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 یک چنین چیزی رو انجام می‌دهید:

```
#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 لیست سمبول‌ها یا 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 type، size 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، نـه در هـدر فایل. برای این که مقدار اولیه‌ی ثابت کلاس وقتی فراهم می‌شود که مقدار ثابت declare می‌شود(یعنی numTurns مقـدار ۵ را وقـتی declare می‌شـود میگـیرد، و مقـدار اولیه‌ای در هنگـام define کـردن نمی‌تواند بگیرد.

دقت کنید که، نمی‌توان constant مربوط به یک class را با اسـتفاده از #define ایجـاد کـرد، چـرا کـه #define اهمیتی به scopeهای ما نمیدهد. زمانی که یک ماکرو تعریف شود، بـرای همه‌ی قسـمت‌های کامپایل آن ماکرو صحیح و معتبر است(مگر این که شما در جایی در بین خطوط کـد آن را #undefed کنید). که این بدین معنی است که نـه تنهـا نمی‌تـوان از #define بـرای مقـادیر constant مربـوط بـه 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 داده‌هـای 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 نتیجـه، کامپایلرهـای قـدیمی‌تر بـه شـما اجـازه 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 اسـتثناء وجـود دارد و آن هم وقـتی است که شما مقدار ثابت یک کلاس را در هنگام کامپایل تایم نیاز داشته باشـید، مثـل declartion یـک آرایه که کامپایلر نیاز به دانستن سایز آرایه داشته باشد و کامپایلرهای قدیمی چنین چیزی را پشـتیبانی نمی‌کردند. در این صورت راه حل پذیرفته شده استفاده از enum hack بـود کـه بـه صـورت زیـر 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 نشه. مثل #define، استفاده از enum ها از چنین هدر رفت حافظه اضافه جلوگیری می‌شود.

علت دوم برای آشنایی با enum hack کاملا یک چیز عملی است. خیلی از کدها از آن استفاده می‌کنند،بنابراین لازمه که وقتی چنین چیزی رو دیدیم بشناسیمش. در حقیقت، enum hack پایه‌ی تکنیک template metaprogramming هست.

بگذارید به بحث اصلی این قسمت برگردیم، یعنی دستورات پیش پردازنده، یک استفاده نادرست از #define برای پیاده‌سازیه macro ها می‌باشد که شبیه توابع هستند اما سربار فراخوانی تابع را ندارند. 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 هر تایپی را می‌پذیرفتند و دیگر این که سربار فراخوانی نداشتند، خوشبختانه ما می‌توانیم همه‌ی این مزیت‌ها را بعلاوه‌ی رفتارهای قابل پیش بینی و امنیت تایپ خروجی را با استفاده از توابع عادی داشته باشیم، همه‌ی چیزی که نیاز 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 const، enum و inline نیاز شما به دستورات پیش پردازنده به خصوص #define کاهش پیدا می‌کند، اما واقعا نمیتوان دستورات پیش پردازنده را حذف کرد. #Include کماکان ضروری است، و دستورات #ifdef و #ifndef کماکان نقش مهمی در کنترل کردن کامپایل 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 حافظه مقداری را به خود اختصاص نداده است، بلکه صرفا یک مقدار ثابت در حافظه است، و یک نام دیگر برای حافظه‌ی 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 باشید) 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 واقع nullptr به این اشاره دارد که پوینتری که ساخته شده فعلا به شیء خاصی اشاره ندارد، مگر این که آدرس آن را عوض کنیم.