

כתיבת ספריות ב – C

בעולם התוכנה, קיימים מקרים רבים בהם קוד שנכתב עשוי לשמש תוכניות רבות. הדרך הפשוטה ביותר להשתמש בקוד כתוב היא כמובן לשתול אותו בתוך הקוד החדש. ברור ששיטה זו איננה אידיאלית ברוב המקרים מכיוון שהיא מקשה על תחזוקת התוכנה, מסרבלת את תהליך הקומפילציה ואיננה מודולרית.

אפשרות אחרת לשימוש חוזר בקוד הוא תרגומו לקובץ object ואז שימוש בו ע"י linking. כאשר מדובר בקובץ יחיד זו בהחלט אפשרות סבירה אך כאשר הקוד כולל שורות קוד רבות המחולקות באופן טבעי למספר רב של קבצים, מתבקש מנגנון אחר.

במקום לבצע linking להרבה קבצי object, ניתן ליצור קובץ יחיד המכיל את כל קבצי ה-object ולבצע linking אליו. לקובץ זה קוראים ספרייה - library.

ה-linking לספרייה יכול להתבצע לאחר הקומפילציה באופן דומה ל-linking לקובצי object או בזמן הריצה. כאשר ה-linking מבוצע לאחר הקומפילציה, הספרייה נקראת ספרייה סטטית (static library) וכאשר ה-linking מבוצע בזמן ריצה הספרייה נקראת ספרייה שיתופית (shared library) או ספרייה מקושרת דינאמית (dynamic linked library או DLL).

לספרייה סטטית יש חיסרון שכל אפליקציה שמשתמשת בה, צריכה להחזיק עותק נפרד של הקוד שלה. הדבר גורם לבזבוז מקום במערכת הקבצים וביזרון המחשב. קבצי הריצה של האפליקציות המשתמשות מכילים כולם העתקים של הספרייה ולכן הם גדולים יותר מהנחוץ. כאשר מריצים מספר אפליקציות כאלה במקביל, נטענים לזיכרון המחשב העתקים של קוד הספרייה. בספרייה שיתופית מוחזק עותק אחד של הקוד במערכת הקבצים. כאשר אפליקציה משתמשת רצה, נטענת הספרייה לזיכרון. כאשר אפליקציה נוספת רוצה להשתמש בספרייה, היא משתמשת באותו עותק המצוי כבר בזיכרון. מכאן ברור השם "shared library" והחיסכון הנובע מכך. ה-linking לספרייה שיתופית יכל להתבצע בזמן טעינת האפליקציה לזיכרון או מאוחר יותר, בזמן ריצת האפליקציה.

מכיוון שקובץ הריצה של אפליקציה המשתמשת בספרייה שיתופית אינו מכיל את כל הקוד הדרוש להפעלתו, נוצרת תלות בגורמים חיצוניים. אם מוחקים או משנים את הקובץ של הספרייה השיתופית, האפליקציה איננה יכולה לתפקד. התלות הזאת יכולה להקשות על ניהול מערכת (המונח "dll-hell" מביע את התסכול יכל להיגרם מסוג כזה של בעיות)

יצירת ספרייה סטטית ב - C

כאמור, ספרייה היא קובץ המורכב מאוסף של קבצי object. הכלי שבו משתמשים ליצירת קובץ ארכיון המורכב מקבצי object שונים נקרא 'ar' עבור archiver והוא יכול להיות מופעל באופן הבא:

```
$ ar rc libutil.a file1.o file2.o file3.o
```

הקבצים file1.o, file2.o ו- file3.o הם קבצי object רגילים שאינם כוללים main. הקובץ libutil.a הוא קובץ הארכיון שיווצר כתוצאה מהפקודה. שם קובץ ספרייה מתחיל תמיד ב- 'lib' ולאחר מכן השם שבחרנו לספרייה, במקרה זה, util. הטיפוס של קובץ ספרייה סטטית הוא a. המסמל את העובדה שמדובר ב- archive. הדגלים r ו c שנתנו ל ar הם עבור create - צור ארכיון חדש אם הוא עדיין אינו קיים, ו- replace החלף קבצי object קיימים בקובץ ארכיון קיים בקבצים חדשים בעלי אותו שם. קובץ ספרייה עלול להיות גדול מאד ולהכיל פונקציות רבות, בשביל ליעל את תהליך ה- linking של אפליקציה עם הספרייה משתמשים במנגנון של indexing. הפקודה הבאה מכניסה טבלת index לתוך קובץ הספרייה:

```
$ ranlib libutil.a
```

index זה ישמש את ה- linker שיקשר את הספרייה לאפליקציות.

שימוש בספרייה סטטית

בכדי להשתמש בספרייה יש לבצע linking בין האפליקציה המשתמשת וביינה. נניח שכתבנו קובץ בשם main.c שנראה כך:

```
#include <util.h>
```

```
int main() {
```

```
    foo(); // foo is a function of the 'util' library
```

```
...
```

לפקודת הקומפילציה שלו יש להוסיף את המסלול במערכת הקבצים בו נמצא ה- header (או ה- header-ים) של הספרייה. הדגל I מאפשר הוספה של מסלולים חדשים ל- include:

```
$ gcc -Wall -c -I /usr/include/myCode/ main.c
```

בהנחה שהקובץ util.h נמצא ב- /usr/include/myCode/ ייווצר הקובץ main.o.

לפקודת ה- linking יש להוסיף את המסלול בו נמצאת הספרייה ואת הבקשה להתקשר אליה:

```
$ gcc main.o -L /usr/lib/myLib/ -lutils -o app
```

הדגל L מאפשר להוסיף מסלול נוסף לספרייה. הדגל l- מבטא בקשה לקישור לספרייה ספציפית. שימו לב שבזמן בקשת הקישור, משתמשים בשם הספרייה util ולא בשם הקובץ libutil.a.

יצירת ספרייה שיתופית

לעובדה שקיים רק עותק אחד של ספרייה שיתופית בזיכרון יש השלכות לגבי האופן שבו הקוד

שלה צריך להיות מקומפל. קוד של ספרייה שיתופית לא יכל להניח הנחות לגבי מיקומו

בזיכרון, הוא אמור לפעול כאשר הוא ממוקם בכל אזור בזיכרון. לקוד בעל תכונה כזו קוראים

position independent code או PIC. בכדי לייצר קוד כזה מוספים לקומפיילר את הדגל -fPIC:

```
gcc -Wall -fPIC -c file1.c
```

```
gcc -Wall -fPIC -c file2.c
```

```
gcc -Wall -fPIC -c file3.c
```

בניגוד לספרייה סטטית, ספרייה דינמית איננה קובץ ארכיון, האופן בו היא בנויה תלוי

בארכיטקטורה הספציפית עליה עובדים ולכן מטלת הרכבת הספרייה מוטלת על ה - linker:

```
$ gcc -shared file1.o file2.o file3.o -o libutil.so
```

הדגל -shared מסמן ל - linker שעליו ליצור ספרייה שיתופית. שם קובץ הספרייה מתחיל ב - lib

כמו ספרייה סטטית אך טיפוס הקובץ הוא so עבור shared object.

שימוש בספרייה שיתופית

נשתמש באותו קובץ main.c שייצרנו כאשר דיברנו על שימוש בספרייה סטטית. ונקמפל אותו

באופן הבא:

```
$ gcc -Wall -c -I /usr/include/sharedLibInclude/ main.c
```

כפי שניתן לראות, זו אותה פקודת קומפילציה בה השתמשנו קודם, רק המסלול ל - include

השתנה בכדי שימצא ה - header המתאים. אין צורך לייצר קוד PIC עבור האפליקציה עצמה

מכיוון שהיא איננה קוד שיתופי.

כעת ניתן לקשר את האפליקציה לספרייה השיתופית:

```
$ gcc main.o -L /usr/lib/mySharedLib/ -lutil -o app
```

פקודה זו עלולה לבלבל: אמרנו קודם שה - linking לספרייה שיתופית מתבצע בזמן ריצה. זאת

הייתה המהות של הספרייה ומה שאפשר את שיתוף והחיסכון. מדוע אם כן אנו מבצעים את ה -

linking בשלב הקומפילציה ?

התשובה היא שלא באמת ביצענו כאן linking. למעשה רק אפשרנו ל - linker לבדוק שקיימת

התאמה בין האפליקציה והספרייה מבחינת שמות פונקציות ומשתנים. ה - linking עצמו יתבצע

בזמן הריצה, כאשר app תטען לזיכרון. בכדי שהמנגנון הטוען את התוכנית לזיכרון ידע למצוא

את הספרייה השיתופית שייצרנו יש להוסיף את המסלול אליה למשתנה סביבה מיוחד המכיל

מסלולים לספריות. משתנה זה ניקרא: LD_LIBRARY_PATH. בכדי לבדוק אם המשתנה כבר

מוגדר, ניתן לכתוב את פקודת ה - shell הבאה:

```
$ echo $LD_LIBRARY_PATH
```

(כזכור, ה - \$ הראשון מסמל כאן שמדובר בפקודת shell)

במידה והמשתנה מוגדר יודפסו אוסף של מסלולים שבהם יחפש טוען התוכניות (ה -

loader) ספריות ל - linking. אם המשתנה לא מוגדר תודפס שורה ריקה או הודעה שהמשתנה אינו מוגדר.

הפקודה המאפשרת להוסיף למשתנה מסלול חדש תלויה בסוג ה - shell בו משתמשים. ב - bash shell ודומיו, הפקודה הנחוצה היא:

```
$ LD_LIBRARY_PATH=/usr/lib/myLib/${LD_LIBRARY_PATH}
```

```
$ export LD_LIBRARY_PATH
```

כאשר '/usr/lib/myLib/' מסמל כאן מסלול אבסולוטי לתיקייה בה נמצאת הספרייה. הסימט 'LD_LIBRARY_PATH': אחראית על שימור כל המסלולים שהיו מוגדרים במשתנה קודם.

אם המשתנה לא היה מוגדר לפני, ניתן להשמיט חלק זה.

ב - c-shell ודומיו, הפקודה המקבילה היא:

```
$ setenv LD_LIBRARY_PATH /usr/lib/myLib/${LD_LIBRARY_PATH}
```

כעת ניתן להריץ את app ובזמן טעינתו לזיכרון, ימצא ה - loader את הספרייה util ויקשר אותה.

linking לספרייה שיתופית במהלך ריצת האפליקציה

בדוגמה האחרונה כתבנו אפליקציה המבצעת linking לספרייה שיתופית בזמן טעינת התוכנית לזיכרון. קיימת אפשרות לקישור גמיש יותר: linking תוך כדי ריצת האפליקציה. קישור כזה ניקרא לעיתים קישור דינמי. קישור דינמי מאפשר טעינת קוד לזיכרון רק כאשר הוא באמת נחוץ. הקישור הדינמי מאפשר גם מודולריות של קוד מקומפל. ניתן ליצור אפליקציה הניתנת להרצה ולהוסיף לה פונקציונליות ללא שינויה. באופן כזה אפשר ליצור מנגנון של plug-in כפי שקיים בדפדפנים שונים.

בשביל לממש קישור דינמי, יש להשתמש בספרייה שנקראת dl'. dl' מספקת שירותים של חיבור לספרייה שיתופית תוך כדי ריצה, הפעלת פונקציות שלה, התייחסות למשתנים גלובאליים, התייחסות לטיפוסים והתנקות מהספרייה. בכדי להשתמש ב - dl יש לכלול לבצע את ה - include הבא:

```
#include <dlfcn.h>
```

בכדי להתייחס לפונקציה, יש להפעיל את הפונקציה dlopen השייכת ל - dl ולספק לה את המסלול המלא לספרייה השיתופית:

```
void* libHandle =dlopen("/usr/lib/myLib/libutil.so", RTLD_LAZY);
```

הפרמטר הנוסף ש - dlopen מקבלת קובע אם בדיקת התאמת סמלי הספרייה תבצע מיידית -

RTLD_NOW או כאשר הקוד מופעל - RTLD_LAZY כפי שבחרנו כאן.

הפעלת dlopen גורמת לקישור הספרייה השיתופית והערך המוחזר ממנה הוא מצביע מסוג *void המתייחס לספרייה שקושרה. במקרה שהקישור לא הצליח, יוחזר NULL והפונקציה dlerror תאפשר קבלת מחרוזת המתארת את הבעיה:

```
if (!libHandle) {  
    fprintf(stderr, "Error during dlopen(): %s\n", dlerror());  
    exit(1);  
}
```

בכדי להפעיל פונקציה של הספרייה השיתופית, יש להגדיר מצביע לפונקציה מטיפוס מתאים לזה של הפונקציה אותה מבקשים להפעיל. נניח שהפונקציה שאנו מעוניינים להפעיל היא פונקציה המציירת מעגל. הפונקציה מקבלת מיקום של מרכז המעגל, את הרדיוס שלו ואינה מחזירה ערך:

```
void (*circle)(int x, int y, int r);
```

כעת יש לקשר בין המצביע לפונקציה שהגדרנו, לפונקציה המתאימה בספרייה השיתופית. נניח שלפונקציה המתאימה קוראים drawCircle שטיפוס שלה זהה לזה של circle. הפונקציה שמבצעת את הקישור הדרוש נקראת dlsym:

```
circle = dlsym(libHandle, "drawCircle");
```

dlsym מקבלת את ההתייחסות לספרייה השיתופית ואת שם הפונקציה אותה רוצים לקשר. Dlsym מחזירה מצביע לקוד הדרוש. ניתן לבדוק אם הקישור לא הצליח ולהדפיס את הסיבה, באופן הבא:

```
const char* errorMsg = dlerror();  
if (errorMsg) {  
    fprintf(stderr, "Error locating 'drawCircle' - %s\n", error_msg);  
    exit(1);  
}
```

כעת, כאשר אנו יודעים שהקישור הצליח, ניתן להפעיל את הפונקציה באופן הרגיל:

```
circle(10,10,7);
```

עם סיום השימוש בספרייה השיתופית כדאי להתנתק ממנה ולשחרר את הזיכרון שהיא תופסת:

```
dlclose(libHandle);
```

אם אפליקציה אחרת משתמשת באותו הזמן בספרייה השיתופית, מובן שהמקום שהיא דורשת בזיכרון לא ישוחרר.

מכיוון שטעינת ספרייה לזיכרון דורשת זמן, לא כדאי להתנתק מספרייה אם קיימת אפשרות לשימוש בה בעתיד הקרוב.