برنامه نویسی استاندارد

بهمن عرب‌رضایی – تابستان ۹۵

فهرست مطالب

عنوان **صفحه**

مقدمه

در این کتاب استاندارد کد نویسی زبان C و C++ معرفی می‌شود. منظور از استاندارد کد نویسی ایجاد رویه‌ای یکسان جهت نوشتن متن کد می‌باشد به گونه‌ای که ظاهر کدهای مختلف نوشته شده در گروه برنانه‌نویسی یکسان باشد.

چرا باید استاندارد کد نویسی را رعایت نمود؟

به جرات می‌توان ادعا نمود کدی که افراد می‌نویسند همانند دستخط افراد می‌باشد و پرواضح است که کسی دستخط خرچنگ قورباغه‌ای را نمی‌پسندد لذا افراد همواره به دنبال داشتن دستخطی زیبا هستند. در این راستا روش‌های مختلفی جهت رسم‌الخط وجود دارد چه در زبان فارسی چه در زبان‌های دیگر که همه آنها سعی می‌کنند روشی زیبا را برای رسم‌الخط معرفی کنند.

در برنامه‌نویسی نیز هدف استاندارد کد نویسی چنین می‌باشد. پس همانظور که تندنویسی تا حد زیادی باعث ناخوانا شدن می‌شود در برنامه‌نویسی نیز کد نویسی سریع و بدون رعایت استانداردها باعث می‌شود کدی به دست آید که شاید برای خود برنامه‌نویس نیز بعد از مدتی قابل فهم نباشد و یادآور حکایت نویسنده نامه‌ای شود که نه دیگران توانستند نامه را بخوانند نه خود او؛ پس **همواره نامه را به گونه‌ای بنویسید که علاوه بر شما دیگران نیز بتوانند آن را بخوانند.**

شاید بتوان مزایای زیر را برای رعایت استاندارد کد نویسی ذکر نمود.

۱. هر برنامه‌نویسی می‌تواند کدها را فارغ از برنامه‌نویس آن بخواند.
۲. به سرعت می‌توان کدهای موجود را مهندسی معکوس نمود.
۳. در محیط‌های گروهی افراد کمتر با مشکلات برنامه نویسی مواجه می‌شوند.
۴. همه افراد گروه انگیزه مشترک دارند؛ دشمن مشترک **استاندار کدنویسی**!

اما رعایت استاندارد نیز معایب زیر را به همراه دراد.

۱. استاندارد معمولا خوب نیست چرا که مطابق استاندارد من(برنامه نویس) نیست.
۲. استاندارد خلاقیت را از بین می‌برد.
۳. استاندارد دست و پا گیر است و سرعت کار را کاهش می‌دهد.
۴. استاندارد مانند رییس خشک و سخت‌گیر است.

اما علیرغم موارد بالا تجربه نشان داده است که رعایت استاندارد همواره باعث نتیجه‌گیری بهتر در انتهای کار شده است. به عبارت دیگر اگر شما امروز برای اولین بار با این استاندارد برخورد کنید مانند خودرویی هستید که تازه از خط تولید خارج شده، این خودرو ممکن است بتواند با سرعت بیش از ۲۰۰کیلومتر در ساعت حرکت کند اما در ابتدای امر هیچ کاربری با این توجیه که آب‌بندی خودرو باعث اتلاف وقت در مسیر است با سرعت بالا شروع به حرکت نمی‌کند و همه هزینه زمان را پرداخت می‌کنند تا عمر خودرو را بالا ببرند.

استاندارد نیز برای افراد چنین است خصوصا در محیط‌های گروهی پس سعی نکنید خودرو صفر کیلومتر خود را با سرعت ۲۰۰کیلومتر در ساعت آزمایش کنید.

بنابراین آنچه در این کتاب آمده با هدف افزایش کارایی برنامه‌نویس و محصول است نه ایجاد روال‌های بیهوده لذا سعی کنید آن را رعایت کنید هر چند که در ابتدای کار باعث کندی بیش از حد کار شما شود.

در ادامه ابتدا درباره متغیرهای صحبت می‌شود، سپس قوانین نوشتن کد بررسی می‌شود. پس از آن در بخش ۳ استاندارد توضیحات تشریح شده و بالاخره در بخش ۵ نکات برنامه‌نویسی بیان شده است.

در انتها از خوانندگان گرامی تقاضا می‌شود نظرات خود را درباره این کتاب به ایمیل arabrezaie@gmail.com ارسال کنند. بدیهی است نظرات شما کمک بسیار زیادی در تکمیل این کتاب خواهد نمود. پیشاپیش از همکاری تمامی عزیزان تشکر می‌شود.

۱ متغیرها

متغیرها قلب هر برنامه می‌باشند. مطمئن باشید هیچ چیز به اندازه رعایت قوانین متغیرها نمی‌تواند سرعت و ماندگاری برنامه را تضمین کند. لذا این بخش را با حوصله مطالعه کنید.

اولین نکته نام متغیرها می‌باشد. مطمئن باشید که اگر در زندگی روزمره نیز نام هرشی، اتفاق و ... متناسب با آن انتخاب می‌شد خصوصا افراد، زندگی بسیار راحت بود و پیشرفت بشر سریعتر می‌شد چرا که شما با شنیدن نام هر شخص یا هر اتفاق اطلاعات اولیه مورد نیاز را به دست می‌آوردید. حال که در دنیای واقعی امکان پذیر نشده است در دنیای برنامه نویسی این امر را رعایت کنید.

۱. نام متغیر را متناسب با کاربرد آن تعریف کنید.

سعی کنید نام متغیر حتما بیانگر کاربرد آن باشد به مثال زیر توجه کنید.

```
int xx = 0;

int yy = xx + 1;
```

در این مثال متغیرهای xx و yy به هیچ وجه مفهوم نمی‌باشند. اما در مثال بعد چنین نمی‌باشد.

```
int size = 0;

int maxSize = size + 1;
```

۲. تمامی متغیرها در ابتدای تابع تعریف شوند

اگر قانون ۱ رعایت شود آنگاه این قانون مانند کارت شناسایی تابع است. شما با دیدن متغیرهای مورد استفاده در تابع به راحتی می‌توانید تا حد زیادی الگوریتم و رفتار تابع را حدس بزنید پس سعی کنید کارت شناسایی را در ابتدا نشان دهید نه در انتها.

نکته دیگر اینکه این امر باعث می‌شود خوانایی کد شما افزایش یابد و از تعریف مجدد متغیر خصوصا متغیرهای حلقه خودداری کرده و متعاقبا با خطای کامپایلر مواجه نشوید.

```
int featureCount ()

{
```

```
        int     sum = 0;
        int     i = 0;


        for (i = 0; i  < MAX_FEATURES; i++)
                sum += getFeatures (i);
}
```

۳. نام متغیر با حروف کوچک شروع شود و در متغیرهای چند کلمه‌ای حرف اول هر کلمه، غیر از کلمه اول، با حروف بزرگ باشد[۱].

این قانون ابتدا نحوه نامگذاری را بیان می‌کند ثانیا اینکه از متغیرهای چند کلمه‌ای هراس نداشته باشنید. مطمئنا کلمه <بنز CLK 2009> به مراتب مفهوم‌تر از <بنز> می‌باشد.

نکته مهم در این راستا عدم استفاده از کاکترهای ثانویه[۲] می باشند. معمولا برنامه‌نویسان از کاراکترهای _ و __ بیشتر استفاده می‌کنند. توصیه به استفاده نکردن از کاراکترهایی مانند _ و __ برای جدا کردن کلمات دو دلیل عمده دارد.

۱. نوشتن نام متغیر سریعتر انجام می‌شود چرا که کاراکترهای _ و __ نیاز به گرفتن دو کلید دارند.
۲. پیوستگی نام متغیر حفظ می‌شود و خطای چشم باعث نمی‌شود یک متغیر، چند متغیر خوانده شود.

```
int maxSize; // NOT: int max_size;
int tableCellWidth; // NOT: table_cell_width;
```

۴. برای متغیرهایی که یک هدف دارند نام‌های مختلف و مناسب انتخاب کنید.

در اغلب اوقات متغیرهای تعریف می‌شوند که همه برای یک منظور هستند مانند متغیرهای واسط یا موقت در چنین مواقعی مانند مثال زیر عمل نکنید.

---

[۱] - به این روش نامگذاری Camel Case گویند

[۲] - کارکترهای ثانویه کارکترهایی هستند که با استفاد از کلیدهای کمکی،shift, Alt و Ctrl تایپ می‌شوند.

```
int tmp1, tmp2, …;
```

بلکه سعی کنید نام مناسبی انتخاب نماید. مانند

```
int tmpSize, tmpCount, ..;
```

همواره به یاد داشته باشید که نامگذاری به سبک tmp1 , tmp2 مانند این است که شما در خانواده نام فرزندان را بگذارید بچه اول، بچه دوم و ... .

۵. از پسوند s جهت متغیرهایی که بیانگر تعداد هستند صرفنظر کنید و به جای آن از پسوند List یا پیشوند n استفاده کنید.

پسوند s در انتهای متغیر بسیار خطاخیز است. چرا که :

۱. چشم به راحتی نمی‌تواند s را در انتهای متغیر ببیند.
۲. همواره در جایی که این متغیر وجود دارد معادل آن متغیری بدون s وجود دارد و همواره باعث خطای برنامه نویس می‌شود.

لذا استفاده از این پسوند مناسب نمی‌باشد به جای آن بهتر است از پسوند List یا پیشوند n استفاده شود که توصیه به استفاده از پسوند است اما هر مورد که استفاده شد باید در تمامی برنامه قانون رعایت شود.

```
int featureList; // NOT: int features

int nLine; // NOT: int lines
```

۶. نام متغیر متناسب با حوضه دید آن باشد.

منظور از این قانون این است که مثلا در حوضه دید یک تابع می‌توان متغیری با نام tmp تعریف نمود اما تعریف این متغیر در حوضه دید یک کلاس اصلا مناسب نمی‌باشد. بنابراین سعی کنید طول نام متغیر متناسب با حوضه دید آن باشد و در زمان خواندن کد ایجاد کندی یا سردرگمی نکند.

در این زمینه قانون سرانگشتی این است : « طول نام متغیر عکس حوضه دید آن می‌باشد»

۷. طول نام متغیر متناسب با مفهوم آن باشد

همانطور که گفته شد نام متغیر باید مفهوم باشد اما باید به خاطر داشت که این به معنای ایجاد متغیرهای با طول زیاد نمی‌باشد. با توجه به نقطه‌ای که متغیر تعریف می‌شود گاهی می‌توان با استفاده از سرنام‌ها، نام متغیر را خلاصه اما مفهوم انتخاب کرد گاهی هم خیر. مثلا ممکن است در حوضه دید یک تابع کوچک بتوان از نام stuID برای شماره شناسایی دانش‌آموز استفاده نمود اما گاهی هم مجبور به استفاده از نام studentID شویم. به هرحال قانون مهم این است که با استفاده از نام‌های خیلی طولانی بیخود کد را طولانی نکنیم و با استفاده از نام‌های خیلی خلاصه نیز کد را نامفهوم نکنیم. در یک جمله « **تفریط و افرا در طول نام متغیر ننماییم**»

۸. در هنگام تعریف کردن اشاره‌گر * را به متغیر بچسانید نه به نوع آن.

در برخی از استانداردها توصیه می‌شود * را به نوع متغیر بچسبانید اما این پیشنهاد معایب زیر را دارد:

۱. در هنگام خواندن خطای چشم می‌تواند آن را نبیند و تصور شود که متغیر اشاره‌گر نیست.
۲. در اکثر توصیه‌های برنامه‌نویسی، توصیه می‌شود برای اشارگرها نوع جدید تعریف کنید تا مورد قبل پیش نیاید. در واقع نکته در اینجاست که چسباندن * به نوع خود بیانگر نوع جدید است نه نام متغیری از نوع اشاره‌گر.

بنابراین مانند مثال زیر عمل کنید.

```
RGFeature  *feature = NULL, centerFeature; // NOT: RGFeature*   feature = NULL, centerFeature;
```

علاوه بر موارد بالا در تمامی کتب آموزشی تاکید می‌شود که در تعریفی مانند مثال بالا feature اشاره‌گر است اما centerFeature نیست حال آنکه این موضوع در استاندارد توصیه شده به خوبی دیده نمی‌شود اما در استاندارد این مستند به خوبی دیده شده است.

۹. نام متغیرهای کلاسی با پیشوند m شروع شود.

این نکته جهت بالابردن خوانایی کد است و باعث می‌شود در متن کد به راحتی بتوان متغیرهایی کلاسی را تشخیص داد. پرواضح است که توصیه ۳ نیز باید در اینجا رعایت شود و نام متغیر چون بعد از پیشوند می‌آید حتما باید با حرف بزرگ شروع شود.

```
class RGFeature
{
```

```
privet:
        int mSize; // NOT: int size_; int _size; int m_size; int msize;
}
```

۱۰. شمارنده‌های حلقه را i, j, k بنامید و اگر بیش از سه سطح تودرتویی داشتید از l, m, n برای سطوح داخلی استفاده کنید.

این توصیه تقریبا رایج است چرا که شمارنده در کوچکترین حوضه دید قرار دارد پس با قانون ۶ و ۷ همخوانی دارد و همچنین رایج بودن آن به نوعی قانون شده و خلاف قانون عمل کردن جایز نمی‌باشد.

۱۱. نام کلاس‌ها، انواع type و ساختارها structure حتما با حروف بزرگ و با پیشوند مناسب شروع شود.

پیشوند مناسب، سرنام[۳] شرکت می‌باشد که معرف مالکیت کد است. استفاده از حروف بزرگ جهت تمایز سریع متغیر از نوع آن می‌باشد.

```
class RGFeature
{
        …
};
// NOT:
class rgFeature …
//NOT:
class RgFeature …
```

[۳] - Acronym

۱۲. نام ماکروها تمامی با حروف بزرگ باشد و با خط تیره زیر _ کلمات از یکدیگر جدا شوند.

استفاده از حروف بزرگ برای تمیز سریع ماکرو از سایر متغیرها و انواع می‌باشد. و خط تیره زیر نیز همینطور چرا که در جای دیگری از آن استفاده نمی‌شود.

```
#define MAX_SIZE  5


//NOT:
#define MAXSIZE 5
#define maxSize    5
#define max_size   5
```

۱۳. نام توابع با حروف کوچک شروع شده ،مانند قانون ۳، و بیان کننده کارکرد تابع باشد.

توجه داشته باشید در زبان C توابع در عمل متغیرهای از جنس اشاره‌گر هستند بنابراین با این توجیه می‌توان توصیه ۳ را نیز در اینجا استفاده نمود علاوه بر آن از دیدگاه مفهومی در برنامه نویسی تابع و متغیر از یک مفهوم می‌باشند.

نکته مهم در نام توابع مفهوم بودن نام تابع است. بسیار ضروری است که از روی نام تابع بتوان وظیفه آن را تشخیص داد این موضوع به شدت به مهندسی معکوس کد کمک می‌کند. مثلا تابعی که هدف آن استخراج معدل یک آرایه است نام average برای آن نامناسب است چرا که مفهوم محاسبه در آن دیده نمی‌شود بلکه به نظر می‌آید مقداری را با این مضمون خوانده و برمی‌گرداند در اینجا استفاده از نام calcAverage یا computAverage مناسب‌تر است.

بر همین اساس توصیه‌های زیر در نامگذاری توابع مفید به نظر میآید:

۱. توابعی که برای خواندن یا نوشتن متغیرهای کلاسی هستند با شکل set[var] () و get[var] () نامگذاری شوند.
۲. توابعی که برای جستجو به کار می‌روند با پیشوند find همراه شوند مانند findMax, findNearestFeature.
۳. توابعی که مقداردهی اولیه انجام می‌دهند با پیشوند init یا initialize شروع شوند مانند initDistance.

۴. از پیشوند is برای توابعی که مقدار بولین برمی‌گردانند استفاده شود مانند isActive, isEnabled
۵. در صورت نیاز پیشوندهای has , can و should نیز جایگزین مناسبی برای پیشوند is می‌باشند.

۲ نحوه کدنویسی

کد نوشته شده ظاهر برنامه است نه باطن برنامه پس پرواضح است که باید اهمیت زیادی به باطن برنامه داد نه ظاهر آن. اما هر باطن زیبا حتما همراه ظاهری زیبا است و نمی‌توان موردی را یافت که باطنی زیبا اما ظاهری زشت داشته باشد. ظاهر زیبا همواره باعث جذب افراد می‌شود در نتیجه اگر می‌خواهید سایر برنامه نویس‌ها، که احتمالا از کد شما کاری جز ایراد گرفتن ندارند، به کار شما علاقمند شوند مطمئن باشید رعایت ظاهر زیبا بسیار به شما کمک خواهد کرد.

۱۴. کد خود را شلوغ نکنید.

از نکات مهم در کد نویسی واضح بودن کد است لذا سعی کنید کدها غیر ضروری که معمولا به توضیحات (comment) تبدیل شده‌اند، را حذف کنید زیرا این کدها معمولا فقط باعث طویل شدن کد شما می‌شوند و اشکال‌زدایی، خواندن کد و ردگیری را بی‌جهت سخت می‌کنند. پس همواره سعی کنید اگر از صحت کارکردکدی مطمئن شدید سایر کدهای توضیحی را حذف کنید و اگر چنین نیست ابتدا مشکلات آن قسمت را برطرف کنید سپس ادامه دهید.

توجه داشته باشید که برای رفع خطا معمولا افراد از ایجاد خروجی استفاده می‌کنند مثلا در زبان سی دستور printf به فراوانی استفاده می‌شود. سعی کنید

۱. خروجی‌ها کنترل شده باشند و خروجی بی جهت تولید نکنید.
۲. خروجی به گونه‌ای باشند که حذف آن‌ها راحت باشد
۳. هرگاه مشکلات برطرف شد خروجی‌های را حذف کنید مگر موارد ضروری که باید در برنامه بمانند.

بنابراین خود را در یک اتاق شلوغ و پر از وسایل قرار ندهید.

```
        int scnNum = 0;
        int i;
//      QStringList clientNames;
        QStringList clientIP;
/*      RGSceen *scn;
        QHBoxLayout *layout = NULL;
```

```
        setupUi(this);


        mConfig = config;
*/
        mConfig->get ("screen", "name", clientNames);
        mConfig->get ("screen", "ip", clientIP);
        scnNum = clientNames.size ();
//      if (scnNum == 0 || scnNum != clientIP.size ())
                QMessageBox::critical (this, tr ("Client Ui"), tr ("Invalid screen definition!"));
//      else
        {
                layout = new QHBoxLayout (this);
                for (i = 0; i < scnNum; i++)
                {
/*                      scn = new RGSceen (this, clientIP.at (i));
                        scn->pushButton->setText (clientNames.at (i)); */
                        layout->addWidget (scn);
                }
```

مقایسه کنید با مورد زیر

```
        int scnNum = 0;
        int i;
        QStringList clientNames;
        QStringList clientIP;
        RGSceen *scn;
        QHBoxLayout *layout = NULL;
```

```
setupUi(this);


mConfig = config;
mConfig->get ("screen", "name", clientNames);
mConfig->get ("screen", "ip", clientIP);
scnNum = clientNames.size ();
if (scnNum == 0 || scnNum != clientIP.size ())
        QMessageBox::critical (this, tr ("Client Ui"), tr ("Invalid screen definition!"));
else
{
        layout = new QHBoxLayout (this);
        for (i = 0; i < scnNum; i++)
        {
                scn = new RGSceen (this, clientIP.at (i));
                scn->pushButton->setText (clientNames.at (i));
                layout->addWidget (scn);
        }
}
```

پرواضح است که مطالعه کد دوم به مراتب راحت‌تر می‌باشد.

۱۵. از شکستن بی‌مورد خطوط پرهیز کنید.

به یاد داشته باشید زمانی که متنی را مطالعه می‌کند رفتن به خط بعدی هم زمان‌بر است و هم می‌تواند باعث پریدن از روی خط به دلیل خطای چشم شود پس سعی کنید خطوط را بی‌مورد نشکنید تا قانون ۱۴ را نیز نقض نکرده باشد.

```
int  findMax (QStringList stringList, int startIndex);
// NOT:
int
findMax (
        QStringList stringList,
        int startIndex)
```

اما چنانچه طول خط زیاد طولانی شد بطوریکه نیاز به مرور افقی صفحه پیدا کرد آنگاه مطابق قوانین زیر سعی کنید خطوط را بشکنید.

۱. پارامترهای تابع هر کدام در یک خط و کاما در انتهای خط باشد.
۲. عملگرها همواره در انتهای خط باشند.

```
int findMax (
        QStringList stringList,
        int         startIndex)
// NOT:
int findMax (
        QStringList stringList, int startIndex)
```

```
x =  ( a + b +
     c + d );
// NOT:
x = ( a + b
    + c + d );
```

۱۶. از به کاربردن اعداد جادویی پرهیز کنید.

معمولا برنامه‌نویسان همواره با متغیرهای مواجه می‌شوند که مثلا بیش از دو سه مقدار، حالت، مختلف نمی-گیرند در چنین موادری ممکن است در ذهن خود برای هر مقدار مفهومی متصور شوند و در عبارات شرطی، ورودی توابع و ... به جای مفهوم عدد را ارسال کنند مانند مثال زیر:

```
if ( type == 1)
        getSize (type, 2);
```

همانطور که دیده می‌شود.

۱. زیبایی کد از بین رفته است.
۲. کد بسیار نامفهوم است.
۳. ردگیری کد جهت اشکال زدایی یا مهندسی معکوس سخت است.

بنابراین باید به جای این اعداد از ماکروها استفاده نمود.

```
#define SURFACE         1
#define AIR             2
#define SUBSURFACE  3
..
..
if (type == AIR)
        getSize (type, SURFACE);
```

البته گاهی مواقع در کتب توصیه می‌شود در چنین مواردی نوع شمارشی تعریف شود. این مورد توصیه نمی‌شود چرا که:

۱. زمان کامپایل بالا می رود.
۲. کامپایلرها برای انواع شمارشی در مورد عملگرها و Cast کردن خطاهای زیادی تولید میکنند.

۳. در زمان مهندسی معکوس ردگیری تعداد زیاد انواع متغیر کار را سخت می‌کند.

۱۷. فضای خالی (space) ابزار جادویی زیبا سازی است.

مطمئن باشید هیچ ابزار آرایشی در زمینه کاربردی خود به اندازه فضای خالی در برنامه‌سازی قدرت زیباسازی ندارد. پس بدانید فضای خالی دوست شماست و از آن دوری نکنید. همواره سعی کنید در موارد زیر فضای خالی ایجاد شود.

۱. بعد و قبل از عملگرها فضای خالی ایجاد شود.
۲. بعد از کاما در پارامترهای ورودی توابع فضای خالی ایجاد شود.
۳. بعد از ; در حلقه‌ها.
۴. قبل از) و بعد از (در دستورات فضای خالی ایجاد شود.
۵. نام توابع را به پرانتز بعد از آن نچسبانید و یک فضای خالی ایجاد کنید.
۶. کلمات کلیدی را به پرانتزها نچسبانید.
۷. همترازی بسیار تاکید شده پس از فضای خالی برای همتراز کردن استفاده کنید.

به کد نمونه زیر توجه کنید.

```
int  findMaxArea (int  sizeList[], int count)
{
        int      i;
        float   area = 0.0, maxArea = 0.0;

        for ( i = 0; i < (count - 1); i++)
        {
                area = sizeList [i] * sizeList [ i + 1 ];
                if ( maxArea < area)
                {
                        maxArae = area;
                }
```

```
        }


        return maxArae;
}
```

همانطور که دیده می‌شود در زمان تعریف متغیر با استفاده از فضای خالی سعی شده است همترازی رعایت شود.

قانون کلی این است که نه افراط شود نه تفریط یعنی از فضای خالی به اندازه‌ای استفاده شود که کد زیبا شود و خوانایی و سهولت ردگیری آن نیز حفظ شود پس خود را صرفا مقید به این قانون ندانید اما آن را نیز کاملا نقض نکنید.

۱۸. توابعی با چند هزار خط و یا حتی چندصد خط یعنی شکست.

به یاد داشته باشید که توابع قرار نیست خود یک برنامه کامل باشند بلکه هدف آن‌ها جلوگیری از کد نویسی مجدد و ایجاد مجردسازی[۴] است پس اگر تابعی چند هزار خط شد یعنی این تابع خود یک برنامه کامل است که می‌تواند مجددا به چندین تابع دیگر شکسته شود لذا هر گاه چنین تابعی را در برنامه خود دیدید بدانید که در طراحی یا نحوه پیاده‌سازی ایراد اساسی وجود دارد و باید بازبینی شود.

البته توجه کنید که در اینجا تعداد خطوط مفید تابع مد نظر است چرا که برخی خطوط مانند فضاهای خالی که جهت خوانایی ایجاد شده‌اند به حساب نمی‌ایند. بهترین معیار جهت بررسی اینکه آیا این تابع بلندتر از حد نیاز است این است که علاوه بر طول تابع در تابع بیش از یک یا حداکثر دو الگوریتم پیاده‌سازی نشده باشد.

۱۹. از دستورات تودرتو پرهیز کنید.

برخی مواقع برنامه‌نویسان جهت کوتاه کردن طول برنامه از دستورات تودرتو استفاده می‌کنند. مانند مثال زیر:

```
a = ((x =  sin ( getAngle (feature) / 2.0)) > VALID_SIZE ?  2*x:x);
```

[۴] - Abstraction

فهم این نوع دستورات بسیار سخت و حتی برای خود برنامه‌نویس نیز بسیار پیچیده می‌باشد و می‌توانند از دستورات خطاخیز برنامه باشد لذا از نوشتن این نوع دستورات شدیدا پرهیز کنید و سعی کنید دستور را به چند دستور بشکنید. مانند نمونه زیر:

```
angle = getAngle (feature) / 2.0;
y = sin (angle);
if (y > VALID_SIZE)
        a = 2 * x;
else
        a = x;

// OR:

a = x;
angle = getAngle (feature) / 2.0;
y = sin (angle);
if (y > VALID_SIZE)
        a = 2 * x;
```

پر واضح است که نمونه کد اخیر به راحتی و به سرعت قابل فهم و اشکال زدایی می‌باشد.

۲۰. از goto **استفاده نکنید.**

استفاده از goto باعث شکسته شدن ساختار پیمانه‌ای (Modular) برنامه می‌شود لذا استفاده از آن بسیار نامناسب است اما در یک مورد استفاده از این دستور نه تنها مناسب بلکه پسندیده می‌باشد.

در برخی مواقع در توابع در نقاط مختلف برنامه منابعی ایجاد می‌شود که قبل از خروج از تابع باید آزاد شوند اما در تابع چندین نقطه خروجی وجود دارد در چنین مواقعی برای جلوگیری از نوشتن تکراری دستورات و همچنین شرط‌های زیاد در انتهای تابع برچسب خروج زده می‌شود و تابع از آن نقطه خارج شده و تمامی منابع را در یک نقطه آزاد می‌کند مانند مورد زیر:

```
int sampleFunction ()
{
        …
        x = new foo ();
        ..
        if ( … )
                goto END;
        y = new foo ();
        z = new foo ();
        …
        for ( … )
        {
                fp = fopen ( … );
                …
                if ( … )
                        goto END;
        }

END:
        if (x) delete x;
        if (y) delete y;
```

```
        if (z) delet z;
        if (fp) fclose (fp);
        …
}
```

همانطور که دیده می‌شود چنانچه از دستور goto استفاده نمی‌شد برنامه نویس مجبور بود در نقاط خروج هر بار کد تکراری نوشته تا بتواند منابع را به درستی آزاد کند.

همچنین در حلقه‌های تو در تو برای خروج از حلقه نیز استفاده از goto بلامانع است اما همواره باید توجه داشت که ساختار صحیح کمتر نیازمند به استفاده از goto جهت خروج از حلقه می‌شود لذا در هنگام استفاده از این روش حتما مطمئن باشید راه‌حل بهتری وجود ندارد

۲۱. #include‌ها را دسته‌بندی کرده و با اولویت قرار دهید.

همواره از آوردن #include‌ها بصورت نامنظم و درهم خودداری کنید. زیر این امر باعث می‌شود برخی خطاها در جاهایی خود را نمایان سازند که شما انتظار آن را ندارید به نمونه زیر توجه کنید.

```
/// sample.h
#ifndef RG_SAMPLE_H
#define RG_SAMPLE_H

class RGSample
{
        …
}
#endif

///sample.cpp
#include "sample.h"
```

```
#include "stdio.h"

...
```

کامپایل این کد باعث می‌شود کامپایلر به شما خطایی بدهد که منشا آن را در فایل stdio.h اعلام کند! پر واضح است که در این فایل از دید کامپایلر خطایی نباید وجود داشته باشد اما دلیل چیست؟ در واقع کامپایلر با ندیدن علامت ; در انتهای تعریف کلاس تا ; بعدی را خوانده و آنگاه خطا را در آن نقطه اعلام می‌کند و این ; در فایل stdio.h دیده خواهد شد و حال آنکه خطا در فایل sample.h بوده است. این موضوع می‌تواند به سردرگمی برنامه‌نویس منجر شود.

بنابراین توصیه می‌شود هدر فایل‌ها را دسته بندی کنید و بین هر دسته یک فاصله قرار دهید و آن‌ها را بر اساس اطمینان به صحت کارکرد بالاتر از باقی قرار دهید. به این ترتیب مطمئن هدر فایل‌های سیستمی قبل از همه و هدر فایل‌های خود برنامه در انتها باید قرار بگیرد.

در نمونه زیر الگویی عمومی برای این موضوع بیان شده است.

```
#include system header file

#include external library header file

#include library header file

#include program header file
```

و این هم یک نمونه:

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <QWidget>
#include <QObject>


#include <geos.h>
#include <ogr.h>


#include "sample.h"
```

۲۲.از #include **در هدر فایل استفاده نکنید.**

استفاده از #include در هدر فایل معایب زیر را دارد:

۱. چون هدر فایل‌ها خود در فایل‌های سورس وارد می‌شوند خود باعث نقض قانون ۲۱ می‌شوند.
۲. تقدم و تاخر برخی هدر فایل‌ها باعث رخداد خطا در کامپایلر می‌شود.
۳. ممکن است هدر فایل‌های غیر ضروری در سورس وارد شده و در نتیجه ایجاد وابستگی غیر ضروری کرده و زمان کامپایل را بیهوده افزایش دهد.

هر چند که این قانون باعث خواهد شد که لیست بلندی از هدر فایل در هر فایل سورس دیده شود اما دارای مزایای زیر است:

۱. وابستگی‌های هر سورس فایل در همان‌جا دیده می‌شود.
۲. مهندسی معکوس سورس ساده‌تر می‌شود.

البته در برخی موارد این قانون باید نقض شود مانند:

۱. در کتابخانه‌ها هدر فایلی تعریف می‌شود که تمامی هدر فایل‌های دیگر در آن قرار می‌گیرند و در هنگام استفاده از کتابخانه تنها آن هدر فایل نیاز است.
۲. برخی وقت‌ها ممکن است یک کلاس خود به چندین فایل تبدیل شود اما جهت مدیریت راحت‌تر برای آن یک فایل هدر و سورس در نظر می‌گیرند که تمامی هدر فایل‌ها و سورس فایل‌ها در این دو فایل وارد می‌شوند و کامپایلر فقط این دو را کامپایل می‌کند.

۲۳. در هدر فایل هیچگاه پیاده‌سازی انجام ندهید.

همواره به یاد داشته باشید که **هرسخن جایی و هر نکته مکانی دارد** پس بی‌موقع و لامکان صحبت کردن باعث دردسر است؛ یعنی اگر شما در هدر فایل پیاده‌سازی انجام دهید و پیاده‌سازی شما خطا داشته باشد آنگاه ممکن است کامپایلر کوهی از خطا به شما تقدیم کند. چرا؟ چون هدر فایل ممکن است در چندین سورس فایل وارد شده باشد و در هر کدام خود منشا حداقل یک خطا خواهد شد.

علاوه بر آن ممکن است نیاز شود هدر فایل منتشر[۵] شود آنگاه سورس شما هم نیز منتشر می‌شود.

۲۴.از پیش تعریف (forward definition) پرهیز کنید.

چنانچه قوانین ۲۰ و ۲۱ را به درستی رعایت کنید نیازی به پیش تعریف نخواهید داشت. استفاده از پیش تعریف تنها در زمانی که در استفاده از ساختارها یا کلاس‌ها ایجاد حلقه شده است، مجاز می‌باشد و در سایر موارد نباید این کار را کرد. استفاده از پیش تعریف مشکلات زیر را دارد:

۱. اگر نام ساختار یا کلاس تغییر کند تقریبا هیچ محیطی وجود ندارد که بتواند بدون استفاده از ابزار search replace بصورت هوشمندانه تمامی پیش‌تعریف‌ها را تصحیح کند.
۲. ممکن است کلاسی حذف شود اما پیش‌تعریف آن همچنان در کد بماند که خود می‌تواند باعث خطای کامپایلر یا لینکر شود.

---

[۵] - Release

۳ توضیحات

تقریبا هیچ کتاب برنامه‌نویسی وجود ندارد که به شدت در آن توصیه به نوشتن توضیحات در برنامه نکرده باشد اما معمولا نکته مهمی که بیان نمی‌شود این است که نوشتن توضیحات نیز باید بر اساس قاعده و اصول باشد. نوشتن توضیحات بدون قاعده مانند خط‌خطی کردن یک متن است که هیچ کمکی به خوانایی نمی‌کند.

در نوشتن توضیحات باید از قاعده‌ای پیروی شود که علاوه بر ایجاد توضیحات در برنامه امکان تولید مستندات نیز از آن وجود داشته باشد. بر این اساس استانداردهای زیادی برای نوشتن توضیحات ایجاد شده است که منجر به تولید مستندات برنامه نیز می‌شود مانند doxygen.

۲۵.کد خود مستند[۶] وجود ندارد لطفا توضیحات بنویسید.

امروزه رایج شده است که افراد مدعی هستند زبان‌های برنامه نویسی خود مستند هستند و نیازی نیست در کد توضیحات بگذارند اما به یاد داشته باشید:

۱. زبان برنامه نویسی متفاوت از زبان گفتاری شما می‌باشد.
۲. زبان خود مستند یعنی زبان گفتاری و به نظر نمی‌آید کامپایلری وجود داشته باشد که زبان گفتاری را بفهمد (سراغ دارید معرفی کنید!)
۳. اگر زبانی خود مستند است چرا تولید کننده آن برای آن مستندات نوشته است.

بنابراین مطمئن باشید زبان خود مستند وجود ندارد و باید حتما در کد خود توضیحات بنویسید.

۲۶. توضیح واضحات ندهید.

مهمترین هدف توضیحات روشن کردن و ساده نمودن فهم کدهای پیچیده است و کسی به دنبال درک کدهای ساده نمی‌باشد.

```
max = ( y > x ? y:x);  // find max value
```

حتی فردی هم که یک خط کد ننوشته باشد بدون توضیح هم می‌تواند بفهمد دستور بالا چه می‌کند پس **آب در هاون نکوبید**.

[۶] - self document

بزرگان فرموده‌اند: « من شیطان است». پس در زمان نوشتن توضیحات نگویید: « من در اینجا مقدار ... را محاسبه نمودم». و همیشه تصور کنید دارید مستندات کد را می‌نویسید.

بنابراین سعی کنید در زمان نوشتن توضیحات الگوریتم را بصورت واضح شرح دهید و کارکرد هر خط را جلوی آن ننویسید.

۲۷.از استاندارد پیروی کنید.

>----------<

۲۸.سعی کنید از کلمات کلیدی استفاده کنید.

در هنگامی که توضیحاتی جهت اطلاعا سایر برنامه نویس‌ها می‌نویسید و مایلید به آن توجه کنند سعی کنید از کلمات کلیدی زیر استفاد کنید.

۱. :TODO : جهت بیان مواردی که ناقص است و یا جهت بهبود باید انجام شوند
۲. BUG:[bug id] : جهت توضیح خطا یا اشکالی که در کد وجود دارد اما راه حلی برای آن نیست یا هنوز پیاده سازی نشده است. برای هر خطا یک شناسه ایجا دکنید. بهترین شناسه نام خودتان به همراه یک شماره ترتیبی است مثلا bahman02
۳. :KLUDGE : هرگاه بخواهید به سایرین اعلام کنید کدی که نوشته شده نازیبا یا ناکارامد است و باید مجددا بررسی شود.
۴. :TRICKY : هرگاه در قسمتی از کد از ترفند خاصی استفاده کرده‌اید که سایرین با آن آشنا نیستند یا کدی است که به تغییرات سایر نقاط برنامه حساس است با استفاده از این کلمه کلیدی سایرین را از آن خبر دار کنید.
۵. :WARNING : هرگاه در کد خود نقصی وجود دارد که درحال حاضر مشکلی ایجاد نمی‌کند اما پتانسیل ایجاد اشکال دارد از این کلمه استفاده کنید.
۶. :COMPILER : اگر در قسمتی از کد خطایی وجود دارد که ناشی از کامپایلر است و شما نتوانستید آن را برطرف کنید از این کلمه جهت توضیحات استفاده کنید.

رعایت استفاده از کلمات کلیدی بالا سبب خواهد شد سایرین با دیدن آن، توضیحات شما را مطالعه کنند و از ایجاد خطا جلوگیریی نمایند.

۴ توصیه‌های برنامه نویسی

در این بخش توصیه‌های بررسی می‌شوند که در نگاه اول ممکن است خیلی به استاندارد کد نویسی و یا دست‌کم ظاهر کد ارتباطی نداشته باشد اما رعایت آن‌ها به شدت باعث خوانایی کد، پایداری کد و سهولت فهم کد می‌گردد.

۲۹.از به کار بردن فضای نام[۷] و الگو[۸] خودداری کنید.

فضای نام ابزار مناسبی برای دسته بندی است اما استفاده از آن مشکلات زیر را می‌تواند به همراه داشته باشد:

۱. دستورات را بیهوده طولانی می‌کند. مانند mapserver::map::features::VSField
۲. استفاده از این ابزار برای دسته‌بندی ممکن است برنامه نویس را دچار خطا کرده و کلاسی با نام مشابه در دو فضای نام مختلف ایجاد نماید که برنامه را خطا خیز نماید

بنابراین استفاده از فضای نام غیر ضروری می‌نماید.

الگوها نیز ابزارهای مناسبی برای تولید کلاس‌ها عام منظوره یا چند منظوره نمی‌باشد. چرا که این کلاس‌ها هیچ بهینه‌سازی در کد ایجاد نشده و در زمان کامپایل حجم کد افزایش می‌یابد و همینطور در زمان اشکال زدایی اطلاعات نوع نامشخص است. همچنین با استفاده از وراثت ابزار بسیار قویتری وجود دارد که می‌توان جایگزین آن نمود.

۳۰. سعی کنید در برنامه استثنا[۹] تولید نکنید.

به یاد داشته باشید زمانی که استثنا تولید می‌کنید باید در جایی آن را مدیریت کنید و در آن نقطه مجبور خواهید شد از بلوک try {} catch () استفاده کنید. این بلوک کد شما را بسیار ناکارامد می‌کند چرا که کامپایلر مجبور است به ازای هر خط، خطا تولید و متغیرها را جهت تولید استثنا بررسی کند که می‌توان تصور نمود تا چه حد کارایی کد از بین می‌رود.

علاوه بر این اکثر برنامه نویس‌ها چندان خود را درگیر مدیریت خطا نمی‌کنند لذا استثناها باعث ناپایداری کد خواهد شد.

---

۷ - Namespace
۸ - Template
۹ - Exception

۳۱. از تعریف کلاس‌های منفرد[۱۰] و متغیرهای ایستا[۱۱] جددا خوداری کنید.

یکی از اهداف برنامه نویسی شی گرا پنهان سازی اطلاعات و کپسوله کردن آن است اما کلاس‌های ÷ منفرد و متغیرهای ایستا این موضوع را نقض می‌کنند. استفاده از این کلاس‌ها یا متغیرها باید با توجیه مناسبی همراه باشد.

علاوه بر آن در برنامه‌های مبتنی بر نخ (thread base programming) این نوع کلاس‌ها و متغیرها به شدت خطاخیز بوده و موجبات ناپایداری برنامه می‌باشند.

۳۲.در سازنده‌ها پیاده‌سازی انجام ندهید

همانطور که می‌دانید سازنده‌ها نمی‌توانند خطا برگرداند و هرگاه فراخوانی می‌شوند فضا را برای متغیر اشغال می‌کنند پس اگر خطایی در آن‌ها رخ دهد امکان ردگیری آن وجود ندارد پس سعی کنید در سازندها تنها مقداردهی اولیه کنید و دستوراتی را اجرا کنید که خطا ایجاد نمی‌کنند.

۳۳. توابع حتما شماره خطا برگردانند.

بسیار مهم است که هرگاه تابعی فراخوانی می‌شود و در آن الگوریتمی اجرا می‌شود به نحوی فراخواننده بتواند از نتیجه کار مطلع شود لذا توصیه می‌شود مجموعه‌ای از خطاها برای هر تابع تعریف شود و تابع در صورت اجرا موفقیت آمیز مقدار صفر برگرداند و در غیر اینصورت شماره خطا را برگرداند. البته در برخی مواقع توابع با تعیین محدوده نیز شماره خطا برمی‌گردانند مثلا اگر مقدار بیش از صفر باشد یعنی جواب صحیح و اگر کمتر از صفر باشد یعنی خطا.

به هر حال بسیار مهم است که فراخواننده به نحوی از روند اجرای تابع مطلع شود پس **جعبه سیاه نسازید**.

۳۴.از عبارات ریاضی استفاده کنید

در اغلب مواقع برنامه‌نویس‌ها به جای تجزیه و تحلیل مساله و ارائه راه حل مناسب از دستورات متعدد برنامه-نویسی مانند if, switch, for, … استفاده می‌کنند در حالیکه با اندکی تجزیه و تحلیل به راحتی می‌توان با

---

[۱۰] - Singleton

[۱۱] - Static

یک رابطه ریاضی دستورات متعدد را حذف نمود. مثلا فرض کنید زاویه (teta)θ مشخص است و باید تعیین نمود در کدام ناحیه قرار دارد.

```
if (teta > 0 && teta < 90)
    area = 1;
if (teta > 90 && teta < 180)
    area = 2;
if (teta > 180 && teta < 270)
    area = 3;
if (teta > 270 && teta < 360)
    area = 4;
```

به راحتی می‌توان دید به جای ۸ خط بالا می‌توان خط زیر را جایگزین نمود.

```
area = (int)(teta / 90.0) + 1;
```

بنابراین با تجزیه و تحلیل مساله می‌توان کد را کوتاهتر و خواناتر نمود. راه اول ساده‌ترین راه و اولین راهی است که به ذهن برنامه‌نویس می‌رسد لذا **همواره اولین راه‌حل بهترین راه‌حل نیست.**

۵ لیست توصیه‌ها

۱. **نام متغیر را متناسب با کاربرد آن تعریف کنید.**

۲. **تمامی متغیرها در ابتدای تابع تعریف شوند**

۳. **نام متغیر با حروف کوچک شروع شود و در متغیرهای چند کلمه‌ای حرف اول هر کلمه، غیر از کلمه اول، با حروف بزرگ باشد.**

۴. **برای متغیرهایی که یک هدف دارند نام‌های مختلف و مناسب انتخاب کنید.**

۵. **از پسوند** s **جهت متغیرهای که بیانگر تعداد هستند صرفنظر کنید و به جای آن از پسوند** List **یا پیشوند** n **استفاده کنید.**

۶. **طول نام متغیر متناسب با حوضه دید آن باشد.**

۷. **در هنگام تعریف کردن اشاره‌گر** * **را به متغیر بچسانید نه به نوع آن.**

۸. **نام متغیرهای کلاسی با پیشوند** m **شروع شود.**

۹. **شمارنده‌های حلقه را** i, j, k **بنامید و اگر بیش از سه سطح تودرتویی داشتید از** l, m, n **برای سطوح داخلی استفاده کنید.**

۱۰. **نام کلاس‌ها، انواع** type **و ساختارها** structure **حتما با حروف بزرگ و با پیشوند** RG **شروع شود.**

۱۱. **نام ماکروها تمامی با حروف بزرگ و با خط تیره زیر** _ **کلمات از یکدیگر جدا شوند.**

۱۲. **نام توابع با حروف کوچک شروع شده ،مانند قانون ۳، و بیان کننده کارکرد تابع باشد.**

۱۳. **کد خود را شلوغ نکنید.**

۱۴. **از شکستن بی‌مورد خطوط پرهیز کنید.**

۱۵. **از به کاربردن اعداد جادویی پرهیز کنید.**

۱۶. فضای خالی (space) ابزار جادویی زیبا سازی است.

۱۷. توابعی با چند هزار خط و یا حتی چندصد خط یعنی شکست.

۱۸. از دستورات تودرتو پرهیز کنید.

۱۹. از goto استفاده نکنید.

۲۰. #includeها را دسته‌بندی کرده و با اولویت قرار دهید.

۲۱. از #include در هدر فایل استفاده نکنید.

۲۲. در هدر فایل هیچگاه پیاده‌سازی انجام ندهید.

۲۳. از پیش تعریف (forward definition) پرهیز کنید.

۲۴. کد خود مستند (self document) وجود ندارد لطفا توضیحات بنویسید.

۲۵. توضیح واضحات ندهید.

۲۶. از استاندارد پیروی کنید.

۲۷. سعی کنید از کلمات کلیدی استفاده کنید.

۲۸. از به کار بردن فضای نام (namespace) و الگو (template) خودداری کنید.

۲۹. سعی کنید در برنامه استثنا (exception) تولید نکنید.

۳۰. از تعریف کلاس‌های منفرد (singleton) و متغیرهای ایستا (static) جددا خوداری کنید.

۳۱. در سازنده‌ها پیاده‌سازی انجام ندهید

۳۲. توابع حتما شماره خطا برگردانند.