Table of Contents

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 block scope 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 مربـوطه‌ش 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 میشـه چـون شـیء tb بـه صـورت 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 شـیء non-const ، یـک رفـرنس بـه 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 کـد هسـت، اگر چه در این مورد مجبوریم از cast استفاده کنیم. در اینجا کد این مورد را خواهیم دید ولی بـرای این که کد و چیزی که الان گفتیم براتون مشخص‌تر بشه بهتره که توضیحاتی که در ادامـه میـدیم رو هم بـا 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. مـا می‌خـواهیم کـه اپراتـور non-const فـرم const رو فراخوانی بکنه، اما اگه ما داخـل اپراتـور non-const، اپراتـور[] رو صـدا بـزنیم، در این صـورت بصورت برگشتی فقط خودمون رو صدا خـواهیم زد و تـو لـوپ میفیـتیم. بـرای جلوگـیری از یـک لـوپ بینیهایت، ما باید مشخص کنیم که می‌خواهیم اپراتور[] را از نوع const صـدا بـزنیم، امـا راه مسـتقیمی بـرای این کـار وجـود نـداره. بجـای این کـار، مـا تـایپ طبیعیـه this* را از TextBlock& بـه const TextBlock& تغییر میدهیم یا همان cast می‌کنیم. بله مـا از cast بـرای اضـافه کـردن const اسـتفاده کـردیم! در این صـورت مـا دو تـا cast خـواهیم داشـت: یکی بـرای اضـافه کـردن const بـه this*(برای این که بگیم اپراتور[] ورژن const رو صـدا بزنـه)، و دیگـری بـرای حـذف کـردن 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.

خطای Over loading * operator - must take either zero or one arguments

در برخی موارد یک برنامه‌نویس ممکن است برای overload کردن اپراتورهای * + - / به صورت زیر کار کند.

```
const Complex operator*(const Complex& rhs1,const Complex& rhs2);
```

همانطور که مشاهده می‌کنید برنامه‌نویس دو ورودی برای اپراتور در نظر گرفته است، rhs1 و rhs2 ، اما سوالی که پیش میآید این هست که پس خود کلاس در کجا قرار میگیرد، در واقع داخل اپراتور * ما می‌توانیم به اطلاعات سه شیء دسترسی پیدا کنیم، rhs1,rhs2 و خود کلاس. بنابراین در C++ برای جلوگیری از چنین مشکلی overload این اپراتور نمی‌تواند بیشتر از یک ورودی را بگیرد، چرا که با یک ورودی نیز انگار که اطلاعات هر دو کلاس نیز وجود دارد. برای نوشتن overload با استفاده از یک ورودی می‌توانیم به صورت زیر عمل کنیم.

```
class Complex
{
public:
   Complex(double real,double imag):r(real),i(imag){}
   Complex operator+(const Complex& rhs);
   double r,i;
};
Complex Complex::operator+(const Complex& rhs)
{
```

```
    Complex output;
    output.i=rhs.i+this->i;
    output.r=rhs.r+this->r;
    return output;
}
```