۵. Implementations

بیشتر قسمت‌ها را در مورد تعریف درست کلاس(و template class) و تعریف درست توابع(و توابع template) پرداخته‌ایم. وقتی که این موارد را به درستی توانستید بنویسید، پیاده‌سازی‌ها ساده تر می‌شوند. اما باز هم نکاتی وجود دارد که باید مراقب آن‌ها باشیم. تعریف متغیرها زودهنگام ممکن است باعث افت پرفرمنس شود. استفاده‌ی بیش‌از حد از cast ها ممکن است که باعث شود که کد خیلی کند شود، برای نگه‌داری سخت‌تر شود و به صورت بالقوه باگ برنامه افزایش پیدا کند. برگرداندن مدیریت به داخل شیء ممکن است که کپسوله‌سازی را خراب کند و باعث نارضایتی کاربران شود. و اگر نتوانیم استثناءهایی که در کد وجود دارد را درست پیش‌بینی کنیم ممکن است که منجر به نشت منابع شویم و موجب تخریب ساختار داده شود. استفاده‌ی بیش‌از حد از inlining ممکن است که باعث ایجاد code bloat شود. Coupling بیش از حد ممکن است که build time را افزایش دهد. همه‌ی این مشکلات را می‌توان حل کرد. در این فصل به بررسی این موارد خواهیم پرداخت.

Item 26: Postpone variable definitions as long as possible

هر موقع که شما یک متغیر رو توسط constructor و یا destructor تعریف می‌کنید، شما باید هزینه‌ی ایجاد و ساختن کلاس را وقتی که کنترل به تعریف متغیر می‌رسد بپردازید، و هزینه‌ی destruction را وقتی که کنترل از scope خارج می‌شود، بپردازید. یک هزینه با متغیرهای بدون استفاده وجود دارد که باید از آن دوری کنید.

شاید فکر کنید که هرگز متغیرهایی را که استفاده نشوند را تعریف نخواهی کرد، ولی بهتر است که دوباره فکر کنید. تابعی که در ادامه آورده‌ایم را ببینید، که یک ورژن رمزنگاری شده از پسورد را برمیگرداند. اگر پسورد خیلی کوتاه باشد، تابع یک logic_error را به صورت استثناء برمیگرداند، که در کتابخانه‌ی استاندارد C++ وجود دارد(آیتم ۵۴ را ببینید):

```
//this function defines the variable "encrypted" too soon
std::string encryptPassword(const std::string& password)
{
    using namespace std;
    string encrypted;

    if(password.length()<MinimumPasswordLength)
    {
        ....        //do whatever is necessary to place an
        ....        //encrypted version of password in encrypted
        return encrypted;
    }
```

```
}
```

شیء به صورت کامل بدون استفاده نیست، ولی اگر به استثناء بخوریم این متغیر بدون استفاده خواهد شد. پس شما برای ساختن و تخریب متغیر encrypted باید هزینه‌ای بپردازید، حتی اگر به یک استثناء بخورید. در نتیجه، بهتر است که تعریف encrypted را تا موقعی که به آن نیاز ندارید به تاخیر بیندازید.

```
std::string encryptPassword(const std::string& password)
{
    using namespace std;
    if(password.length()<MinimumPasswordLength)
    {
        throw logic_error("password is too short");
    }
    string encrypted;
    //do whatever is necessary to place an
    //encrypted version of password in encrypted
    return encrypted;
}
```

این کد هنوز به اندازه‌ی کافی خوب نیست، چرا که encrypted بدون هیچگونه آرگومانی است. این یعنی که سازنده‌ی پیش‌فرض کلاس استفاده خواهد شد. در بسیاری از موارد، اولین کاری که باید در مورد یک شیء انجام بدهید این است که مقداری را به آن اختصاص دهید، معمولا با یک انتساب. آیتم ۴ نشان داد که ایجاد یک شیء با استفاده از سازنده‌ی پیش‌فرض و سپس انتساب دادن یک مقدار جدید، دارای پرفرمنس کمتری از حالتی است که شیء را با مقداری که می‌خواهید، بسازید. این تحلیل در این جا نیز صادق است، به طور مثال فرض کنید که، قسمت سنگین encryptPassword در این تابع انجام شود.

```
void encrypt(std::string& s);  //encrypt s in place
```

سپس encryptPassword را می‌توان مشابه حالت زیر تعریف کرد:

```
//this function postpones encrypted's definition until
//it's necessary, but it's still needlessly inefficient
std::string encryptPassword(const std::string& password)
{
    using namespace std;
    string encrypted;  //default construct encrypted
    encrypted=password;  //assign to encrypted

    encrypt(encrypted);
    return encrypted;
```

```
}
```

یک رویکرد بهتر این است که encrypted را با password بسازیم، بنابراین قسمت سازنده‌ی کلاس که پر هزینه است را رد می‌کنیم.

```
//finally,the best way to define and initialize encrypted
std::string encryptPassword(const std::string& password)
{
   string encrypted(password);  //define and initialize via copy constructor

   encrypt(encrypted);
   return encrypted;
}
```

در واقع در این آیتم ما معنای واقعی << تا جایی که ممکن است >> را نشان دادیم. نه تنها باید تعریف متغیر را به تاخیر بیندازید، بلکه لازم است که تعریف را هم تا جایی که آرگومان مقداردهی وجود ندارد به تاخیر بیندازید. با این کار، شما از ایجاد و تخریب اشیاء غیر ضروری جلوگیری کرده‌اید، و از سازنده‌های پیش‌فرض نیز جلوگیری کرده‌اید. به علاوه، به خوانایی کد نیز کمک کرده‌اید چرا که مشخص است که متغیر کجا تعریف شده و ورودی آن چیست. در مورد حلقه‌ها چه کار باید کرد؟ اگر متغیر تنها در داخل حلقه استفاده می‌شود، بهتر است که متغیر را بیرون حلقه تعریف کنید و هر بار روی آن انتساب انجام دهید، و یا بهتر است که یک متغیر داخل حلقه ایجاد کنیم؟ کدام یک از این ساختارها درست‌تر و بهتر است؟

```
//Approach A:define outside loop
Widget w;
for (int i = 0; i < n; ++i) {
   w= some value dependent on i;
}

//Approach B:define inside loop
for (int i = 0; i < n; ++i) {
  Widget w= some value dependent on i;
}
```

در اینجا من از شیء string به یک شیء به نام Widget سوییچ کردم تا از هرگونه پیش‌داوری در مورد اجرای سازنده، مخرب و یا انتساب شیء جلوگیری کنم. هزینه‌ی اجرای این دو رویکرد به صورت زیر خواهد بود.

```
• Approach A: 1 constructor + 1 destructor + n assignments.
• Approach B: n constructors + n destructors.
```

برای کلاس‌ها هزینه‌ی انتساب کمتر از سازنده-مخرب می‌باشد، بنابراین رویکرد A خیلی کارآمدتر بوده. این وقتی که n خیلی بزرگ باشد، مشخص‌تر خواهد بود. در غیر این صورت، رویکرد B شاید بهتر باشد. به علاوه، رویکرد A نام w را در یک scope بزرگتر قابل مشاهده می‌کند، که این مورد مخالف قابلیت نگه‌داری کد می‌باشد. در نتیجه، اگر می‌دانید که اولا انتساب هزینه‌ی بیشتری از ساختن-تخریب دارد و ثانیا شما با یک قسمت از کد کار می‌کنید که حساسیت پرفرمنس بالاست، باید از رویکرد B استفاده کنید.