاده شوند. بنابراین string و نگه‌دارنده‌های STL برای این طراحی نشده‌اند که به عنوان کلاس base استفاده شوند. برخی کلاس‌ها برای این طراحی شده‌اند که به عنوان کلاس base استفاده شوند، و به صورت چندریختی نیستند. چنین کلاس‌هایی مثل Uncopyable از آیتم ۶ و input_iterator_tag از STL (آیتم ۴۷ را ببینید)، که به صورتی طراحی نشده‌اند که اجازه‌ی تغییر دادن در کلاس‌های مشتق شده را از طریق رابط داشته باشند. در نتیجه آن‌ها دارای مخرب virtual نیستند.

Factory چیست؟

طراحی factory در شرایطی مفید است که نیاز به ساخت اشیاء زیاد با تایپ‌های متفاوت هستیم، همه‌ی کلاس‌های مشتق شده از یک base هستند. متد factory یک متد برای ساختن اشیاء تعریف می‌کند که در آن یک زیر کلاس می‌تواند نوع کلاس ساخته شده را مشخص کند. بنابراین، در روش factory در لحظه اجرای کد، اطلاعاتی که هر شیء می‌خواهد را بهش پاس میدهد(به طور مثال، رشته‌ای که توسط کاربر گرفته می‌شود) و یک اشاره‌گر از کلاس base به نمونه‌ی جدید ساخته شده برمیگرداند. این روش در موقعیتی بهترین خروجی را میدهد که رابط class base به بهترین نحو طراحی شده باشد، بنابراین نیازی به case شیء برگردان شده نخواهیم داشت.

مشکلی که برای طراحی خواهیم داشت چیست؟

ما می‌خواهیم که در هنگام اجرای برنامه تصمیم بگیریم که بر اساس اطلاعات برنامه و یا user چه شیءای باید ساخته شود. خب در هنگام نوشتن کد ما که نمی‌دونیم که user چه اطلاعاتی را وارد خواهد کرد در این صورت چطور باید کد این را بنویسیم.

راه حل!!

یک رابط برای ساخت شیء طراحی می‌کنیم، و اجازه میدهیم که رابط تصمیم بگیرد که کدوم کلاس باید ساخته شود.

در مثالی که در ادامه آورده‌ایم، روش factory برای ساخت شیء laptop و desktop در runtime استفاده می‌شود. اجازه بدهید یک کلاس base به نام computer تعریف کنیم، که یک کلاس تجریدی base هست(به عنوان رابط) و کلاس‌های مشتق شده Laptop و Desktop هستند.

```
class Computer
{
public:
   virtual void run()=0;
   virtual void stop()=0;
   virtual ~Computer(){}
};
class Laptop: public Computer
{
public:
```

```
    void run() override  {m_Hibernating = false;}
    void stop() override {m_Hibernating = true;}
    virtual ~Laptop(){}       // becuase we have virtual functions, we need virtual
destructor
private:
    bool m_Hibernating;
};
class Desktop : public Computer
{
public:
    void run() override {m_ON=true;}
    void stop() override{m_ON=false;}
    virtual ~Desktop(){}
private:
    bool m_ON;
};
```

کلاس زیر برای تصمیم گیری در این مورد ساخته شده است.

```
class ComputerFactory
{
public:
    static Computer *NewComputer(const std::string &description)
    {
        if(description=="laptop")
            return new Laptop;
        if(description=="desktop")
            return new Desktop;
        return nullptr;
    }
```

```
};
```

بیایید مزیت این طراحی را با همدیگه آنالیز کنیم. اول این که، چنین طراحی‌ای مزیت کامپایلی داره. اگر ما رابط Computer و factory را به هدر فایل دیگری منتقل کنیم، می‌توانیم پیاده‌سازی NewComputer را به یک فایل پیاده‌سازی دیگر منتقل کنیم. در این صورت پیاده‌سازی تابع () NewComputer تنها کلاسی است که نیاز به اطلاعات در مورد کلاس‌های مشتق شده دارد. بنابراین، اگر هر تغییری بر روی کلاس‌های مشتق شده از Computer انجام پذیرد، تنها فایلی که نیاز به کامپایل دوباره دارد NewComputer است. هر کسی که از factory استفاده می‌کند تنها باید نگران رابط باشد، که در طول اجرای برنامه هم ثابت است.

همچنین، اگر نیازی به اضافه کردن یک کلاس داشته باشیم، و کاربر برای اشیایی که می‌خواد از رابط استفاده کنه، کدی که factory رو فراخوانی می‌کنه نیازی به تغییر نداره. کدی که از factory استفاده می‌کنه تنها یک رشته به رابط میده و شیء رو پس میگیره، و این موضوع اجازه میده که تایپ‌های جدید رو توسط همین factory پیاده‌سازی کنیم.

Abstract class چیست؟

یک کلاس abstract کلاسی است که برای این طراحی شده که به عنوان base class استفاده شود. یک abstract class حداقل یک pure virtual function خواهد داشت. شما می‌توانید چنین تابعی را با استفاده از pure specifier(=0) در اعلان عضو virtual ایجاد کنید.

```
class AB{
public:
   virtual void f()=0;
};
```

در اینجا، AB::f یک pure virtual function خواهد بود. یک فانشکن pure نمی‌تواند هم اعلان داشته باشد هم تعریف. به طور مثال، کامپایلر هرگز اجازه‌ی کامپایل کد زیر را نخواهد داد.

```
struct A{
   virtual void g(){}=0;
};
```

کاربرد استفاده از abstract class چیست؟

همانطور که قبلا بیان شد برای طراحی interface استفاده می‌شود.

مثال از abstract class

```
// Base class
class Shape {
public:
   // pure virtual function providing interface framework.
   virtual int getArea() = 0;
   void setWidth(int w) {
      width = w;
   }
   void setHeight(int h) {
      height = h;
```

```
   }
protected:
   int width;
   int height;
};
// Derived classes
class Rectangle: public Shape {
public:
   int getArea() {
      return (width * height);
   }
};
class Triangle: public Shape {
public:
   int getArea() {
      return (width * height)/2;
   }
};
int main(void) {
   Rectangle Rect;
   Triangle  Tri;
   Rect.setWidth(5);
   Rect.setHeight(7);
   // Print the area of the object.
   cout << "Total Rectangle area: " << Rect.getArea() << endl;
   Tri.setWidth(5);
   Tri.setHeight(7);
   // Print the area of the object.
```

```
    cout << "Total Triangle area: " << Tri.getArea() << endl;
    return 0;
}
```

Item 8 : Prevent exceptions from leaving destructors

C++ رخ دادن exceptions رو توی مخرب‌هارو منع نکرده، ولی قطعا استفاده ازش دلسرد کننده خواهد بود. برای این موضوع دلایل خوبی را خواهیم آورد، مثال زیر رو ببینید:

```
class Widget
{
public:
    ~Widget() {}    //assume this might emit an exception
};
void so_something()
{
    std::vector<Widget> v;
}                       // v automatically destroyed here
```

وقتی که بردار v نابود میشه، مسوول این هستش که هر چیزی که Widget داره رو هم نابود کنه. در نظر بگیرید که v دارای ده تا Widget باشه، و در طی destruction شیء اول، یک exception رخ بده. در این صورت همه‌ی Widget های دیگر هنوز نابود نشده‌اند(در غیر این صورت منابعی که نگه داشته‌اند به عنوان leak شناخته می‌شود)، بنابراین v باید destructor نه تای باقی مانده را invoke کند. حال فرض کنید که در طی فراخوانی دومی هم یک exception رخ بده. در این صورت دو تا exception فعال توی برنامه مون داریم و همین یه دونه اضافه هم برای C++ خیلی به حساب میاد. بر اساس این که دقیقا تحت چه شرایطی این دو تا استثنا به وجود اومده ممکنه برنامه terminate بشه و یا رفتار نامشخص داشته باشه. در مورد مثال خودمون رفتار نامشخص نتیجه‌ی همچین چیزی خواهد بود. در مورد نگه‌دارنده‌های STL رفتار نامشخص را خواهیم گرفت و در مورد نگه دارنده‌های TR1 (آیتم ۵۴ رو ببینید) و یا حتی array نیز به همین صورت است. در این صورت C++ قرار دادن exception رو توی destructor رو به هیچ وجه دوست نداره.

فهم مفهومی که گفتیم سخت نیست، ولی چطور یک مخرب بسازیم که نیاز به اجرای یک عملیات داره و ممکنه این عملیات نیز fail بشه؟ به طور مثال فرض کنید که شما دارید روی یک کلاس برای ارتباط دیتابیس کار می‌کنید:

```
class DBConnection{
public:

   static DBConnection create();  //function to return DBConnection
                                  //objects; params ommited for simplicity
   void close(); //close connection;throw an exception if closing fails
};
```

برای این که مطمعن بشیم که که مشتری‌های کد، فراموش نمی‌کنند که یک شیء Dbconnection رو close کنند، یک روش عقلانی این است که یک مدیریت منابع برای DBConnection بنویسیم که متد close رو توی مخرب کلاس فراخوانی کنه. چنین مدیریت منابعی به طور مفصل در فصل سوم مورد بررسی خواهد گرفت، اما در اینجا، ما صرفا به بررسی مخرب برای چنین کلاسی خواهیم پرداخت:

```
class DBconn{       //class to manage DBConnection
public:             //objects
   ~DBconn()        //make sure database connections
   {                //are always closed
      db.close();
   }
private:
   DBConnection db;
};
```

این به برنامه‌نویس اجازه میده که یک کد مثل زیر رو بنویسه.

```
{                                                  //open a block
   DBconn dbc(DBConnection::create());       // create DBConnection object
```

```
                                        //and turn it over to a DBCon object to manage
                                        //use the DBConnection object via the DBconn
interface



}                       //at the end of block, the DBconn object is destroyed, thus
automatically
                        // calling close on the DBConnection object
```

این کد در صورتی که close با موفقیت انجام بشه، مشکلی نداره، اما فرض کنید که در این کد یک استثناء رخ بده، در این صورت ما توی مخرب این کلاس مشکل داریم. دو روش برای جلوگیری کردن از این مشکل وجود داره.

یکی این که وقتی close به ما استثناء داد برنامه رو terminate کنیم، که این کار معمولا با فراخوانی abort انجام میشه

```
    try{db.close();}
    catch(...)
    {
       //make log entry that the call to close failed;
       std::abort();
    }
```

این کاری که کردیم در صورتی که برنامه نتونه بعد از این اجرا بشه منطقیه. و این مزیت رو داره که، اگه اجازه بدیم که یک exception از مخرب صادر بشه ممکنه که به یک رفتار undefined برخورد کنیم، که این کار از این مشکل جلوگیری می‌کنه. در این صورت فراخوانی کردن abort ممکنه از undefined behavior جلوگیری کنه.

روش دوم بلعیدن exception در هنگام فراخوانی کردن close است.

```
    try{db.close();}
    catch(...)
    {
```

```
        //make log entry that the call to close failed;
    }
```

عموما، بلعیدن exception ایده بدی است، چون در این مورد اطلاعات مفید رو از دست میدیم – یه چیزی اشتباهه، اما بلعیدن exception به این که برنامه terminate بشه و یا دچار undefined behavior بشه ارجحیت داره. برای این که این راه حل یک راه حل قابل اعتماد باشه، باید اطمینان داشته باشیم که برنامه پس از این که وارد exception میشه حتما بتونه ادامه‌ی برنامه رو بره.

مشکلی که در مورد این دوتا روش گفته شده داریم این است که راهی برای برخورد با این شرایط وجود ندارد که close منجر به exception میشه.

یک استراتژی بهتر این است که رابط DBConn رو به نحوی طراحی کنیم که مشتری این امکان رو داشته باشه که بر اساس مشکلی که وجود داره یکی رو انتخاب کنه. به طور مثال، DBConn می‌تونه یک تابع رو پیشنهاد بده که، که به مشتری این شانس رو میده که بتونه استثناء به وجود اومده رو تصحیح کنه. این تابع می‌تونه بفهمه که مشکل از کجا بوده، خودش رو توی destructor ببنده یا نبنده. این به ما این اجازه میده رو که ارتباط leak نداشته باشه. در هر صورت اگر close به ما دوباره یک استثنا بده می‌تونیم برنامه رو terminate کنیم و یا این که اون رو ببلعیم.

```
class DBconn{
public:
   DBconn(DBConnection){}
   void close()
   {
      db.close();
      closed=true;
   }
   ~DBconn()
   {
      if(!closed)
      {
         try {
```

```
            db.close();
        } catch (...) {
            //make log entry that call to close friend
        }
    }
}
private:
    DBConnection db;
    bool closed;
};
```

Moving the responsibility for calling close from DBConn 's destructor to DBConn 's client (with DBConn 's destructor containing a “backup” call) may strike you as an unscrupulous shift of burden. You might even view it as a violation of Item 18's advice to make interfaces easy to use correctly. In fact, it's neither. If an operation may fail by throwing an exception and there may be a need to handle that exception, the exception has to come from some non-destructor function. That's because destructors that emit exceptions are dangerous, always running the risk of premature program termination or undefined behavior. In this example, telling clients to call close themselves doesn't impose a burden on them; it gives them an opportunity to deal with errors they would otherwise have no chance to react to. If they don't find that opportunity useful (perhaps because they believe that no error will really occur), they can ignore it, relying on DBConn 's destructor to call close for them. If an error occurs at that point — if close does throw — they're in no position to complain if DBConn swallows the exception or terminates the program. After

all, they had first crack at dealing with the problem, and they chose not to use it.

Item 9: Never call virtual functions during construction or destruction

نباید توابع virtual رو طی construction و یا destruction فراخوانی کنیم، چرا که نمیشه فهمید منظور شما از این فراخوانی چی بوده، و حتی اگر این کارو بتونه انجام بده، باز هم از نتیجه ناراضی خواهیم بود.

فرض کنید که شما یک مدل برای تراکنش‌های انبارداری نوشتید، یعنی سفارشات خرید، سفارشات فروش و غیره. این خیلی مهمه که چنین تراکنشاتی قابل حسابرسی باشه، بنابراین هر زمان که یک شیء تراکنش ایجاد میشه، یک log مناسب باید برای حسابرسی ساخته بشه. در این صورت یک روش معقول برای انجام این کار کد زیر است:

```
class Transaction
{
public:
   Transaction();
   virtual void logTransaction() const=0;  //make type-dependent log entry
};
Transaction::Transaction()
{
   logTransaction();
}
class BuyTransaction: public Transaction{   //derived class
public:
   virtual void logTransaction() const;  //how to log transactions of this type
};
class SellTransaction: public Transaction{   //derived class
public:
```

```
    virtual void logTranstion() const;   //how to log transactions of this type
};
```

ببینید چه اتفاقی میفتد وقتی کد زیر رو اجرا می‌کنید.

```
BuyTransaction b;
```

مشخصا سازنده‌ی کلاس BuyTransaction در این مورد فراخوانی خواهد شد، اما قبل از اون، باید سازنده کلاس **Transaction** صدا زده شود. قاعده‌ی کلی به این صورت است که قسمت‌های base class مربوط به کلاس‌های مشتق شده زودتر فراخوانی می‌شوند. خط آخر از سازنده کلاس **Transaction** منجر به فراخوانی ***logTransaction*** خواهد شد که virtual هست، اینجا جایی است که ممکنه در موردش تعجب کنید.

ورژن logTransaction ای که فراخوانی می‌شود مربوط به کلاس Transaction می‌باشد نه ورژنی که در BuyTransaction هست(حتی اگه شیء از Buytransaction ساخته شده باشه). طی ایجاد کلاس base ، توابع virtual هرگز وارد کلاس‌های مشتق شده نمی‌شوند. به جای این کار، شیء به نحوی رفتار می‌کند که انگار همان کلاس base هست. به بیان ساده‌تر، وقتی که کلاس base در حال ایجاد و ساخته شدن هست، توابع virtual وجود ندارند.

دلیل این رفتار هم خیلی منطقی به نظر میرسه چون سازنده کلاس base قبل از سازنده کلاس‌های مشتق شده اجرا می‌شود، در این صورت وقتی که سازنده‌ی کلاس base در حال اجرا هست داده‌های عضو مربوط به کلاس‌های مشتق شده هنوز مقداردهی اولیه نشده‌اند. اگر هنگام اجرای سازنده‌ی کلاس base یک تابع virtual اجرا بشه، در این صورت ما نمی‌توانیم وارد کلاس‌های مشتق شده بشیم، چون اگر این کار رو بکنیم داریم به اعضای داده‌ی کلاس نیز اشاره می‌کنیم در حالی که این اعضاء هنوز initialized نشده‌اند. اگر چنین اشتباهی را انجام بدید، ممکنه ساعت‌ها صرف دیباگ کردن کدتون بکنید. در این صورت، فراخوانی اعضای پایین دستی از یک شیء که هنوز initialized نشده‌اند عمل خیلی خطرناکیه، به همین خاطر C++ راهی برای انجام دادن چنین کاری رو پیشنهاد نداده.

در واقع این موضوع بنیادی‌تر از این حرفاست. در طی ساختن کلاس base از یک کلاس مشتق شده، نوع شیء همان کلاس base هست. در واقع این نقطه نظر تنها از دیدگاه توابع virtual نیست، بلکه برخی از قسمت‌های زبان که از اطلاعات runtime استفاده می‌کنند مثل dynamic_cast (آیتم ۲۷) و tipeid شیء را از نوع کلاس base می‌بینند. در مورد مثال ما، وقتی که سازنده کلاس Transaction در حال اجرا برای initialize کردن قسمت buytransaction ، نوع شیء همان Transaction می‌باشد. این نحوه‌ی برخورد زبان C++ با آن است که منطقی هم به نظر میرسد: چرا که هنوز قسمت‌هایی از BuyTransaction هنوز initialize نشده‌اند، پس امن‌ترین راه برای برخورد با چنین موردی این هست که فرض کنیم اصلا چنین چیزی وجود خارجی ندارد. یک شیء تا وقتی که سازنده‌ی کلاس مشتق شده اجرا نشود به کلاس مشتق شده تبدیل نمی‌شود و base کلاس محسوب می‌شود.

همین منطق در هنگام اجرای مخرب کلاس نیز پا برجاست. وقتی که مخرب یک کلاس مشتق شده اجرا می‌شود، عضو داده شیءای که از یک کلاس مشتق شده ساخته شده، تعریف نشده فرض می‌شوند، بنابراین C++ به نحوی با آن‌ها برخورد می‌کند که انگار وجود ندارند. و به محض این‌که وارد مخرب کلاس base می‌شوند، شیء تبدیل به شیء کلاس base شده، و همه‌ی قسمت‌های مختلف زبان C++ (توابع virtual و dynamic_cast و غیره) به همین نحو برخورد می‌کنند.

در مثال بالا، سازنده کلاس Transaction یک فراخوانی مستقیم به تابع virtual خواهد داشت، که مشاهده‌ی اشتباهی که انجام شده در این مثال نیز ساده هست، برخی از کامپایلرها در این مورد به ما یک هشدار میدهند(و برخی دیگر این هشدار را نمیدهند، آیتم ۵۳ رو برای این موضوع ببینید). حتی اگه چنین خطایی را نمیگیرفتیم، مشکل قبل از runtime نیز قطعی است، چرا که تابع logTransaction یک تابع pure هست. مگر این که تعریفش کرده باشیم(بله می‌تونیم حتی چنین کاری رو نیز انجام بدهیم آیتم ۳۴ رو ببینید)، در این صورت برنامه نمیتونه link بشه، چون که لینکر نمی‌تونه پیاده سازی مناسب رو برای Transaction::logTransaction پیدا کنه.

در برخی مواقع تشخیص فراخوانی به توابع virtual هنگام تخریب و یا اجرای سازنده چندان هم ساده نیست. اگر Transaction چندین سازنده داشته باشد، که هر کدام میبایست یک کاری رو انجام بدهند، به لحاظ طراحی بهتره کد رو طوری بنویسیم که از دوباره کاری توی کد بپرهیزیم، این کار رو می‌تونیم با کد یکسان initialization و موارد دیگر مرتفع کنیم، فرض کنید برای این مورد یک تابع به نام init نوشته باشیم:

```
class Transaction
{
public:
   Transaction()
   {
      init();    //call to non-virtual
   }
   virtual void logTransaction() const=0;
private:
   void init()
   {
```

```
    logTransaction(); //that calls a virtual
  }
};
```

این کد به لحاظ مفهومی مشابه همون کد اول هست، ولی یه مقدار پیچیده تره، چون معمولا کامپایل میشه و بدون مشکلی link میشه. در این مورد، چون logTransaction یک virtual pure در کلاس Transaction هست، بیشتر runtime system ها برنامه را هنگام فراخوانی virtual pure متوقف و یا abort خواهند کرد. خطایی که در سیستم عامل linux به من داده به این صورت هست.

```
pure virtual method called
terminate called without an active exception
```

حالا اگر logTransaction یک تابع virtual عادی بود(یعنی pure نبود) که یک پیاده سازی هم در Transaction داشت، این ورژن فراخوانی میشد.(در کامپایلرهای جدید اجازه‌ی چنین کامپایلی به ما داده نمیشه). تنها راه برای حل این مشکل این هست که اطمینان پیدا کنیم که هیچکدام از سازنده‌ها و مخرب‌های شما نمی‌توانند توابع virtual رو فراخوانی بکنند.

ولی چطور می‌تونید اطمینان حاصل کنید که ورژن درستی از logTransaction در موقع ساخت شیء فراخوانی شده است یا نه؟ واضح است که فراخوانی یک تابع virtual بر روی شیء از سازنده‌ی کلاس Transaction یک راه اشتباه برای انجام این کار است.

راه‌های متفاوتی برای مرتفع کردن این مشکل وجود دارد. یک راه تبدیل logTransaction به یک تابع non-virtual در Transaction هست، سپس نیازه که سازنده‌ی کلاس‌های مشتق شده اطلاعات لازم log رو برای سازنده‌ی Transaction بفرستند. این تابع می‌تواند به صورت امنی non-virtual logTransaction رو فراخوانی کنه. مثل این مورد:

```
class Transaction
{
public:
   explicit Transaction(const std::string& logInfo);
   void logTransaction(const std::string& logInfo) const; //now a non-virtual function

};
Transaction::Transaction(const std::string& logInfo)
```

```
{
   logTransaction(logInfo);
}
class BuyTransaction: public Transaction
{
public:
   BuyTransaction(parameters)
      :Transaction(createLogString(parameters))
private:
   static std::string createLogString(parameters);
};
```

به عبارت دیگر، از اونجایی که شما نمی‌تونید از توابع virtual پایین دستی در کلاس base استفاده کنید(در هنگام ساختن کلاس base)، در این صورت می‌تونید این مورد رو این طوری جبران کنید که اطلاعات لازم رو از کلاس‌های مشتق شده به کلاس base بفرستید.

در این مثال، به استفاده از تابع createLogString که به صورت private static هست توجه کنید. استفاده از یک تابع کمکی برای ساخت و فرستادن مقدار به کلاس سازنده معمولا راه حل مناسب‌تری است. با static کردن تابع، دیگر خطر اشاره به اشیاء BuyTransaction که هنوز initalize نشده‌اند وجود ندارد. این خیلی مهم است، به خاطر این واقعیت که این اعضاء داده‌ای در حالت undefined قرار دارند، و این همان دلیلی است که چرا فراخوانی یک تابع virtual در سازنده‌ی کلاس base نمی‌تواند وارد کلاس‌های مشتق شده شود.

Item 10: Have assignment operators return a reference to *this

یکی از جالب توجه‌ترین ویژگی‌های assignment این است که شما می‌توانید زنجیره‌ای از آن‌ها را داشته باشید.

```
   int x,y,z;
   x=y=z=15;
```

همچنین در نظر داشته باشید که انتساب یک عمل راست به چپ هست، بنابراین انتساب‌های بالا به صورت زیر parse خواهند شد:

```
x=(y=(z=15));
```

در اینجا ۱۵ به z انتساب داده می‌شود، سپس نتیجه این انتساب به y انتساب می‌شود و نتیجه‌ی آن هم به x انتساب داده می‌شود.

راهی که این انتساب پیاده‌سازی شده این است که انتساب یک رفرنس به آرگومان سمت چپ برمیگرداند، و این رویه رو هم شما بهتره برای پیاده‌سازی کلاس‌هاتون در نظر بگیرید.

```
class Widget{
public:
   Widget& operator=(const Widget& rhs)  //return type is a refrence to the current class
   {
      return *this;   //return the left-hand object
   }
};
```

چنین رویکردی برای همه‌ی اپراتورهای انتساب قابل تعمیم هست، نه فقط اپراتوری که در بالا دیدیم.

```
class Widget{
public:
   Widget& operator+=(const Widget& rhs) //the convention applies to +=,-=,*=,etc.
   {
      return *this;
   }
   Widget& operator=(int rhs)
   {
      return *this;
   }
};
```

در C++ این یک اجماع نظر است، کدی که از این رویه طبعیت نکند نیز کامپایل خواهد شد. اگر چه، این رویکرد برای همه‌ی تایپ‌های built-in رعایت شده، برای stl ها نیز همینطور هست. مگر این که شما دلیل بهتری برای یک رویکرد متفاوت داشته باشید.

آیتم ۱۳ از فصل ۳

منابع یا resource چیزی است که، وقتی که نیازی بهشون ندارید باید آزادشون کنید و به سیستم برشون گردونید. اگر این کار رو نکنید، اتفاقات بدی میفته. در برنامه‌های C++ ، معمول‌ترین منبعی که وجود داره، اختصاص حافظه به صورت داینامیک هست(اگر حافظه‌ای را اختصاص دهید و هرگز آن را deallocate نکنید، در این صورت نشت حافظه خواهید داشت)،توجه داشته باشید که حافظه تنها یکی از منابعی است که شما باید مدیریت کنید. برخی منابع رایج از سیستم عبارتند از ، file descriptors fonts ، mutex locks و brush ها در رابط کاربری، ارتباط دیتابیس و سوکت‌های شبکه می‌باشد. صرف نظر از این که منبع چه باشد، این مهمه که وقتی دیگر با آن منبعی کاری نداریم آن را آزاد کنیم.

تلاش برای اطمینان از این موارد در هر شرایطی سخت می‌باشد، حال فرض کنید مواردی دیگری وجود داشته باشند که این شرایط را برایمان سخت تر کنند مانند exceptions ، توابع با چندین مسیر return ، و نگه‌داری تغییرات برنامه‌نویسان بر روی نرم‌افزار بدون این که درک مناسبی از تغییراتی که داده اند داشته باشیم، واضح هست که روش‌های این چنینی برای برخورد با مدیریت منابع کافی نخواهد بود.

در این فصل ما یک رویکرد مستقیم بر اساس **شیء** برای مدیریت منابع بر روی سازنده، مخرب و اپراتورهای کپی C++ خواهیم داشت.

Item 13: Use objects to manage resources.

فرض کنید که ما روی یک کتابخانه به منظور مدل کردن یک سرمایه‌گذاری کار می‌کنیم، که در آن سرمایه‌گذاری های مختلف از یک کلاس base به نام Investment ارث بری کرده‌اند.

```
class Investment   //root class of hierachy of investment types
{
};
```

علاوه بر این فرض کنید که این کلاس برای تهیه‌ی یک Investment خاص از طریق یک factory function عمل می‌کند(برای اطلاعات بیشتر فصل هفتم رو ببینید).

```
Investment*  createInvestment(); //return ptr to dynamically allocated
                                 //object in the investment hierarchy;
                                 //the caller must delete it
                                 //(parametes omitted for simplicity)
```

همانطور که کامنت کد بالا اشاره کرده، کسی که CreateInvestment رو فراخوانی کرده مسوول حذف شیء برگردان شده است. حال در نظر بگیرید که یک تابع به نام f برای انجام چنین کاری نوشته شده است.

```
Investment *pInv=createInvestment(); //call factory function

...                                  //use pInv

delete pInv;                         //release Object
```

این به نظر مشکلی نداره، اما چندین احتمال هست که f نتونه شیء investment رو که از createInvestment گرفته شده، نتونه حذف کنه. یکی این که ممکنه یک return زودهنگام در داخل تابع وجود داشته باشه. اگر چنین return ای اجرا بشه، در این صورت هرگز خط مربوط به delete کردن اجرا نخواهد شد. یک مشکل مشابه وقتی است که استفاده از createInvestment و delete در داخل یک حلقه باشه، و حلقه با استفاده از break و یا goto شکسته بشه و هرگز به delete نرسیم. در نهایت، ممکنه کد وارد یک exception بشه، در این صورت نیز control هرگز به delete نخواهد رسید. صرف نظر از این که چرا delete اجرا نشه، ما نه تنها بر روی حافظه‌ای که شیء investment گرفته نشت داشتیم بلکه هر منبعی که این شیء گرفته نیز نشت دارد.

قطعا، اگر درست برنامه‌نویسی کنیم و محتاط باشیم می‌تونیم از چنین مشکلاتی دوری کنیم، اما فرض کنید که این کد قراره در گذر زمان عوض بشه. طی این مرحله که نرم‌افزار در حال نگه‌داری است فرض کنید که یک نفر بیاد و یک return به کد اضافه کنه و یا continue به کد اضافه کنه بدون این که در مورد مدیریت منابع دقت کنه، و یا حتی بدتر، ممکنه داخل تابع f یک تابعی فراخوانی شده باشه که هرگز به exception نمی‌خورده ولی یک دفعه شروع کنه به exception خوردن. بنابراین نمی‌تونیم به f در مورد این که حتما منابع رو delete می‌کنه اطمینان داشته باشیم. برای اطمینان از این که منابعی که createInvestment گرفته همواره آزاد خواهند شد، نیاز داریم تا منابع را در داخل مخرب شیء قرار دهیم تا وقتی که کارمان با f تمام شد و مخرب صدا زده شد، اون منابع نیز حذف بشه. در واقع با قرار دادن منابع در داخل شیء ما توانسته‌ایم که روی این ویژگی زبان C++ تکیه کنیم که مخرب همواره صدا زده میشه.

بسیاری از منابع به صورت داینامیک بر روی حافظه‌ی heap رزرو شده‌اند، و بر روی یک block تنها و یا یک تابع استفاده می‌شوند، و می‌بایست وقتی که کنترل block و یا تابع رو گذر کرد اون قسمت از حافظه رها بشه. auto_ptr از کتابخانه‌ی استاندارد برای چنین شرایطی ساخته شده است. auto_ptr یک شیء شبه اشاره‌گر بوده(smart pointer) که destructor آن به صورت اوتوماتیک به چیزی که اشاره به آن شده را delete می‌کند. در اینجا نحوه‌ی استفاده از auto_ptr را برای جلوگیری از leak احتمالی در تابع f را توضیح داده‌ایم.

```
#include <memory>


   std::auto_ptr<Investment> pInv(createInvestment());
```

این مثال ساده دو جنبه‌ی خیلی مهم از استفاده شیء برای مدیریت منابع را نشان میدهد:

- **منابع بلافاصله به شیء مدیر منبع داده می‌شود.** در کد بالا، منبعی که توسط createInvestment برگردان شده برای initialize کردن auto_ptr استفاده می‌شود که آن را مدیریت خواهد کرد. در واقع، این ایده که برای مدیریت منابع از اشیاء استفاده بشه معمولا Resource Acquisition Is Initialization نامیده میشود(به اختصار RAII)، چرا که منطقیه در یک عبارت هم منابع رو بگیریم و هم شیء مدیریتمون رو initialize کنیم. البته در برخی موارد منابع گرفته شده را بعدا به شیء مدیریت منابع انتساب می‌کنیم.
- **شیء مدیر-منبع از مخرب خود برای اطمینان از آزاد شدن منابع استفاده می‌کند.** چرا که مخرب‌ها به صورت اوتوماتیک بعد از نابود شدن شیء فراخوانی می‌شوند(یعنی وقتی یک شیء از scope خارج می‌شود)، در این صورت منابع به صورت مناسبی آزاد می‌شوند، صرف نظر از این که چطور از بلاک خارج شده‌ایم. وقتی که در هنگام آزاد کردن منابع به exception بخوریم ممکن است یک مقدار ریزه کاری داشته باشه، اما این مشکل رو ما در آیتم ۸ بررسی کرده‌ایم و نگرانی‌ای در این مورد نداریم.

از اونجایی که auto_ptr به صورت اوتوماتیک آنچه را که به آن اشاره میکند را هنگام destroy شدن auto_ptr حذف می‌کند، این مهم است که بیشتر از یک auto_ptr به یک شیء اشاره نکند. اگر این اتفاق بیفتد در این صورت یک شیء بیشتر از یک بار حذف خواهد شد، و این برنامه شمارا در شرایطی قرار میدهد که منجر به undefined behavior خواهد کرد. برای جلوگیری از چنین مشکلاتی، auto_ptr یک خصوصیت غیر عادی را با خود دارد: کپی کردن اون‌ها(با استفاده از کپی سازنده و یا اپراتور انتساب) آن‌ها را برابر با null قرار میدهد، و اشاره‌گر جدید تنها مالک به شیء خواهد بود.

```
   std::auto_ptr<Investment>

         pInv1(createInvestment());  //Pinv1 points to the object returnd from
createInvestment
```

```
    std::auto_ptr<Investment> pInv2(pInv1); //pInv2 now points to the object; pInv1 is
now null



    pInv1 = pInv2;  //now pInv1 points to the object, and pInv2 is null
```

این رفتار عجیبی که auto_ptr در کپی کردن دارد و این که نمیشه بیشتر از یک auto_ptr به یک شیء اشاره کند نشان میدهد که auto_ptr ها بهترین گزینه برای مدیریت منابع داینامیک نیست. به طور مثال، نگه‌دارنده‌های STL نیازمند این هستند که محتوایشان یک رفتار کپی نرمال داشته باشند، بنابراین نگه‌دارنده از نوع auto_ptr امکان پذیر نیست.

یک جایگزین برای auto_ptr یک اشاره‌گر هوشمند با قابلیت شمارش refrence هست (refrence-counting smart pointer یا RCSP) یک RCSP اشاره‌گر هوشمندی است که می‌تواند حساب کتاب تعداد اشاره‌گرهایی که به یک شیء خاص اشاره دارد را داشته باشد و وقتی که کسی به این منبع اشاره نمیکند آن را حذف کند. بنابراین، RCSP رفتاری مانند garbage collection را دارد. برخلاف garbage collection ، ولی RCSP نمی‌تواند سیکل رفرنس‌ها را بشکند(یعنی دو شیء که استفاده نمی‌شوند و دو به دو به همدیگه اشاره میکنند).

بنابراین می‌توانیم کدمون رو به صورت زیر بنویسیم.

```
    std::shared_ptr<Investment>
        pInv1(createInvestment());
```

کد خیلی شبیه به همان کد قبلی است ولی این کد خیلی طبیعی‌تر رفتار می‌کند:

```
    std::shared_ptr<Investment>         //pInv1 points to the object returned
        pInv1(createInvestment());   // from createInvestment


    std::shared_ptr<Investment> pInv2(pInv1); //both pInv1 and pInv2 now point to the
object


    pInv1 = pInv2; //nothing has changed


}                       // pInv1 and pInv2 are destroyed, and the object they
```

```
                                        // point to is automatically deleted
```

هم auto_ptr و هم shared_ptr از delete در مخربشون استفاده می‌کنند، اما از [] استفاده نمی‌کنند(آیتم ۱۶ تفاوتشون رو توضیح داده). این بدین معنی است که استفاده از auto_ptr و یا shared_ptr برای آرایه‌هایی که به صورت داینامیک اختصاص داده شده اند ایده بدی است، ولی خب اگر هم چنین اتفاقی بیفتد کامپایلر آن را کامپایل خواهد کرد.

```
std::auto_ptr<std::string> aps(new std::string[10]); // bad idea! the wrong
                                                     // delete form will be used
std::shared_ptr<int> spi(new int[1024]); //same as before
```

شاید تعجب کنید که چیزی شبیه auto_ptr و یا shared_ptr برای آرایه‌های داینامیک در C++ وجود ندارد. اگر فکر می‌کنید که داشتن یک همچین چیزی براتون خوبه می‌تونید از Boost استفاده کنید. Boost::scoped_array و boost::shared_array چنین رفتاری را برای شما آماده کرده‌اند.

در این آیتم این موضوع رو بررسی کردیم که از اشیاء برای مدیریت منابع استفاده کنیم. کلاس‌های مدیریت منابع آماده‌ای برای این موضوع آماده شده است که بدانها اشاره کردیم مثل auto_ptr و shared_ptr ، ولی در برخی موارد این کلاس‌ها نمی‌توانند چیزی که شما می‌خوایند رو برآورده کنند در ااین صورت شما نیاز دارید که یک کلاس برای مدیریت منابع بنویسید. نوشتن این کلاس خیلی سخت نخواهد بود، و در آیتم ۱۵ و ۱۴ در این مورد با همدیگر بحث خواهیم کرد.