תרגול מס' 4: דוגמאות בסיסיות ב-C++.

1 מחלקות ועצמים

1.1 C++כשפת תכנות מונחה עצמים

שפת C++ היא שפת תכנות המספקת תמיכה מעולה בתכנות מונחה עצמים (Object Oriented Programming – OOP).

תכן מונחה עצמים Object Oriented Design (OOD) מורכב משלושה שלבים:

1. **מזהים את העצמים** (אובייקטים) במערכת אותה מנסים לבנות, ונותנים להם שמות.
2. לכל עצם,**מגדירים את המאפיינים שלו** (attributes/characteristics) -מה הוא יודע, ומה מבדיל בינו לבין עצם דומה (מאותה מחלקה.)
3. לכל עצם, **מגדירים את הפעולות (מתודות) שלו** – איך ידבר איתו העולם החיצון ? (אך ורק דרך המתודות הנ"ל.)

1.2 הצהרת מחלקה

נניח, כי זהינו במערכת עצמים שהם תאריכים, שלושת השלבים לעיל יתנו לנו את התיאור הבא:

שם האובייקט: תאריך

מאפיינים של העצם (מה הוא יודע:)

- היום שלו
- החודש שלו
- השנה שלו

המתודות של העצם (מה הוא יכול לעשות:)

- לאתחל את עצמו לתאריך מסוים
- להגיד מהו היום שלו
- להגיד מהו החודש שלו
- להגיד מהי השנה שלו
- להתקדם לתאריך הבא אחריו

נתרגם את התיאור הנ"ל להצהרה של מחלקה (class) ב-C++ בקובץ (Date.H
שימו לב כי ב Unix-קבצים שכתובים ב C++ חייבים להסתיים ב H ו- C
אותיות גדולות:)

```
// A simple date type (Date.H)
#ifndef DATE_H
#define DATE_H

class Date {                    /* שם המחלקה */
public:                         /*אתחול המחלקה לתאריך נתון */
    // Constructor: initializes date
    // to a (valid) date day/month/year
    Date(int day, int month, int year);

                                /* הצהרה על המתודות של המחלקה */
    // Methods to access data individually
    int theDay() const;    // Day of this
date
    int theMonth() const; // Month of this
date
    int theYear() const;   // Year of this
date

    // A method to change the date
    void advance(); // Advance to the next
date

private:
    // Data                     /* המאפיינים של המחלקה */
    int day_;      // Date's day
    int month_;    // Date's month
    int year_;     // Date's year
};

#endif     // DATE_H
```

Constructor – ככה ניצור ונאתחל תאריך

להגיד מה היום
להגיד מה החודש
להגיד מה השנה

Const בסוף הפונקציה אומר שהמתודה לא משנה את האובייקט

לקדם תאריך

_ כי זה משתנה בprivate

נבחין כי:

- הצהרת המחלקה מכילה אך ורק את המנשקים של המתודות ולא את המימושים.
- קיימות מתודות אשר לא משנות את המצב של האובייקט ופשוט מחזירות ערכי המשתנים כתשובה לשאילתות, מתודות כאלה נקראות "מתודות גישה" והגדרתן מסתיימת ב -const, למשל:

```
int theDay() const;
```

- המתודה Date היא מתודה מיוחדת, היות ושמה כשם המחלקה עצמה והיא לא מחזירה כלום. מתודה כזאת נקראת constructor ועל תפקידה נעמוד בהמשך.
- ההצהרה של מחלקה ב- C++ מחייבת להצהיר על המשתנים והפונקציות הפרטיים של המחלקה, דבר העלול לחשוף את פרטי המימוש, והנוגד את מה שנלמד בחומר על – ADT בשיעורים הבאים נלמד איך להתגבר על "חסרון" זה.
- כדי להפעיל מתודה יש צורך בעצם מהטיפוס המתאים.
- לכל מתודה יש פרמטר נסתר בשם this והוא מצביע לעצם עליו המתודה הופעלה. עבור עצם מטיפוס X אז this הינו מצביע מטיפוס X* const, כלומר הכתובת של this קבועה. אם המתודה מוגדרת כפועלת על עצם קבוע (const בסוף הפונקציה) אז this יהיה מצביע מטיפוס const X* const.
- ב class ברירת המחדל עבור בקרת הגישה היא private.

1.3 מימוש המחלקה

המימוש של המחלקה Date מוגדר בקובץ .Date.C והוא מכיל מימוש של כל המתודות החברות : (member methods)

מימוש: theDay

```
int Date :: theDay() const {
        return day_;
}
```

המימוש של theDay גם כן מציין שהמתודה לא משנה את האובייקט.

יש להגדיר את השייכות של המימוש למחלקה Date, ע"י שימוש באופרטור :: הנקרא **scope resolution operator**.

המימושים של מתודות theMonth ו theYear-דומים ל.theDay-

מימוש:advance

מתודה advance משנה את התאריך לתאריך של היום הבא:

```
void Date :: advance() {
	day_ ++;		// advance the day
	switch (month_) {
		case 1: case 3: case 5:
		case 7: case 8: case 10:
		// January, March, May, July, August, October
			if (day_ == 32 ) {
				day_ = 1;
				month_ ++;
			} break;
		case 12:		// December
			if (day_ == 32) {
				day_ = 1;
				month_ = 1;
				year_ ++;
			} break;
		case 4: case 6: case 9: case 11:
		// April, June, September, November
			if (day_ == 31) {
				day_ = 1;
				month_ ++;
			} break;
		case 2:		// February
			if (day_>29 || day_==29 && (year_%4>0
				|| year_%100==0) && year_%400>0) {
				day_ = 1;
				month_ ++;
			}
	}
}
```

1.4 constructors

המתודה הראשונה המוגדרת במחלקה Date, ששמה כמו שם המחלקה - Date() היא מיוחדת, ונקראת constructor של המחלקה. ה constructor-נקרא אוטומטית כאשר נוצר אובייקט חדש מסוג Date.

מטרת ה constructor: לאתחל את האובייקט.

מאפיינים של constructor:

1. אין לו טיפוס חזרה (אין אפילו void!)
2. השם שלו זהה לשם המחלקה (כולל case)
3. לבנאי שאינו מקבל פרמטרים קוראים גם default constructor
4. אם לא מוגדר בנאי בצורה מפורשת למחלקה, הקומפיילר ייצור בנאי חסר פרמטרים בעצמו. הבנאי הנוצר על ידי הקומפיילר קורא לבנאי חסר הפרמטרים של כל אחד מהשדות שלו.
5. אם מוגדר בנאי כלשהו הקומפיילר לא ייצור בנאי.
6. באתחול של מערך נקרא בנאי חסר פרמטרים לכל אחד מהאיברים(לכן לא יהיה ניתן ליצור מערך של עצמים שאין להם בנאי מתאים)

המימוש של ה constructor-דומה למימוש של מתודה רגילה:

```
Date :: Date(int day, int month, int year) {
   day_ = day;
   month_ = month;
   year_ = year;
}
```

ניתן לראות שאין טיפוס חזרה, וכי השם זהה לשם המחלקה

במצב זה קודם מבוצע אתחול עם בנאי חסר פרמטרים לכל אחד מהשדות של ה classורק לאחר מכן מבוצעת השמה בתוך ה{}

תזכרות:
אתחול מבוצע על ידי בנאי למשל:

```
int n(5);      אתחול
int n = 5;     אתחול
```

השמה מתבצעת על ידי אופרטור:=

```
n = 5;         השמה
```

צורה עדיפה להציב ערכים התחלתיים למשתני המצב ב constructor-היא להשתמש ב-**רשימת אתחול** (initialization list), למשל:

```
Date :: Date(int day, int month, int year) :
   day_(day), month_(month), year_(year) { }
```

כאן רשימת האתחול עושה את כל העבודה עבור ה constructor, ולכן הגוף ריק.
אם יש שדות שמוגדרים כ const אז חייב לקבוע את ערכם ברשימת האתחול של הבנאי.

הערה: לא חייבים לספק constructor לכל מחלקה .

1.5 כללי גישה

במחלקה של תאריכים מוגדרים משתנים day_, month_, year_ שהם private ומתודות Date(), theDay(), theMonth(), theYear() ו advance() שהן public.

תוויות private ו public- מגדירות כללי גישה (access control) למשתנה או למתודה. אם חבר (member) במחלקה הוא private, רק קוד ששייך לאותה מחלקה (קוד במתודות שלה) יכול לגשת לאותו חבר. לעומת זאת, כל קוד שרוצה יכול לגשת לחבר שהוא public.

החלק הציבורי (public) של הגדרת המחלקה מגדיר את המנשק החיצוני של המחלקה, כלומר החלק שאומר איך ניתן להשתמש במחלקה ובאובייקטים שלה. למשל, החלק הזה מגדיר מהן ההודעות שניתן לשלוח לאובייקט מאותו סוג.

משתנים במחלקה הם כמעט תמיד **לא**! public
כדי להדגיש זאת, משתמשים ב- _ אחרי השם, למשל day_.

1.6 דוגמה לשימוש במחלקה Date

```
/* main1.C */
#include "Date.H"   // capital H

void main() {
        // defining a Data Object
        Date   HerBirthDay(19,8,1976);

        printf("Her birthday is on %d / %d / %d \n",
                HerBirthDay.theDay( ) , HerBirthDay.theMonth( ),
                HerBirthDay.theYear( ) );

        HerBirthDay.advance( );

        printf("On %d / %d / %d she'll be very happy\n",
                HerBirthDay.theDay( ) , HerBirthDay.theMonth( ),
                HerBirthDay.theYear( ) );

        // HerBirthDay.day_ ++ ;   // illegal: day_ is private
        // HerBirthDay.month_=8;// illegal: month_ is private
}
```

לא ניתן לגשת ישירות לחבר מחלקה שהוא private

הרצת התוכנית תוציא את הפלט הבא:

```
Her birthday is on 19/8/1976
On 20/8/1976 she'll be very happy
```

1.7 העמסה של פונקציות(overloading)

מושג	מימוש	הטיפוס שקובע איזו פונקציה תתבצע	תכונות
Overload (העמסה)	1. אותו שם לכל המתודות. 2. חתימה שונה. 3. כל המתודות מוגדרות באותה מחלקה[1].	טיפוס סטטי (לפי טיפוס המצביע שהוגדר)	

ב ++C-פונקציה מזוהה על ידי שמה וגם על ידי מספר וסוג הפרמטרים שלה.
(בניגוד לשפת c בה פונקציה מזוהה על ידי שמה בלבד).
ניתן ב ++Cלכתוב מספר פונקציות בעלות אותו שם, ו"להעמיס" על אותו שם, פונקציות שונות עבור טיפוסים שונים.
הקומפיילר יבחר את הפונקציה המתאימה לפי מספר וסוגי הפרמטרים.
בחירת הפונקציה אינה תלוייה בערך ההחזרה, כדי לאפשר קריאות של הקוד.

כדי לייצר אובייקט חדש מסוג Date, חייבים למלא את כל הפרמטרים ב-constructor, ז"א, לא ניתן לבנות תאריך חדש בלי להגיד במפורש מה הוא היום, החודש והשנה של התאריך, כלומר לבנות תאריך בלי אתחול. זה שונה מאוד ממשתנים פרימיטיביים, למשל מסוג int, אותם ניתן להגדיר בלי אתחול.

```
Date yesterday;      // illegal! must be
                     // initialized with 3 ints
Date arrayOfdates[100]; // illegal! All 100 Dates
                        // must be initialized with 3 ints
```

כיצד ניתן לפתור את הבעיה? בנוסף ל constructor-הנוכחי, דרוש **default constructor**, כזה שלא דורש פרמטרים. נוסיף אותו להגדרת המחלקה:

```
class Date {
public:
    Date(int day, int month, int year);
    Date();   // Default constructor
    …
};
```

ניתן לממש default constructor כפונקציה ריקה בלי אתחולים. או ניתן להוסיף קוד של אתחול. במקרה של Date חייבים לאתחל משתני מצב לערך תקין כלשהו, למשל 31 לדצמבר, 2000:

```
// Original constructor
```

```
Date :: Date(int day, int month, int year) :
        day_(day), month_(month), year_(year) { }

// Default constructor
Date :: Date() :
        day_(31), month_(12), year_(2000) { }
```

איך מהדר יבדיל בין שני ה-constructor-ים ? הוא בודק לפי מספר וסוגי הפרמטרים : אם אובייקט מוגדר עם 3 פרמטרי int ה constructor-המקורי נקרא ; אם אובייקט מוגדר בלי פרמטרים, ה default constructor-נקרא.

```
Date hisBirthDate(2,11,1997); // original constr.
Date someDay; // default - someDay is December 31, 2000
Date goodDays[100]; // default - 100 Dates representing
                    // December 31, 2000
```

שימו לב : אסור לשים סוגריים ריקות אחרי שם האובייקט, למשל:

```
Date someday();          // Wrong!!!
```

מהדר חושב שזוהי הכרזת פונקציה ששמה someday, שמחזירה Date ולא מקבלת פרמטרים.

ניתן לתת אותם שמות גם לפונקציות ומתודות אחרות, לא רק ל.constructor-

העמסה של שמות הפונקציות (function overloading) פירושה הגדרת מספר פונקציות או מתודות עם אותו שם, אך עם מספר ו/או טיפוסי פרמטרים שונים,

למשל : ניתן להעמיס את הפונקציה date::advance כך שתתקדם במספר נתון של ימים ולאו דווקא ביום בודד:

```
class Date {
public:
        void advance();
        void advance(unsigned int k);   // advance in k days
        …
};
```

ניתן לממש את המתודה החדשה ע"י k קריאות למתודה המקורית:

```
void Date::advance(unsigned int k) {
        int j;
```

```
    for(j=1;j<=k;j++)
        advance();
}
```

דוגמא:

```
Date   HerBirthDay(19,8,1976);
HerBirthDay.advance(5);      // now it's 24/8/1976
HerBirthDay.advance();       // advance to become 25/8/1976
```

ניתן להעמיס פונקציות רגילות, ולא רק מתודות של מחלקות דוגמא:

```
void swap(int * , int * );
void swap(double* , double* );
```

המהדר תמיד יודע האם להשתמש בגרסה הראשונה או השנייה לפי **התאמת הארגומנטים** (argument matching), תהליך שמשווה פרמטרים אקטואליים ופורמליים בכל קריאה לפונקציה, למשל:

```
int i = 27;
int j = 35;
double z = .98;
double w = 12.1;
swap(&i,&j);     // swap(int*,int*) is called
swap(&z,&w);     // swap(double*,double*) is called
```

1.8 ארגומנטי ברירת מחדל(default arguments)

ארגומנטי ברירת מחדל הם מקרה פרטי של העמסת פונקציות.
ניתן לתת ערכי ברירת מחדל אך ורק **בסוף** רשימת הארגומנטים.
נותנים ארגומנטי ברירת מחדל פעם אחת בלבד בהצהרה ולא במימוש.

ניתן להגדיר רק constructor אחד, אשר יממש 4 הסוגים הבאים של הגדרות (כוללים 2 הסוגים שלמדנו:)

```
Date hisBirthDate(2,11,1997); // Nov. 2, 1997
Date someDay;        // December 31, 2000
```

```
Date anotherDay(3,8); // August 3, 2000
Date yetAnother(15);  // December 15, 2000
```

ב ++C-ניתן להגדיר ארגומנטי ברירת מחדל שנותנים בצורה אוטומטית ערכים עבור פרמטרים חסרים בקריאת לפונקציות (לאו דווקא ל.(constructor-

הצהרת ה constructor-בדוגמא:

```
class Date {
public:
        Date(int day=31, int month=12, int year=2000);
        …
};
```

המימוש זהה למימוש ה constructor-המקורי: נותנים ארגומנטי ברירת מחדל רק בהצהרה ולא במימוש .

שימו לב שההצהרה הזאת מכילה גם את ה default constructor-זה שאין לו פרמטרים). לכן אסור להגדיר default constructor נוסף!

ניתן לתת ערכי ברירת מחדל אך ורק **בסוף** רשימת הארגומנטים, לכן הדוגמא הבאה **לא חוקית**:

```
Date(int day=1, int month=12, int year);
        // illegal! year is not a default argument
```

ניתן להגדיר ארגומנטי ברירת מחדל גם בפונקציות ומתודות רגילות.

1.9 המלה השמורהconst

1.9.1 הגדרת מתודה של עצם

השימוש הראשון שראינו במילה היה, להגדרת מתודה של עצם, שלא אמורה לשנות את המצב של העצם (את המשתנים שלו), דוגמא:

```
class Date {
public:
        …
        int    theMonth( )   const;
        void advance( );        //changes internal fields - not const!
 };
```

במקרה הזה יש לרשום את ה const גם במימוש של המתודה:

```
int    Date::theMonth( ) const {
        // month_++;    is illegal - the method is const
        return month_;
}
```

1.9.2 הגדרת משתנים קבועים

בהכרזה על משתנה ניתן להוסיף לשם הטיפוס את המילה השמורה.const
ערכים מטיפוס X יומרו אוטומטית לטיפוס const X כאשר צריך, נסיון
להשתמש ב const X-עבור מקום בו דרוש הטיפוס X יגרום לשגיאת קומפילציה.

ניתן להגדיר ב- C++ משתנים קבועים, שלא ניתן לשנותם כל עוד הם "חיים" במהלך התוכנית. במקרה כזה יש לספק להם את הערך שלהם יחד עם ההגדרה.
דוגמאות:

א. משתנה גלובלי:

```
const double pi = 3.141592;

double rad2deg( double rad ) {
        // pi = 22/7 - illegal , pi is declared const;
        return ( rad/pi*180 );
}
```

אם יש צורך בלהצהיר על המשתנה הזה, יש לרשום את ה const גם כן:

```
extern const double pi;          // declaring const
```

ב. משתנה לוקלי:

```
int f( int k ) {
        const int j = k*k;
        const int days_in_week = 7;
        int    m;

        m = j * days_in_week; // ok
        // days_in_week = 7;  -- illegal , it's const
        return m/k;
}
```

1.9.3 עצמים קבועים

באופן זהה, ניתן להגדיר גם עצמים קבועים: עצם קבוע ניתן לאתחל באמצעות הקונסטרקטור, אך לא ניתן לערוך בו שינויים מרגע זה והלאה.

לגבי עצמים כאלו, מותר להשתמש בפונקציות שאינן משנות אותן בלבד (אלו שמוגדרות כ- const) דוגמא.

```
const Date newYear99(1,1,1999);
Date  someday;

newYear99 = someday;    // illegal. newYear is const.
newYear99.advance();    // illegal. advance is non-const func.
newYear99.theDay();     // ok. theDay is a const func.

someday = Date(21,4,98);      // ok
someday.advance();            // ok
```

אפשר לבצע השמה של משתנה קבוע לתוך משתנה לא קבוע כי רק מבצעים אופרטור השמה בכל אחד מהשדות !! כלומר, newYear99 עדיין נשאר קבוע.

```
someday = newYear99;          // ok. A non-const copy is created
```

1.9.4 מצביעים לקבועים

ניתן גם להגדיר מצביע לקבוע: לא ניתן לשנות את הערך אליו הוא מצביע.
עם זאת ניתן לשנות את הכתובת אותה מחזיק מצביע לקבוע.
כלומר הערך הנשמר במשתנה המוצבע קבוע.
int const* -ו const int* המשמעות היא ש ptrמצביע על int שהוא מתחייב לא לשנות את ערכו. אבל כמצביע הוא איננו קבוע והוא יכול לשנות את הכתובת עליה הוא מצביע כרצונו.

דוגמא:

```
const Date constDate(1,1,1999);
Date  normDate;
const Date* constPointer;
Date* normPointer;

normPointer = &normDate;  //ok
constPointer = &normDate; //ok
normPointer->advance();    //ok
//constPointer->advance();   //illegal – even if we're actually
                             //not pointing at const
constPointer->theDay();    //ok

//normPointer = &constDate; //illegal!

constPointer = &constDate; //ok
constPointer->theDay();    //ok
```

לא חייב לאתחל אתconstPointer כי המצביע עצמו הוא לא const אלא הערך שהוא מצביע אליו

ערכים מטיפוס X יומרו אוטומטית לטיפוס const X כאשר צריך, אבל הכיוון ההפוך אינו תקין. ניתן שפוינטר מטיפוס const Date יצביע על משתנה מטיפוס Date.

הפוינטר מתחייב לא לשנות את הערך שהוא מצביע עליו

ערכים מטיפוס X יומרו אוטומטית לטיפוס const X כאשר צריך, אבל הכיוון ההפוך אינו תקין. זוהי דוגמה לכיוון ההפוך, לא ניתן שפוינטר לDate יצביע על const Date כי אז נוכל לשנות עצם קבוע

ניתן לשנות את הכתובת עליה מצביע המצביע

הערה למתקדמים: ניתן גם להגדיר את הכתובת שמחזיק מצביע לקבועה (מבלבל מאוד, ובדרך-כלל לא נחוץ): (יש טעות בקובץ המקורי)

```
Date* const cpoint = new Date(1,1,1999); //the address is const
const Date* const ccpoint = new Date(1,1,1998);
                          //address and value are const
cpoint->advance();          //ok
ccpoint->advance(); //illegal
```

במקרה זה חייב לאתחל את המצביעים כי הכתובת שהם מצביעים אליה קבועה

הכתובת הנשארת במצביע קבועה – כלומר לא ניתן לשנות הכתובת עליה הוא מצביע

```
cpoint = &constDate;          //illegal.
cpoint = new Date(2,1,1999);  //illegal ccpoint also illegal
```

1.9.5 שימוש בערכים שפונקציה מקבלת או מחזירה

את הגדרות ה- const השונות ניתן לערוך גם על משתנים שפונקציה מחזירה או מקבלת. כזכור, פונקציה עובדת על עותקים שונים של הערכים שהיא מקבלת או מחזירה. מכאן, שברוב המקרים אין משמעות להגדרת משתנים אלו כקבועים.

עם זאת, יש משמעות חשובה מאוד לשימוש במצביע לקבוע.

א. פרמטר של הפונקציה

```
void printDate(Date d);
void printDateP(Date* d);
void printDatePConst(const Date* d);

printDate(normDate);   //ok
```

אפשר לבצע השמה של משתנה לא קבוע לתוך משתנה קבוע כי מבוצעת ההעתקה!! אז מה שמועבר לפונקציה זה בעצם העתק, ולא המשתנה עצמו. (בשונה מאשר מצביעים או רפרנסים)

```
printDate(constDate);   //On good compilers - ALSO OK!!!!
printDateP(&normDate); //ok
//printDateP(&constDate); //Illegal!
printDatePConst(&normDate);  //ok. Remember - const pointer
                             // can point to non-const
printDatePConst(&constDate); //ok
```

ב. ערך מוחזר מהפונקציה

שימוש נפוץ להחזרת מצביע לקבוע – הוא כאשר מעוניינים להחזיר מצביע לשדה פנימי של האובייקט. כך מובטח לנו שמי שקיבל את המצביע לא יוכל להשתמש בו ע"מ לשנות את תוכן המחלקה "מבחוץ"

```
const int * Date::getYearPointer() {
        return &_year; //the returned pointer is const,
                       // though the original field wasn't
}
```

פונקציה זו מחזירה מצביע מטיפוס const int ולכן לא ניתן לבצע את האתחול הזה כי אז היה אפשר לשנות את ערך _year דרך מצביע זה

```
//int * p = normDate.getYearPointer();
        //illegal – function returns const pointer!
```

```
const int * pc = normDate.getYearPointer(); //ok
```

פוינטר לקבוע מתחייב לא לשנות את הערך שהוא מצביע עליו

```
//*pc=3 //illegal! pc is a const pointer!
```

1.9.6 משתני אובייקט קבועים

ניתן להגדיר ב- C++ משתני אובייקט קבועים. המשתנים האלה הם חברי מחלקה שניתנים לאתחול אך ורק ברשימת האתחול של הקונסטרקטור.

דוגמא:

```
class Person{
 public:
      Person(int id, const char* name);
      void changeName(const char* newname);
 private:
      const int id_;          //will never change once created
      char* name_;
}
```

האתחול של משתנה קבוע מבוצע ברשימת האתחול

```
Person::Person(int id, const char* name):
      id_(id) , name_(createNewCopy(name)) {     //ok
//    id_ = id // illegal
```

לא ניתן לשנות ערך של משתנה קבוע לאחר אתחולו.

```
//    name_ = createNewCopy(name)  //would've been ok
}

void Person::changeName(const char* newname){
      delete name_;
      name_ = createNewCopy(name);
      // id_ = 3 illegal
}
```

הערה: משתני אובייקט קבועים הם אמנם "יפים" עיצובית, אך השימוש בהם נוטה לסבך תוכנות "שלא לצורך", ולכן הוא לא נפוץ.

ללימוד עצמי: קובץ Makefile עבור++C

כאשר אנו יוצרים "פרוייקט ++C "בעזרת קובץ Makefile, יש לשים לב לשינויים הבאים שצריך לעשות:

תוכנית ה make-משתמשת בכלל סטנדרטי חדש לבניית קובץ o.בהנתן קובץ) C.עד עתה אנו למדנו על הכלל בהינתן קובץ :(c.

$(CCC) $(CCFLAGS) -c <file>.C

כדי להגדיר את המשתנים עבור הכלל הסטנדרטי החדש, יש להחליף את שורות הגדרת המשתנים בתחילת ה Makefile-לשורות הבאות:

```
CCC = g++
CXXFLAGS = -Wall -g
CXXLINK = $(CCC)
```

להלן דוגמא לקובץ Makefile עבור החומר הנלמד בתרגול זה:

```
#This is a Makefile for the Date example
CCC = g++
CXXFLAGS = -Wall -g
CXXLINK = $(CCC)
OBJS = Date.o main1.o
RM = rm -f

#Default target (usually "all")
all: example

#Creating the executables
example: $(OBJS)
	$(CXXLINK) -o example $(OBJS)

#Creating object files using default rules
Date.o: Date.C Date.H
main1.o: main1.C Date.H

#Cleaning old files before new make
clean:
	$(RM) example *.o *.bak *~ "#"* core
```

ללימוד עצמי: קלט/פלט ב++C-

בקובץ הגדרות iostream.h מוגדרים זרמים (streams) של קלט cin ושל פלט cout כך שהמתכנתים יכולים לקרוא ערכי קלט לתוכנית ולכתוב ערכי פלט ממנה. כל תוכנית ++C המשתמשת בקלט/פלט חייבת להגדיר שורה:

```
#include <iostream.h>
```

שימו לב שב ++C-משתמשים בזרמים המוגדרים ב iostream.h-במקום פונקציות קלט/פלט של C שלמדנו המוגדרים ב.stdio.h-

קלט

```
int x;
float f;
int y;
cin >> x >> f >> y;
```

הביצוע של הפעולה האחרונה דורשת ממשתמש להקליד 3 מספרים עם רווחים ביניהם. למשל, אם משתמש מקליד:

```
21 13.4     -4
```

נקבל ש21- נשמר ב x-, 13.4 ב f-ו-4- ב.y-

ניתן להשתמש באופרטור הקלט >> לתוך זרם cin עם כל הטיפוסים הסטנדרטיים של ++C-ו: int, char, float. double-מפרידים ביניהם בעזרת הרווחים או ה-tab-ים או המעבר לשורה חדשה.

אך לא תמיד כדאי להשתמש באופרטור הקלט >>. למשל, רוצים לקרוא קלט תו-תו כמו שהוא כולל הרווחים. נעזרים במתודה get של:cin

```
char ch;
cin.get(ch);
```

פלט

```
int x=2;
float f=29.7;
int y=5;
cout << “The answer is “ << x << “.” << endl;
cout << “Next is “ << y;
cout << “ and next is “ << f << “.” << endl;
```

הקבוע endl מסמן את ״סוף שורה״ והוא זהה לתו.“\n”

התוצאה בפלט היא:

```
The answer is 2.
Next is 5 and next is 29.7.
```

ניתן לאחד את כל שלושת פעולות הפלט בפעולה אחת:

```
cout << “The answer is “ << x << “.” << endl
        << “Next is “ << y
        << “ and next is “ << f << “.” << endl;
```

או להפך – להשתמש בסדרת פעולות פלט קצרות, למשל:

```
cout << “The answer “;
cout << “is “ << x;
cout << “.”;
cout << endl;
cout << “Next is “;
cout << y;
cout << “ and next is “;
cout << f;
cout << “.”;
cout << endl;
```