כתיבה בשפת C – סטנדרטים לכתיבת קוד

קריאות של קוד מתייחסת לקלות שבה ניתן להבין מה קוד עושה מעיון בו. חישבו על מתכנת שנאלץ להיכנס לקוד שמישהו אחר כתב ולפענח מה הוא מבצע וכיצד (למשל כי התגלה בו באג). הדבר קורה באופן יומיומי בתעשייה ועל כן נושא זה חשוב ואנחנו רוצים לספק לחברינו ולעצמנו קוד ברור ונוח. בנוסף, קוד קריא מונע טעויות ומקל על תחזוקת התוכנית. המטרה היא שהקוד ייקרא בקלות ויסביר את עצמו, כמו סיפור ☺

בתעשייה, כשקבוצת מתכנתים עובדת על פרויקט משותף הם חייבים לעבוד לפי סטנדרטים אחידים וקבועים מראש של תכנות, אפילו לפרטים ויזואליים בסגנון הכתיבה. בקורס זה קריאותו של הקוד תקל על הבודקים להבין היכן הטעות הלוגית ולהוריד נקודות על הקטע הרלוונטי בלבד. בקורס נאכוף סטנדרטים של סגנון כתיבת קוד. ככלל, **יש להקפיד על אחידות בקוד**, גם עבור נושאים שלא זכו להתייחסות מפורטת להלן.

ישנם מספר כללים פשוטים שעליכם לעמוד בהם (שימו לב להערה של כל דוגמה, כדי להבין אם זו דוגמה טובה או רעה):

1. תיעוד

1. תיעוד ראשי - מסביר מה כל התוכנית/ספריה עושה. בקורס תיעוד זה יופיע לרוב בקובץ ה README, ורק במקרה בו התוכנית מכילה יותר מקובץ בודד.
2. תיעוד בסגנון **Doxygen** לכל פונקציה ו-`struct`, כולל פרמטרים ומשמעותם, ערך החזר ותיאור קצר של פעולת הפונקציה. יש להוסיף הערה אשר מתחילה ב `/**` ומסתיימת ב `*/` כאשר כל שורה ביניהם מתחילה בכוכבית. בקורס זה, אם ההצהרה על פונקציה נפרדת ממימושה, נקפיד על תיעוד הפונקציה במקום בו אנו מצהירים עליה (לרוב קובץ ה header). ניתן להוסיף תיעוד נוסף הקשור לפרטי מימוש (למשל, החלטה על מימוש באמצעות אלגוריתם מסויים) במקום בו הפונקציה ממומשת.

   לקריאה נוספת ראו http://en.wikipedia.org/wiki/Doxygen, וכמובן באתר הרשמי http://www.stack.nl/~dimitri/doxygen.
3. הערות להסבר של קטעי קוד לא טריוויאליים שיעזרו להבנת הקוד.
4. שימו לב שההערות דרושות בכדי להבין את האלגוריתם ומי שקורא את הקוד גם הוא (בתקווה) מתכנת, לכן החשיבות שלהן היא בקטעים מסובכים יותר ואין צורך לכתוב הערות על פקודות מובנות. דוגמא להערה מיותרת:

```
scanf("%d", &nNumber); // get a number from the user
```

2. **אורך פונקציה**. אם פונקציה צומחת להיות ארוכה מדי, יש לפצל אותה לקריאות לתת פונקציות. כאן יש מקום לשיקול דעת ולניסיון. אם אורך הפונקציה נגרם, למשל, מבלוק של switch עם הרבה מקרים, אז זה בסדר ובלתי נמנע (וגם לא מפריע, כי זה יחסית מסודר ומובן), אבל אם הפונקציה מורכבת משלבים נפרדים שיכולים להיות מופרדים לתת פונקציות, יש לפצל אותה. בכל אופן, התוצאה הסופית תמיד צריכה להיות מסודרת, ברורה וקריאה.
3. **יש להימנע משכפול קוד**. אם יש קטע קוד משותף שחוזר על עצמו מספר פעמים הוא צריך כנראה להיות מוצא לפונקציה משותפת שתיקרא מכמה מקומות שונים.
4. **שמות משתנים**
   1. הקפידו לתת שמות משמעותיים למשתנים. על השם לשקף:
      1) את תפקידו של המשתנה בצורה הברורה ביותר
      2) טיפוסו - בד"כ ייגזר מהשם בקלות אם התפקיד מוגדר היטב

      לדוגמא:

```
int         numOfCakes;
char*       name;
float       randomNumber;
```

   2. יוצאי דופן להערה האחרונה הם משתנים מונים ללולאות, אינדקסים של קואורדינאטות...
   3. קבועים (const וגם define) יופיעו באותיות גדולות. אם השם מורכב מכמה מלים הן יהיו מופרדות ע"י קו תחתון.

```
#define     DIMENSION   3
#define     MAX_LENGTH 10

const double DOLLAR_RATE = 3.6;
```

   4. לרוב שם יהיה מורכב מיותר ממילה אחת, כדי להבין את משמעותו. בשמות משתנים שאינם קבועים כל מילה חדשה (חוץ מהראשונה) תתחיל באות גדולה (גרסא של שיטה שנקראת camelCase).

      לדוגמא:

```
int   sum;              // hmmm... sum of what?
int   sumOfSalaries;    // better ☺
```

   5. על משתנים גלובלים (שאינם קבועים) להתחיל באות g ולאחר מכן שם המשתנה יתחיל באות גדולה.
      לדוגמא:

```
int   gTotalCount;
```

5. **שמות פונקציות**

1. גם לפונקציה יש לתת שם משמעותי המצביע על תפקידה.
2. שם פונקציה יתחיל באות קטנה וכל מילה חדשה תתחיל באות גדולה (ללא קו תחתון).

```
// bad:
void print_message(char *s)
{
   printf("message: %s\n",s);
}

// bad:
void PrintMessage(char *s)
{
   printf("message: %s\n",s);
}

// good:
void printMessage(char *s)
{
   printf("message: %s\n",s);
}
```

6. **שמות של קבצים, מחלקות וטיפוסים** שמוגדרים באמצעות typedef יתחילו באות גדולה, וכל מילה חדשה תתחיל גם באות גדולה. כמו כן, גם שם של struct יהיה בנוי באופן דומה.

```
typedef struct PrivateCar
{
      ...
} PrivateCar;
```

7. **הכרזה על פונקציות.** ישנם דרכים חוקיות רבות ל forward declaration של פונצקיות. עם זאת, אנו נקפיד לרשום את ההכרזה על הפונקציה בדיוק כמו בקוד שמגדיר את מימושה. כך, שמות טובים למשתנים ופונקציות יעזרו להבין מה הפונקציה עושה רק משורת ההצהרה שלה, ויהיה קל יותר להתאים את המימוש להצהרה. למשל, בקוד הבא

```
int foo(int a); // good
int foo(int);   // bad
int foo();      // bad
...
int foo(int a)
{
   return a;
}
```

8. **מבני בקרה (תנאים ולולאות)** תמיד יופיעו עם סוגריים מסולסלים, גם אם יש בהם פקודה אחת. הדבר מקל משמעותית על קריאת הקוד. חשוב מכך, הדבר מונע שגיאות חמקמקות, בעיקר כאשר מוסיפים עוד פקודה בתוך התנאי או הלולאה בשלב מאוחר יותר.

```
// bad:
if (!winLottery)
   printf("oh no\n");

// good:
if (winLottery)
{
   printf("yey\n");
}
```

9. **סוגר מסולסל** הפותח בלוק יופיע בשורה חדשה, מעומד מתחת למילה הפותחת את ראש הבלוק, ולא בסוף השורה שלפני הבלוק.

```
// bad:
void printMessage(char *s) {
   printf("message: %s\n",s);
}

// good:
void printMessage(char *s)
{
   printf("message: %s\n",s);
}
```

10. **עימוד** יעשה ע"י התו tab (ולא ע"י רווחים). הפקודות בתוך כל בלוק (מובחן ע"י סוגריים מסולסלים) יופיעו בהטיה של טאב בודד פנימה. ראו דוגמאות לעיל ובקוד המצורף.

11. **אורך שורה** לא יחרוג מעבר ל 100 תווים.

12. **אין להשתמש בליטראלים מספריים** (ערכים מספריים מפורשים) בתוך הקוד הלוגי. במקום זה יש להשתמש בקבועים (const או #define) בעלי שם משמעותי ואינפורמטיבי, שמתאר את מה שהקבוע מייצג (ולא את הערך המספרי שלו).

```
// bad.
// Reading this we don't understand what's so special about 200:
if (strlen(s) > 200)
...

// still bad. And a bit ridiculous:
#define TWO_HUNDRED 200
...
if (strlen(s) > TWO_HUNDRED)
...

// good. Now we know what this 200 means:
#define MAX_LINE_LENGTH 200
...
if (strlen(s) > MAX_LINE_LENGTH)
...
```

13. **פרמטר קבוע בפונקציה, ופונקציה קבועה.** אם מגדירים פונקציה שלא אמורה לשנות את ערכו של אחד מהפרמטרים שלה, הפרמטר צריך להיות מוגדר כ const. אלמנט זה יילמד במהלך הקורס ומאז צריך להקפיד על כך. באופן דומה לגבי פונקציה שמוגדרת כ const וכן לגבי ערך ההחזרה של פונקציה.

14. **אין להשתמש בתווי UNICODE** בתוך קבצי הקוד.

15. **ערכים מספריים**. כשמשימים ערך מספרי שלם לתוך משתנה מטיפוס double יש להוסיף את הנקודה העשרונית, כדי להבהיר ולוודא שהערך צריך לקבל ייצוג וטיפול של טיפוס double ולא טיפוס int. כשמשימים ערך מספרי לתוך משתנה מטיפוס float יש להוסיף לסוף הליטראל המספרי את הסימן f, שמבהיר שזהו ערך float ולא double.

```
const double DOLLAR_RATE0 = 3; // bad. Confusing with int
const double DOLLAR_RATE1 = 3.0; // good.
const float DOLLAR_RATE2 = 3.6; // bad. Confusing with double
const float DOLLAR_RATE3 = 3.6f; ; // good.
```

16. **עימוד פונקציה ארוכה.** פונקציה המקבלת מספר רב של פרמטרים החורג מאורך שורה מותר, יש להמשיך בשורה נפרדת ובעימוד כך שהקוד ימשיך מתחת לרשימת הפרמטרים.
אותו הדבר נכון לרשימת תנאים ארוכה.

```
void functionB(int a, int c,
                    int d) // bad.
void functionC(int a, int b,
                int c); // good.
```

17. **ריווח בין שורות.** לפני קטע קוד חדש תופיע שורת רווח (ראו שורת הרווח בדוגמא בין good ל-bad לעיל). דאגו לרווח שורות וגם בתוך שורה, כך שהקוד יהיה כמה שיותר ברור וקריא.

18. **רווחים בין תווים.** לפני ואחרי אופרטור בינארי יופיע רווח (אופרטור בינארי כולל אופרטור השמה =). לאחר פסיק (ברשימת פרמטרים) או נקודה פסיק (בלולאת for) יופיע רווח.
לאחר שם פונקציה בקריאה לפונקציה לא יופיע רווח. לאחר שם מערך בגישה למערך לא יופיע רווח.
לאחר מילת מפתח keyword יופיע רווח (למשל בהצהרה על לולאת for או תנאי if).

```
for (a;b;c);  // Wrong. It should be (a; b; c)
hello(a,b,c); // Wrong. It should be (a, b, c)
int k = 2+3;  // Wrong. It should be 2 + 3
int k=5;  // Wrong. It should be k = 5;

int k = (2 + 3); // OK.
int k = -2;      // OK. Minus can be used
for (a; b; c);   // OK
hello(a, b, c);  // OK
tt[c++];         // OK. Unary operator is used in the [ ( [
```

כתיבה בשפת CPP – סטנדרטים לכתיבת קוד

כל ההנחיות לעיל (עבור שפת C) הינן בתוקף. בנוסף:

1. **קונבנציית שמות:**
   - בדומה לשמות קבצים, שם של מחלקה יתחיל באות גדולה. אם השם מורכב ממספר מילים, כל מילה תתחיל באות גדולה. לדוגמא:

```
Cat, WashingMachine
```

   - שם של data member לא סטטי של המחלקה יתחיל בקו תחתון. לדוגמא:

```
_personName
_x
```

   - שם של data member סטטי של המחלקה יתחיל באות s וקו תחתון. לדוגמא:

```
s_dataMemberName
```

   - שמות של מתודות ופונקציות יתחילו באות קטנה, וכל מילה נוספת תתחיל באות גדולה. לדוגמא:

```
getX
calcComplicatedThing
```

   - שם של private method של המחלקה יתחיל בקו תחתון. לדוגמא:

```
_calculateReference
```

2. **חלוקה לקבצים:**
   - בכל קובץ header תופיע הגדרה של מחלקה אחת בלבד (למעט מקרים מיוחדים).
   - קובץ CPP יכיל את כל מימוש המחלקה (למעט פונקציות inline אשר ימומשו בקובץ ה header).
   - שם הקובץ יהיה שם המחלקה בתוספת הסיומת המתאימה (ראה להלן).
   - **קובץ ה header (עם סיומת h.)**:
     - לפני מחלקה תהייה הערה המתארת אותה באופן כללי (מה היא מייצגת ותפקידה), והערות על members של המחלקה ירשמו ליד כל member בהתאם (<u>כמו בקובץ לדוגמא</u>).
     - ניתן <u>לחילופין</u> לרכז בראש הקובץ את תיאור המחלקה הכללי וה members שלה (פונקציות ושדות) ,תוך דגש על ממשק השימוש במחלקה. כלומר, תיאור קצר של כל הפונקציות של המחלקה המוגדרות כ- public.

במקרה זה יש לבצע את התיעוד על ידי שימוש בפקודות doxygen מתאימות (special commands), כגון:

```
@class, @fn, etc...
```

למשל, על מנת לתעד את האופרטור ! של המחלקה Foo בראש הקובץ ניתן לכתוב בתחילת הקובץ:

```
/**
 * ... <previous comments>
 *
 * @fn Foo::operator!() const
 * @brief Info about the operator
 * @return What is returned
 *
 * ... <more comments>
 */
```

- זכרו! קובץ זה מיועד למשתמש של המחלקה אותה אתם כותבים, לכן הימנעו מפרטים טכניים מייגעים והתמקדו בנושאים הקשורים בממשק ומעניינים את המשתמש במחלקה (למשל, סיבוכיות מקום וזמן ריצה יתוארו, אבל לא האלגוריתם עצמו).
- בכל class תופיע לכל היותר כותרת אחת מסוג public, private, או protected. כך, כל ה members מאותו סוג ירוכזו יחדיו. הכותרת תופיע גם אם היא ה default (למשל, class לא יתחיל בשדות מסוג private ללא כותרת).

- **קובץ ה CPP (מימוש, עם סיומת cpp.)**:
  - בראש הקובץ תהייה הערה כללית קצרה על תוכן הקובץ והמחלקה המיושמת בו).
  - לפני כל מתודה תופיע הערה ספציפית למתודה זו בהתאם לצורך, כהערה כללית או כהערת doxygen. הפעם אנו פונים למתכנת העובר על הקוד שלנו, ולכן התיעוד יתמקד בנושאים הקשורים למימוש, ובמיוחד ב"טריקים" או פעולות שאינן ברורות. אם המתודה טריוויאלית (למשל, getX), אין צורך בהערה.

3. דגשים עיצוביים וטכניים:

- גישה ל data members של אובייקט מחוץ למחלקה יש לבצע דרך access methods ולא ישירות. גם בקוד שבתוך המחלקה עצמה אפשר לעשות זאת ולהסתמך על הקומפיילר שייעל את הפניה לשדה (אם המימוש של פונקצית הגישה הוא טריוויאלי, ולא כולל בדיקות מיוחדות, הקומפיילר ידע לוותר על הקריאה לפונקציה ולבצע גישה ישירה).
- שימו לב ל code reuse, ואל תממשו אותם חלקי קוד מספר פעמים.
- יש להשתמש בפונקציות סטנדרטיות של CPP במקום פוקנציות של C שנועדו לאותו שימוש (לדוגמא, יש להשתמש ב new ולא ב malloc).