Table of Contents

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 **virtual**ای 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 **base class** 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;
}
```