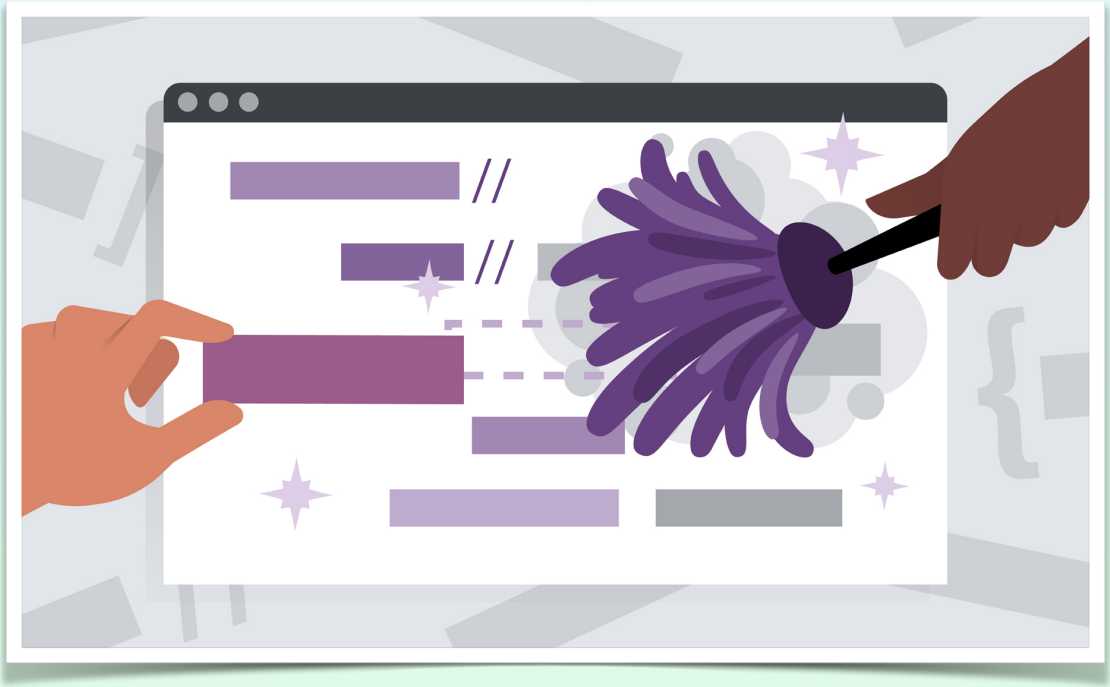
به نام خداوند بخشنده‌ی مهربان

شماره‌ی ۴ ۶ آبان ۱۴۰۰

کد نامه

ویژه‌ی دانش‌جویان مبانی برنامه‌سازی نیم‌سال اول ۱۴۰۱-۱۴۰۰ دانشکده‌ی مهندسی کامپیوتر دانشگاه صنعتی شریف



تمیز بنویسیم!

با یکی از مهم‌ترین عادت‌ها در برنامه‌نویسی آشنا شوید

آن چیزی که یک برنامه نویس معمولی را از یک برنامه نویس خبره جدا می‌کند، توجه به جزئیاتی هم‌چون کدنویسی تمیز است، به طوری که هر فرد دیگری بتواند تنها با نگاه کردن به کد شما، مفهوم آن را درک کند. تمیز نویسی کد، در دروسی مانند برنامه‌سازی پیشرفته در نیم‌سال آینده ضروری و الزامی است، و لازم است که از اکنون به تمیزنویسی در برنامه‌ها عادت کنید. با کدنامه همراه باشید.

تقویم درس در هفته‌ی پیش‌رو

۸-۹ آبان

مهلت تمرین ۲ و انتشار تمرین ۳ – شرط و حلقه

۹ آبان

دستورات شرط و حلقه

۱۱ آبان

تابع

چرا کدی که به ظاهر درست است، به خطای Wrong Answer می‌خورد؟

خطاهای برنامه‌هایی که می‌نویسیم، به چند دسته تقسیم می‌شوند. تعدادی از این خطاها، خطاهای «زمان کامپایل» هستند که ناشی از عدم رعایت قوانین زبان (مثلا بزرگ نوشتن حرف اول for و به اشتباه For نوشتن آن، یا عدم قرار دادن سمی‌کالن (;) در انتهای خطوط و...) بوده و هنگام اجرای برنامه، توسط کامپایلر به ما گوشزد می‌شوند و در نتیجه رفع آن‌ها با تسلط بر قوانین زبان، آسان است. تعدادی خطا از نوع زمان–اجرا (Runtime) بوده و ناشی از عملیات غیرمجاز محاسباتی (مانند تقسیم عددی بر صفر) هستند که با نگاه به کد، غالبا به راحتی پیدا می‌شوند.

سومین و پیچیده‌ترین نوع خطاها، خطاهای «منطقی» هستند که یا ناشی از algorithm نادرست برای حل یک مسئله بوده و یا ریشه‌ی آن‌ها، استفاده‌ی ناصحیح از امکانات زبان و یا افتادن در دام‌های متداول برنامه‌نویسی است که در صفحه‌ی ۳ این شماره از کدنامه به آن پرداخته‌ایم.

تمیز بنویسیم!

متین داغیانی

کد کثیف، کد تمیز

تا حالا پیش آمده که یک متن به خصوص را بخوانید و از همان اول از کارتان پشیمان شده باشید؟ اشتباهات املایی فراوان، جملات طولانی، فاصله‌ی کم خطوط، اندازه‌ی کوچک متن و... همان چیزهایی هستند که باعث شده‌اند احتمالا جواب شما به این سوال مثبت باشد. برعکس آن هم درست است. شاید تعداد انگشت‌شماری از متون هم در ذهنتان باشند که وقتی آنها را می‌خواندید، به معنای واقعی کلمه از آن لذت برده باشید، یا به قول معروف با آن ارتباط برقرار کرده باشید.

برنامه‌نویسی هم چندان بی‌شباهت به نوشتن نیست. فرقی نمی کند که یک برنامه‌ی ساده ی چند خطی باشد، یا یک پروژه‌ی بزرگ. چیزی که یک برنامه نویس معمولی را از یک برنامه نویس خبره جدا می‌کند، توجه به همان جزئیاتی است که باعث تفاوت تجربه شما از خواندن یک متن اعصاب خرد کن و یک شاهکار ادبی می‌شود.

چرا باید تمیز کد بزنیم؟

بتوانیم کدمان را بخوانیم!

یکی از تجربیات ناخوشایند هر برنامه‌نویس آن است که بعد از مدتی دوباره به سراغ برنامه‌ای که نوشته است برود و خود را میان انبوهی از خطوط بی‌معنی پیدا کند که حتی خودش هم متوجه نشود که چطور آنها را نوشته است! به جرئت می‌توان گفت که رعایت برخی اصول نه چندان پیچیده، کمک می‌کنند تا این تجربه‌ی تلخ، کم‌تر برایمان پیش بیاید.

دیگران بتوانند کدمان را بخوانند!

برنامه‌نویسی، در بسیاری از مواقع، یک کار گروهی است و شما ناگزیر هستید با دیگران تعامل و همکاری داشته باشید. یکی از مواردی که باعث شکست بسیاری از پروژه‌های بزرگ تیمی می‌شود، عدم درک متقابل میان اعضا است. اگر نتوانید متوجه شوید که کدی که همکارتان نوشته است چگونه کار می‌کند، هرگز نخواهید توانست مشکل آن را برطرف کنید و یا قابلیت‌های آن را افزایش دهید.

کد کثیف، محکوم به نابودی است!

نادیده گرفتن جزئیات و نکات ایمنی و ساختمانی، همان عاملی است که باعث می‌شود یک ساختمان به ظاهر زیبای تازه‌ساز، بعد از چند سال، تبدیل به یک مخروبه شود. حتی اگر بتوان قسمتی از آن را با هزینه‌ی گزاف، بازسازی و تعمیر کرد، مشکل از جایی دیگر سر در می‌آورد؛ درست مانند کد کثیف!

بر خلاف آن، داشتن چارچوب و رعایت قواعد کد تمیز، همان چیزی است که مسیر توسعه‌ی یک برنامه را هموارتر می‌کند و به طور چشم‌گیری از هزینه‌های بالقوه می‌کاهد.

با کدنامه، تمیز بنویسیم!

حالا که در حال فراگیری برنامه‌نویسی هستیم، خیلی بهتر است که در کنارش اصولی را نیز یاد بگیریم که به ما کمک می‌کنند تا بهتر کد بزنیم؛ یا به عبارت دیگر، **تمیز بنویسیم**.

در شماره‌های آینده‌ی کدنامه، مطالبی در اختیارتان قرار خواهند گرفت تا نکات موسوم به Clean Code را به طور خلاصه و جامع با هم مرور کنیم. در این شماره از کدنامه نیز تعدادی از نکات مهم در کدنویسی تمیز را با یک‌دیگر می‌بینیم.

چند نکته‌ی مهم در کدنویسی تمیز

متغیرها، اسم‌اند!

سعی کنید در نام‌گذاری متغیرهایتان از اسم‌ها استفاده کنید. استفاده از فعل یا صفت به تنهایی توصیه نمی‌شوند. هم‌چنین، اسامی مخفف یا خاص را در نام‌گذاری به کار نبرید.

```
int best_number;           // Correct!
int best;                  // Wrong!
int the_sample_result;     // Correct!
int smplerslt;             // Wrong!
```

اسم‌های معنادار، بهترین راهنماها

بر روی نام‌گذاری متغیرها فکر کنید. سعی کنید از اسامی یا گروه‌های اسمی استفاده کنید که بتوانند به این سوالات پاسخ دهند:

- چرا تعریف شده است؟
- چه کار می‌کند؟

```
int d;                     // Wrong!
int elapsed_time_in_days; // Correct!
```

از نام‌های طولانی نترسید!

طولانی بودن نام متغیر، بهتر از بی‌معنا بودن آن است. در یک برنامه شاید صدها یا هزاران متغیر وجود داشته باشند و اگر تمام آن‌ها را با حروف الفبا یا ترکیب آنها با اعداد تعریف کرده باشید، شاید بعدا نتوانید متوجه شوید که کارکرد متغیر چیست و چه اطلاعاتی را در خود ذخیره می‌کند. البته نباید در

این موضوع، زیاده‌روی کنید. بر اساس استانداردها، بهتر است طول نام هر متغیر، کمتر از ۳۱ کاراکتر باشد. به عنوان مثال، اگر متغیری قرار است حاصل جمع متغیرهای a و b را در خود نگه دارد، استفاده از sum_of_a_and_b بهتر از استفاده از s به عنوان نام است.

جدا کردن کلمات

در زبان‌های مختلف برنامه‌نویسی، انواع مختلفی از قواعد برای جداکردن واژه‌های تشکیل‌دهنده نام متغیرها وجود دارد. در بیشتر کتابخانه‌های استاندارد زبان C، از **حروف کوچک** برای تعریف متغیرها و از کاراکتر _ برای جداکردن واژه‌ها استفاده شده است. به عنوان مثال، استفاده از my_new_variable را به myNewVariable یا mynewvariable، ترجیح می‌دهیم.

عبارات شرطی و حلقه‌ها

عبارت‌های شرطی از پراستفاده‌ترین جملات در مکالمات عادی و روزانه ما و هم‌چنین زبان‌های برنامه‌سازی هستند. حلقه‌ها نیز از پررنگ‌ترین ویژگی‌های هر زبان برنامه‌نویسی محسوب می‌شوند که امکان مدیریت عملیات پی‌درپی و متوالی را در اختیار توسعه‌دهنده‌ها قرار می‌دهند. البته این ابزارهای مفید می‌توانند زمانی که با تعداد بسیاری پرانتز، آکولاد، حلقه‌های تو در تو و... دست و پنجه نرم می‌کنید، بسیار گیج‌کننده نیز باشند!

شرط‌های پرمعنا

در زبان C، اعداد ۰ و ۱ علاوه بر اینکه به عنوان متغیرهای int ظاهر می‌شوند، می‌توانند توصیف‌کننده‌ی متغیرهای صحیح/غلط نیز باشند. در واقع مقدار **صفر** به معنای غلط و مقدار **یک** به معنای درست است. برای آنکه هنگام خواندن یک عبارت شرطی بتوانیم این دو کاربرد را از یک‌دیگر متمایز کنیم، به این صورت عمل می‌کنیم که اگر قصد مقایسه‌ی یک عدد با عدد صفر را داشتیم، حتما عبارت a == 0 را به !a ترجیح می‌دهیم، اما اگر متغیر ما، یک متغیر حالت بود (برای مثال، found)، از آن در عبارت if به شکل if(!found) و نه به شکل if (found == 0) استفاده می‌شود.

فاصله‌گذاری اجتماعی در اول خط!

برای تمایز میان قسمت‌ها و بلاک‌های مختلف (در شرط‌ها، حلقه‌ها و...)، بهتر است از تورفتگی‌های مناسب استفاده کنیم. به دو مثال زیر در زمینه استفاده از indent در ابتدای خط، توجه کنید:

مثال غلط:

```
if (a == 5) {

printf(“Hello!”);

}
```

مثال درست:

```
if (a == 5) {

       printf(“Hello!”);

}
```

شرط‌های مرکب

از جمله مواردی که می‌تواند به شدت کدمان را ناخوانا کند، شرط‌های مرکب یا چندخطی است. برای آنکه بتوانیم بهتر آن‌ها را تحلیل و بررسی کنیم، بهتر است هر شرط را در یک خط بنویسیم (**تمام** عملگرها باید یا در ابتدا و یا در انتهای خطوط آورده شوند).

```
if (condition1 ||
    (condition2 && condition3) ||
    condition4) {

       printf(“Hello!”);
}
```

به این نکته هم توجه داشته باشید که به طور کلی، بهتر است از به کار بردن شرط‌های طولانی و پیچیده، اجتناب کنید و سعی کنید تا جای ممکن، با اصلاح منطق برنامه‌تان، آن‌ها را ساده‌تر کنید.

مارپیچ شرط‌ها!

اگرچه در استفاده از شرط‌ها، حلقه‌ها و ترکیب آن‌ها با یک‌دیگر، محدودیتی وجود ندارد، با این حال بهتر است تا جای ممکن از عبارات تو در تو و طولانی پرهیز کنیم. در بسیاری از اوقات، می‌توان با کمی فکر، راه مناسب‌تری برای ساده‌تر کردن کدمان پیدا کرد. یکی از این راه‌ها، استفاده از توابع است که با آن‌ها در آینده آشنا می‌شوید.

ارتباط با کدنامه

خوش‌حال می‌شویم اگر پیشنهادات و انتقاداتی نسبت به کدنامه دارید یا سوال خاصی دارید که تمایل دارید در کدنامه پاسخ داده شود، به دستیاران آموزشی درس اطلاع دهید.

اشتباهات متداول کدزنی – بخش اول

مهدی لطفیان

سمی‌کالن‌های خطرناک!

یکی از متداول‌ترین اشتباهات در میان برنامه‌نویسان مبتدی، استفاده‌ی اشتباه از سمی‌کالن و دریافت نتیجه‌ی ناخواسته است. در صورت فراموشی سمی‌کالن در انتهای خط‌های دستوری عادی، با خطای کامپایل مواجه شده و به راحتی مشکل را حل می‌کنیم، اما گاهی مشکل از یک خطای کامپایل ساده فراتر می‌رود؛ برای مثال، قطعه‌کدهای زیر را در نظر بگیرید:

```
for (int i = 0; i < 10; i++);

    printf("Hi");
```

در این‌جا، احتمالا قصد داشتیم که ۱۰ بار عبارت Hi را چاپ کنیم، ولی در نهایت فقط یک بار چاپ می‌شود؛ زیرا با گذاشتن سمی‌کالن بعد از for، در واقع به پایان رسیدن حلقه را اعلام کرده‌ایم و دستور پایین، عضوی از بدنه‌ی حلقه نخواهد بود. گاهی وضع از این هم خراب‌تر می‌شود:

```
int a = 0;

while (a < 10);

        a++;
```

در قطعه کد بالا، انتظار داریم که حلقه ۱۰ بار اجرا شده و سپس پایان یابد، ولی به دلیل سمی‌کالن بعد از while، هیچ وقت به خط پایین نخواهیم رسید و در حلقه گیر خواهیم کرد، زیرا دستوری در حلقه نیست که مقدار a را زیاد کند و a همواره صفر مانده و هرگز از حلقه خارج نخواهیم شد!

گاهی هم سمی‌کالن‌ها از حد مورد نیاز کمتر هستند؛ در قطعه کد زیر، برنامه‌نویس قصد داشته تا بدون انجام کاری از تابع خارج شود، اما به دلیل فراموشی سمی‌کالن بعد از return، در واقع تابع مقدار متغیر a را برمی‌گرداند (به زودی با تابع در درس آشنا خواهید شد).

```
int someFunction() {

        int a;

        return

        a = 10;

}
```

اولویت عملگرها

هر عملگر در زبان C، یک اولویت دارد که باعث می‌شود نسبت به عملگرهای محاسباتی دیگر، زودتر یا دیرتر اجرا شود که لیست کامل آن‌ها را می‌توانید از این‌جا مشاهده کنید؛ اما حفظ کردن و رعایت کردن همیشگی این ترتیب‌ها، کاری طاقت فرسا است و در بسیاری از مواقع منجر به ایجاد خطاهای منطقی در کدهای برنامه‌نویسان می‌شود. راحت‌ترین راه برای جلوگیری از این مشکل آن است که به کمک پرانتزگذاری، عملیاتی که می‌خواهیم زودتر انجام شود را مشخص کنیم تا خطاهای غیرمنتظره‌ای دریافت نکنیم.

نقیض‌های دردسرساز

فرض کنید می‌خواهید یک حلقه while نوشته و تا زمانی که متغیر a از ۵ کوچک‌تر شود یا متغیر b زوج شود، در حلقه بمانید و به محض برقراری شروط فوق، از حلقه خارج شوید. شما شرط درون حلقه را چگونه می‌نویسید؟ بسیاری از برنامه‌نویسان مبتدی، هر تیکه را جدا نقیض کرده و شرطی مانند زیر می‌نویسند:

```
while (a >= 5 || b % 2 != 0) {

    // some code

}
```

اما اگر از آمار و احتمال پایه‌ی یازدهم به یاد داشته باشید، می‌دانید که برای نقیض کردن چنین عبارت ترکیبی‌ای، نباید هر بخش آن را جداگانه نقیض کنید، بلکه باید طبق **قانون دمورگان** عمل کنید؛ در نتیجه، شکل درست شرط حلقه، به یکی از دو صورت زیر خواهد بود:

```
while (a >= 5 && b % 2 != 0) {

    // some code

}
```

یا:

```
while (!(a < 5 || b % 2 == 0)) {

    // some code

}
```

حلقه‌های بی‌پایان

یکی از اشکالاتی که خیلی از برنامه‌نویسان – چه مبتدی و چه باسابقه – با آن برخورد می‌کنند، گیر کردن کد در حلقه‌هایی است که هیچ‌گاه تمام نمی‌شوند. یک نمونه از این حلقه‌ها را در قسمت «سمی‌کالن‌های

خطرناک» دیدیم. غالبا این خطا هنگامی رخ می‌دهد که یا فراموش کنیم شرط اتمام حلقه را فراهم کنیم، یا آن را در لایه‌های پیچیده‌ی شرطی که هیچ وقت اجرا نخواهد شد قرار دهیم. برخی از IDEهای پیشرفته، در صورت یافتن این اشکالات، به شما اخطار می‌دهند، اما بسیار خوب است که همواره خود به این نکته دقت کنیم که که وقتی یک حلقه می‌نویسیم، از پایان یافتن درست آن، مطمئن شویم.

علامت‌دار و بی‌علامت

در بسیاری از مواقع و در حین استفاده‌ی هم‌زمان از متغیرهای صحیح signed int و unsigned int و انجام عملیات منطقی و ریاضی روی آن‌ها، به خطاهای منطقی عجیبی برخورد می‌کنیم:

```
unsigned int a = 1000;

signed int b = -1;

if (a > b) {

	printf("a is bigger than a");

} else {

	printf("b is bigger than a");

}
```

در این مثال با توجه به بزرگ‌تر بودن ۱۰۰۰ از ۱- انتظار می‌رود که عبارت a is bigger than b چاپ شود، ولی در هنگام اجرای برنامه، عبارت b is bigger than a چاپ می‌شود. ساده‌ترین راه جلوگیری از این مشکل، استفاده‌ی حداقلی از این دو نوع متغیر به طور هم‌زمان است. اگر به خاطر شرایط خاص یک مسئله، مجبور به استفاده‌ی هم‌زمان از این دو نوع عدد صحیح شدیم، باید قبل از انجام عملیات منطقی و ریاضی بین آن‌ها، خودمان تبدیلات لازم را انجام دهیم تا کامپایلر ما را غافلگیر نکند.

کنجکاوی: به نظر شما، دلیل این خروجی چیست؟ (راهنمایی: به شماره‌ی ۲ کدنامه و نیز بخش محاسبات در کامپیوتر از درس رجوع کنید)

مقایسه‌ی float و double

قطعه کد زیر را در نظر بگیرید:

```
float n = 0.1;

if (n > 0.1) {

	printf("Hi");

}
```

پس از اجرای کد فوق، بر خلاف انتظار ما، کامپایلر خروجی Hi را چاپ می‌کند و گویی ۰.۱ را بزرگ‌تر از ۰.۱ در نظر گرفته است! قبل از شک در خصوص بنیان‌های درس ریاضی که ۱۲ سال و اندی در حال مطالعه‌ی آن هستیم، لازم است بدانیم که عدد ۰.۱ ای که در صورت شرط آمده است، در کامپایلر به طور خودکار به عنوان double در نظر گرفته می‌شود و در مقایسه‌ی آن با یک float، به دلیل تفاوت در دقت ذخیره‌سازی اعداد که در دروس بعد با آن بیش‌تر آشنا می‌شوید، رایانه دچار خطا شده و بدنه‌ی شرط، اجرا خواهد شد. ساده‌ترین راه جلوگیری از این اشکال، استفاده از double به جای float است، مگر در مواقعی که با محدودیت حافظه سر و کار داریم که باید در چنین موقعیت‌هایی دقت فراوان داشته باشیم.

یکی بیش‌تر!

در بسیاری از اوقات، شرط حلقه به صورت a < n در نظر گرفته می‌شود و نه a <= n؛ در صورت بروز حالت اول، لازم است توجه کافی داشته باشیم که متغیر a، مقادیر تا **قبل از** مقدار n را اخذ خواهد کرد و نه خود مقدار n، ولی در حالت دوم، متغیر a برابر خود مقدار n نیز خواهد شد.

استفاده‌ی جابه‌جای break و continue

می‌دانیم که برای خروج از حلقه در حین اجرای آن، باید از کلیدواژه‌ی break و برای بازگشت به اول حلقه و اجرای دور بعد باید از continue استفاده کرد. استفاده‌ی جابه‌جای این دو کلیدواژه ممکن است در میان تازه‌کاران سبب بروز خطا شود. هم‌چنین، لازم است توجه داشته باشید که عبارت بخش **سوم** حلقه‌ی for پس از اجرای continue اجرا می‌شود، ولی پس از اجرای break، اجرا نشده و بلافاصله اجرای برنامه به خط پس از اتمام حلقه منتقل می‌شود. عدم توجه به زمان اجرای بخش سوم for، گاه سبب بروز خطا در برنامه‌ها می‌شود.

سایر خطاهای جزئی

تعدادی از خطاها، به خصوص در زبان C و به خصوص در میان برنامه‌نویسان مبتدی‌تر، متداول‌تر از دیگر خطاها هستند. در ادامه، لیستی از سه مورد بسیار متداول آن‌ها ذکر شده تا هنگام کدزنی به آن‌ها دقت کرده و در صورت برخورد با خطاهای عجیب، رعایت شدن آن‌ها را بررسی کنید.

- فراموشی علامت **&** در هنگام ورودی گرفتن با تابع `scanf()`
- استفاده از **=** به جای **==** در شرط دستورات شرطی و حلقه
- استفاده از مقادیر متغیرهایی که مقداردهی نشده‌اند