מעבדה מתקדמת בשפת C

תרגיל מספר 2 - הנחיות עבודה וטיפים

מטרת המסמך הזה היא לספק לכם הסטודנטים הנחיות עבודה שבתקווה יקלו עליכם את פתרון התרגיל. ניתן לפתור את התרגיל במספר דרכים, וכל דרך שמסתיימת בקוד איכותי כפי שלמדנו, ושעובר את הטסטים, טובה באותה מידה. הדרך הזו היא פשוט, לדעתי, הכי קלה (או אולי הכי פחות קשה?)

איך מתחילים:

הכי חשוב: לא כותבים טונות של קוד בלי לבדוק אותו, ורק אז מתחילים לבדוק. כותבים קצת קוד, בודקים אותו, מצרפים טסט אוטומטי לאוסף הטסטים שלכם, חושבים על מקרה קצת יותר מתוחכם, בודקים אותו, מצרפים עוד טסט, וחוזר חלילה. אם עברה כבר רבע שעה שלמה ועוד לא בדקתם את הקוד שלכם - משהו פה לא בסדר, ואתם מבזבזים לעצמכם זמן יקר שהיה יכול במקום זה לשמש אתכם להכנת דו"ח המעבדה הקרוב!

לגבי טסטים: חייבים לעבוד עם קובץ שמריץ מספר רב של טסטים בצורה אוטומטית. כל דבר אחר הוא פתח לבזבוז קולוסלי של זמן בבדיקות ידניות. השתמשו בטסטים שלי כתבנית וחיסכו לכם כמה דקות של "הקמה ראשונית" של הטסטים, אבל בכל מקרה, הטסטים חייבים להיות אוטומטיים לחלוטין, ושלא יצריכו בדיקה בעיניים של הפלט שוב ושוב, מעבר ל"הכל תקין" או "משהו לא בסדר". הכי פשוט שאם הכל תקין, שום דבר לא יודפס למסך, כמו הטסטים שלי. בידקו את הקוד לאחר כל שינוי, וודאו שכל הטסטים עוברים - אל תמשיכו להוסיף סוויצ'ים במצב שיש טסטים שלא עוברים!

תשמעו מסטודנטים מסמסטרים קודמים:

1. פראפראזה על מה שסטודנט אמר לי בע"פ: "כן, סטודנטים תמיד לא מבינים למה לכתוב טסטים במקום כאילו לכתוב את כל הקוד מראש ולגמור עם זה, כי הם לא מבינים שתכל'ס זה יותר מהיר לעבוד עם טסטים ולכתוב את הקוד בשלבים".

2. קופי-פייסט ממייל שסטודנט שלח לי:

""""שלום נמרוד,

בהמשך לשיחה שלנו ממקודם אני חייב להודות ששיטת הטסטים עשתה לי פלאים בתרגיל בית 2.
...
ממה שראיתי בתרגיל הזה, לעבוד ללא טסטים בתרגיל הזה זה פשוט סיוט - ס-י-ו-ט. רק לחשוב על זה שאתה יכול להוסיף שורת קוד אחת שנראית לך תמימה ואז בטעות לדפוק לעצמך קוד שעבד עד עכשיו בלי להרגיש וממש מתחת לאף.

...
לכתוב טסטים תוך כדי העבודה על הקוד זה פשוט כיף (בייחוד כשמוסיפים את הקומפילציה לשם). כי אתה כותב קוד, מריץ את הקובץ test_run.sh, הקובץ עובר קומפילציה ומריץ את הדוגמאות שלך. וברגע שטיפלת בחלק המקרים, אתה כל הזמן מריץ ועושה בדיקה שהם עדיין עובדים - וזה פשוט כיף לראות שהוספת עוד ארגומנט והשאר עדיין עובדים.

מה גם שזה מוסיף לך הבנה עמוקה יותר של הקוד שאתה כותב. אתה יכול לשחק עם סדר השורות ולראות את ההתנהגות של הקוד.
אני רואה פה רק יתרונות, העניין היחידי הוא שצריך להתרגל לזה. זה כלי חזק לתכנון ובדיקה של קוד מרובה שורות וכדי לשלוט בו.
"""

אז יאללה, לעבודה! נתחיל מהמקרה הכי פשוט: קובץ של שורה אחת. זה די פשוט לטפל במקרה הזה - לקרוא שורה אחת מהקובץ עם getline, להריץ strncmp, וסיימנו. מוסיפים שני טסטים אוטומטיים שמוודאים שהפלט תקין גם אם יש התאמה וגם אם אין, ויופי.

עוברים ל**מקרה קצת יותר מסובך: כמה שורות**. עכשיו, השינוי העיקרי שצריך לעשות הוא להכניס את מרבית הקוד הקיים ללולאה. מוסיפים עוד טסט או שניים לאוסף, וסיימנו.

בשלב הזה כדאי להיזכר שלא בדקנו מקרה חשוב: **קובץ ריק**. הוא כבר אמור להיות מטופל בעזרת הקוד הקיים, אבל כדאי להוסיף אותו לאוסף הטסטים ולוודא שהוא עובר.

אחרי שטיפלנו במקרים שלא כוללים סוויצ'ים, כדאי להתחיל להוסיף סוויצ'ים, מהפשוט למסובך. אין מה להגיד - השילובים הפוטנציאליים בין הסוויצ'ים יכולים מאוד לסבך את הקוד, וזה תרגיל בכלל לא פשוט, בעיקר בגלל זה. אבל שמירה על כמה כללים בסיסיים תקל פה מאוד על החיים שלכם:

<u>כל פעולה לוגית שהקוד עושה, צריכה לקרות בדיוק במקום אחד בקוד, בפונקציה שנקראת לפי השם של הפעולה</u>. לדוגמה, הפעולה של לקרוא שורת קלט צריכה לקרות אך ורק בפונקציה אחת שכנראה נקראת read_line או משהו דומה. זאת למרות שלכאורה יש שני אופנים בהם אנחנו קוראים קלט: מקובץ או מ-stdin. הפונקציה צריכה להיות מסוגלת להתמודד עם שני האופנים האלה, כך שכשאתם צריכים לקרוא שורה, אתם פשוט קוראים לה וזהו.

פעולה בסיסית נוספת שאנחנו עושים היא בדיקת התאמה בשורה: גם היא צריכה לשבת אך ורק בפונקציה אחת, שכנראה תיקרא is_match_in_line. פה, בטח כבר שמתם לב שיש מספר רב של אופנים בהם אנחנו בודקים התאמה, לפי סוויצ'ים שונים ומשונים -

הפונקציה צריכה להתמודד עם כל האופנים האלה, כך שאתם פשוט קוראים לה, וזהו. היא כנראה צריכה לקבל את הסוויצ'ים שנעשה בהם שימוש בעזרת ארגומנט מסוג סטראקט, כדי שהיא תוכל להגיב אליהם.

נתחיל להוסיף סוויצ'ים, מהפשוט למסובך:

i- התכנית תתעלם מההבדל בין אותיות גדולות וקטנות

זה יחסית פשוט - או שמחליפים את strcmp בבדיקה ידנית יותר שמתחשבת ב-case רק אם צריך, או שמעבירים את השורה והביטוי לאותיות קטנות ואז קוראים ל-strcmp. באילו מהשיטות לבחור? רמז: איזה שיטה תגרום לקוד שסביר שישאר ברובו גם בגירסה הסופית של התרגיל?

כמובן שלאחר שטיפלנו בסוויץ' הזה, מוסיפים כמה טסטים אוטומטיים לאוסף.

v- הפוך את פעולת התכנית, כלומר הדפס רק את השורות שלא מכילות את הבטוי

גם פה, השינוי יחסית פשוט: כנראה שאיפשהו בפונקציה שקוראת ל-is_match_in_line צריך להכניס if שמגיב להאם יש התאמה, והאם יש -v. וכמובן, מוסיפים טסטים.

נעצור פה ונטפח לעצמנו על השכם בקטנה - כבר בשלב הזה, התמזל מזלנו ויש לנו תוכנית שעובדת, עם שני סוויצ'ים שלמים! לא רק שזה מרגש בפני עצמו, חסכנו לעצמנו המון חוסר וודאות של האם הקוד שלנו בכלל עובד, או שנצטרך להשקיע עוד ימים בדיבוג. זאת ועוד, כשיש קוד קונקרטי מול העיניים, הרבה יותר קל להבין איפה עושים את השינוי בשביל להכניס סוויץ' נוסף, בניגוד לשיטת ה"מתכננים הכל מראש" שמקשה למדי על העניין הזה.

השלב הבא: הוספת טיפול ב-stdin - בנוהל

שלב נוסף: c- במקום להדפיס את השורות יודפס רק מספר השורות בהן נמצא התאמה של הבטוי.

פה נשים לב שכנראה שצריך להוסיף פונקציה חדשה: אם עד עכשיו הקוד שלנו היה הקוד הכי פשוט שעובר את הטסטים, הוא כנראה קרא ל-printf כשהיה צריך להדפיס שורה. עכשיו, בעצם לא באמת צריך להדפיס אותה, אלא לדווח שיש התאמה (או אי התאמה, אם יש שילוב עם -v), והפונקציה שמטפלת בהתאמה צריכה להגיב לכך בהדפסה של השורה למסך אם יש צורך, או בעדכון מספר השורות המתאימות. שם לדוגמה לפונקציה הנ"ל: report_line_match. טיפ לחיים: סביר שכדאי לדווח על כל שורה האם היא מתאימה או לא מתאימה, כי במקרים שיש -v מעניינות אותנו גם שורות שלא מתאימות.

העבירו את המידע הזה בארגומנט בוליאני לפונקציה הנ"ל, וקבצו את כל הקוד שמטפל במה עושים כשיש התאמה או אי התאמה בשורה, בתוכה (וכמובן שהיא יכולה לקרוא לפונקציות נוספות אם היא מתארכת מעבר ל-30-20 שורות).

n- הדפס לפני כל שורה את מספרה בקובץ הקלט

אמור להיות יחסית פשוט בשלב זה

b- לפני כל שורה יודפס ההסט שלה מתחילת הקובץ ולאחריו נקודותיים

כנ"ל

x- הדפס רק שורות שלא מכילות דבר פרט לבטוי (התאמה הדוקה)

לא בהכרח מאוד פשוט, אבל: כמה מקומות צריך לשנות בקוד בשביל לטפל בזה? אם יצא לכם שהרבה יותר ממקום אחד, משהו לא בסדר...

A NUM- התכנית תדפיס גם NUM שורות אחרי השורה המכילה את הבטוי

זה כבר לא פשוט, אבל עדיין ניתן למימוש בלי המון עבודה: אחרי כל שורה בה יש התאמה, צריך לזכור להדפיס NUM שורות, ולעדכן את הערך הזה בכל פעם שיש התאמה. שימו לב שכל שינוי שאתם עושים צריך לשמור על ההנחיה שעושים כל דבר במקום אחד, בפונקציה שנקראית בהתאמה. הדבר מקל מאוד על התמצאות בקוד, ומונע מכם להגיע למצב ש"איבדתם את הידיים והרגליים", וספציפית שאתם חס וחלילה לא מסוגלים לדבג קוד שכתבתם - אם הגעתם למצב הזה, קחו רברס מהר!

סביר שכדאי בשלב הזה (אם לא קודם) להתחיל להחזיק שורות בסטראקט שכולל לא רק את הטקסט של כל שורה, אלא האם היתה בה התאמה או לא, ההיסט שלה מתחילת הקובץ, המספר שלה וכו'.

ביטויים רגולריים

רק בשלב הזה, אחרי שכתבנו כמות קוד גדולה ככל הניתן בלי להגיע לזה, כדאי להתחיל להתעסק בביטויים רגוליים. בינתיים צברנו היכרות עם עולם הבעיה, טסטים, וקוד שכבר מחולק לחלקים שאפשר להשתמש בהם.

אני מציע לטפל בביטויים רגולריים (כולל הנושא של הברחה) כך: מפרסרים את הביטוי למערך של סטראקטים, כשכל סטראקט מייצג אחד מהבאים:

א. תו רגיל

ב. נקודה

ג. סוגריים עגולים

ד. סוגריים מרובעים.

בשלב הזה אם עדיין אין לכם פונקציה פנימית בשם is_match_at_place שנקראת מתוך is_match_in_line, כדאי שתהיה כזו. המימוש הכי פשוט של is_match_at_place הוא רקורסיבי: בודקים האם יש התאמה של האיבר הבא בפרסור של הביטוי למקום הנוכחי בשורה, ואם כן קוראים ל-is_match_at_place עם שאר הביטוי ושאר השורה.

למרות כל האמור לעיל, גם פה, כדאי לעבוד אינקרמנטלית: לטפל קודם במקרה הכי פשוט - נקודה. אח"כ לעבור למקרה היותר מסובך - סוגריים מרובעים. רק אחרי כל זה, כדאי להגיע לחלק הכי קשה של סוגריים עגולים. שימו לב שעד שמגיעים לטיפול בסוגריים עגולים, אין באמת צורך ברקורסיה.

בהצלחה,
נמרוד