הכרות עם C++

C++ כ- C משופר

תוכנית ראשונה ב C ++

הערות של שורה אחת

```
// First C++ program
#include <iostream>
int main() {
   /* This is not a good program:
    we are not using C++ I/O */
   printf("Hello World\n");
   // but notice the new style for
   // one line remarks:
   return 0; // comment till end of line
}
```

זיכרון דינמי ב C ++

הקצאת אובייקטים בודדים:

new *type* **במקום:**

malloc(*sizeof(type)*)

delete *pointer* **במקום:**

free (*pointer*)

אין צורך ב casting

```
int* pi = new int ;
if (pi == NULL)  ….
*pi = 6;
delete pi   ;
```

הקצאת מערכים:

new *type[# of elements]* להקצאת מערך.

delete[] *pointer to array* לשחרור מערך

```
int* pi = new int[5] ;
if (pi == NULL)  ….
pi[3] = 6;
delete[] pi   ;
```

const in C ++

ב C++ משתמשים ב-const כתחליף ל #define

```
const int max = 100;
const char digits[] = {'0','a','2','c','4','b'};
```

what happens when we try: "max = 5;"?

מה יקרה במקרים הבאים ?

```
digits[3] = 'u' ; // מותר?
int* max2= &max ; // מותר ?
const int* max2 = &max ; // פתרון
int k; max2 = &k; // מותר ?
```

שימוש נוסף הינו פרמטרים וערכי החזרה מפונקציות:

```
int cparr (int* to, int size, const int* from);
const char* getName (struct Person* p) ;
```

מי שיקבל את המחרוזת שהפונקציה החזירה, לא יוכל לשנות אותה

טוב לתיעוד ולקומפיילר

Reference

- reference הוא **שם אחר** למשתנה קיים.
- לדוגמה: הביטוי **int&** משמעותו reference לטיפוס int (לא לבלבל עם &x שמשמעותו "הכתובת של משתנה x").
- reference חייב להיות מאותחל כאשר הוא נוצר כדי לדעת מי המשתנה הקיים שהוא "יתחפש" אליו. **אתחול זה אינו השמה.**
- לאחר האתחול לא ניתן להגיד ל-reference להתחפש למשתנה אחר - הוא "תקוע" עם מי שאתחל אותו.
- כל פעולה המופעלת על ה reference פועלת על המשתנה שהוא מהווה שם נוסף אליו.

```
int i = 8 ;
int& j = i;
int x = i;
i = 5 ; // j = 5 , x = 8
j++;   // i = j = 6
```

Reference II

- ב-reference משתמשים בעיקר להעברת פרמטרים אל ומפונקציות.
- ב- C יכולנו להעביר רק את ערכם של הפרמטרים (call by value). ב- C++ נוכל להעביר את הפרמטרים עצמם. (call by reference).
- העברת פרמטרים גדולים by reference זולה יותר מהעברתם by value כי אין צורך להעתקם.

swap in C

```
void swap(int* p, int* q) {
        int t = *p ;
        *p = *q; *q = t ;
}
```

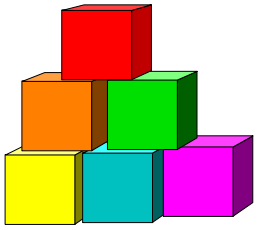
swap in C++

```
void swap(int& p, int& q) {
        int t = p ;
        p = q; q = t ;
}
swap(x,y); // OK
swap(x,5) ; // Error
```

דוגמא נוספת לשימוש ב reference

```
int&  arrayBounds (int* array, int size, int& max, int& min) {
        int i,sum = 0 ;
        max = min = array[0];
        for (i=0 ; i<size; i++) {
                if (max < array[i])
                        max = array[i];
                if (min  > array[i])
                        min = array[i] ;
                sum += array[i] ;
        }
        return array[size/2] ; // what if returns sum ?
}
```

מימוש מבני נתונים מופשטים ב- C++



מבנה הנתונים מחסנית

תזכורת

מחסנית הינה מבנה נתונים התומך בפעולות הבאות:

push - הוסף איבר למחסנית.

pop - הוצא את האיבר ה"צעיר" ביותר במחסנית.

top - החזר את האיבר ה"צעיר" ביותר במחסנית.

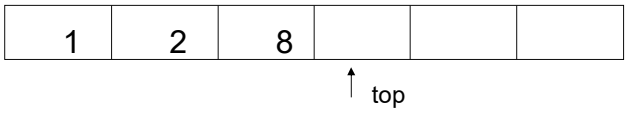
ובנוסף:

create - צור מבנה נתונים מסוג מחסנית (ריק).

destroy - הרוס את מבנה הנתונים מחסנית.

print - הדפס את תוכן מבנה הנתונים.

דוגמא למימוש אפשרי:
מערך + מצביע למקום הפנוי הבא



מימוש ב ++ C לעומת C

ב-C כאשר הגדרנו ADT נעזרנו:

- בקובץ header אשר הכיל את הממשק: הוגדר בו הטיפוס של ה ADT והוגדרו בו פונקציות המטפלות ב ADT
- בקובץ ה source הופיע המימוש של הפונקציות

ב ++C הוטמעה צורת עבודה זו בשפה, וניתן להגדיר טיפוסים (מחלקות) אשר כוללים בתוכם הן את השדות של כל משתנה מאותו טיפוס (אובייקט של אותה מחלקה) והן את הפונקציות (מתודות) המופעלות עליו

מימוש ב-++ C: Header File

```
enum Result {Success, Fail} ;
struct Stack {
        int* array;
        int size, top_index ;

        Result init (int size) ;
        void destroy() ;
        Result push (int e);
        Result pop ();
        Result top(int& e);
        Result print() ;
} ;
```

שדות

data members

פונקציות של המחלקה

method or member functions

מימוש ב ++C

Source File

מסמן שזוהי של method class Stack

שם ה method

ערך מוחזר

```
Result    Stack::init (int s) {
        if ((array = new int[s] )==NULL)
                return Fail;
        size = s; top_index = 0 ;
        return Success ;
}
void    Stack::destroy () {
        delete[] array ;
}
```

גישה ישירה לשדות של ה- struct כאילו הם משתנים לוקליים

מימוש ב C++

Source File

```
Result Stack::push(int e) {
        if (top_index == size)
                return Fail ;
        array[top_index] = e;
        top_index ++  ;
        return Success;
}

Result Stack::pop () {
        if (top_index == 0) return Fail;
        top_index--;
        return  Success;
}
```

מימוש ב C++

Source File

```
Result Stack::top (int& e) {
    if (top_index == 0) return Fail;
    e = array[top_index-1];
    return Success;
}
```

שימוש ב-Stack ב- C

```
#include "Stack.h"
int main() {
    Stack s ;
    int  i;
    init (&s,100) ;
    push (s,1); push(s,213);
    pop (s); top(s,&i);
    destroy (s);
}
```

שימוש ב- Stack ב- C++

```
#include "Stack.h"
int main() {
    Stack s ;
    int  i;
    s.init (100) ;
    s.push (1); s.push(213);
    s.pop (); s.top(i);
    s .destroy ();
}
```

קריאה לפונקציה מתבצעת בדיוק כמו גישה לשדה

המצביע this

- שימו לב לאופן השונה בו נקראו הפונקציות ב- C וב- C++. ב- C נדרשנו להעביר מצביע לאובייקט (מבנה) עליו על הפונקציה לעבוד, בעוד שב- C++ לא העברנו פרמטר כזה, מצד שני קראנו לפונקציה של המחלקה (מתודה) באופן שונה:

```
s.push(3); // calls push on object s with parameter 3
```

- נשאלת השאלה - כיצד הפונקציה יודעת לאיזה אובייקט התכוונו כאשר קראו לה ?
- הפתרון: כאשר נקראת מתודה כלשהי נשלח לה פרמטר סמוי בשם this שטיפוסו הנו מצביע למחלקה בה מוגדרת המתודה, ומכיל מצביע לאובייקט עמו נקראה המתודה. בכל גישה לשדה של האובייקט או למתודה שלו מתוך מתודה אחרת הקומפיילר מוסיף הצבעה דרך this.

```
void Stack::top(int& j) {
    j = array[top_index-1];
}
```

```
void Stack::top(int& j) {
    j = this->array[this->top_index-1];
}
```

המתודה, לאחר שהקומפיילר הוסיף את המצביע this

בעייה: גישה למשתנים

ב ++C העבודה עם מחלקות מהווה את דרך העבודה העיקרית.

הבעיה עם האופן בו הוגדרה קודם לכן המחלקה Stack הייתה כי כל פונקציה יכלה לגשת ישירות לשדות של אובייקטים מטיפוס Stack.

בכדי למנוע זאת הוספו לשפה ה access modifier הקובעים למי מותר לגשת לאיזה חלק במחלקה.

private - רק למתודות של המחלקה מותר לגשת לשדות פרטיים או לקרוא למתודות פרטיות.

public - כל אחד יכול לגשת לחלק הציבורי.

```
struct Stack {
  private:
      int* array;
      int size, top_index ;
  public:
      Result init (int size) ;
      Result destroy() ;
      Result push (int e);
      ....
  } ;
```

שמוש ב- class במקום ב- struct

מקובל להגדיר מחלקות ע"י המילה השמורה **class** ולא **struct** בה משתמשים לרוב לייצוג מבנים (מחלקות ללא מתודות).

ההבדל בין struct ל class הנו כי ב- class, אלא אם כן צוין אחרת, אופן הגישה הנו private בעוד שב- struct המוד הדיפולטי הנו public.

```
class Stack {
      int* array;
      int size, top_index;
public:
      Result init (int size) ;
      Result destroy() ;
      Result push (int e);
      Result pop ();
      Result top(int& e);
      Result print() ;
} ;
```

Default Parameters

לעיתים פונקציה צריכה ברוב המקרים פרמטר בעל ערך קבוע, ורק במקרים נדירים ערך אחר. נרצה שבמקרה השכיח לא נצטרך להעביר את הפרמטר הנ"ל, אלא שזה יבחר באופן דיפולטי על ידי הקומפיילר.

לדוגמא: הדפסת מספרים בבסיסים שונים. לרב הבסיס יהיה 10.

```
void print_num(int num, int base = 10) ;
void f() {
    print_num(24,7);
    print_num(12,10);  // no need to send 10.
    print_num(12);
}
```

Default Parameters

מימוש של פונקציה עם פרמטרים דיפולטים הנו רגיל. ההבדל היחיד הנו באופן הקריאה. אם לא יסופק פרמטר אזי הפונקציה תקבל את הערך הדיפולטי.

ניתן לתת ערכי ברירת מחדל רק לפרמטרים האחרונים:

```
int f1(int a = 0, int b = 0, int c = 0);        // Ok
int f2(int a , int b = 0, int c = 0);           // Ok
int f3(int a = 0, int b = 0, int c);            // Error
int f4(int a = 0, int b, int c = 0);            // Error
```

את ערכי ברירת המחדל יש לכתוב פעם אחת בלבד (רצוי בהצהרה הראשונה על הפונקציה).

Friend functions

- אם נרצה בכל זאת לאפשר לפונקציות אשר אינן מתודות של המחלקה גישה לחלק ה private של המחלקה, נוכל להגדירן כ- friend של המחלקה
- נרצה לבצע זאת לרוב למען מימוש יעיל יותר של הפונקציה

```
class Stack {
 private:
        int* array;
        int size, top_index;
        friend  int second (Stack);
public:
        …
```

```
int second(Stack s) {
   if (s.top_index < 2)
         exit(1);
   return s.array[s.top_index-2];
}
```

Friend classes and Methods

```
class A { ….
   int f() ; …
};

class B { …
};

class C {
   friend int A::f();
   friend class B;
};
```

method f() of class A is a friend of class C

all methods of class B are friends of class C

- בדומה ל **friend function** ניתן להגדיר גם method של מחלקה אחת כ- Friend של מחלקה אחרת
- ניתן להגדיר מחלקה שלמה כחברה של מחלקה אחרת ואז כל ה- methods של אותה מחלקה יוכלו לגשת לחלק ה- private של המחלקה האחרת
- השימוש ב Friends פוגע בקריאות התוכנה, ולכן מומלץ לא להשתמש בכך פרט למקרים יוצאי דופן

Function Overloading

נניח נרצה לכתוב פונקציה המעלה בחזקה.

```
double power(double x, double y);
```

בעיה - יעילות. נרצה מימוש שונה כאשר המעריך מטיפוס שלם.

פתרון סטייל C:

```
double power_dd(double x, double y);
double power_di(double x, int y);
int power_ii(int x, int y);
```

מה רע? פתרון סטייל C++:

```
double power (double x, double y);
double power (double x, int y);
int power (int x, int y);
```

Function Overloading

הפתרון שהוצג מסתמך על כך, שב C++ פונקציה מזוהה על פי שמה והפרמטרים שהיא מקבלת.

כאשר ישנן מספר פונקציות בעלות שם זהה, הקומפיילר ינסה להחליט מהי הפונקציה ה"קרובה" ביותר מבחינת סוג הפרמטרים המועבר, ולה יקרא. אם הקומפיילר אינו מסוגל להחליט אזי הוא מוציא הודעת על שגיאת ambiguity - חוסר בהירות. בתהליך ההחלטה לא מתחשבים בטיפוס ערך ההחזרה של הפונקציה.

מקובל להשתמש ב Function Overloading כאשר יש פונקציות המבצעות פעולה דומה על טיפוסים שונים (לדוגמא power או print).

Function overloading resolution rules

1 void print(int) ; 2 void print(const char*) ;

3 void print(double) ; 4 void print(long) ;

```
char c; int i ; short s; float f;
print( c );    print(i);        print(s) ;     print(f) ;
print('a');    print(45);       print(0.0) ;   print ("a");
```

rules:

1. exact match (+trivial int ->int&, to const) .
2. match with promotions (char,short int -> int, float->double) .
3. match with conversions (int->float, float -> int) .
4. match with user defined type conversions.
5. match with ellipsis (…)

member function overloading

- בדומה ל **function overloading** ניתן להגדיר גם מספר מתודות בעלות שם זהה, אולם עם חתימה שונה

```
class calculator { …
    double power(double, double);
    double power(double, int);
    int power(int, int); …
```

- יכולת זו שימושית במיוחד עבור **constructors** ניתן להגדיר מספר constructors אשר כל אחד מקבל פרמטרים שונים וכך לאפשר מספר צורות אתחול

const member functions I

- כאשר מתודה אינה משנה את מצב האובייקט עליו היא נקראת (לדוגמא size() במחלקה Stack) ניתן לציין זאת ע"י הוספת const בסוף ההצהרה על המתודה
- הסיבה שרצוי לבצע זאת הנה כי עבור const objects מותר לקרוא רק ל const methods

```
class C {
   int read() const;
   void write(int j);
   …
};
```

```
int func (const C& con, C& mut) {
   con.read () ;        // ok
   con.write(5);        // error
   mut.read();          // ok
   mut.write(5);        // ok
}
```

const member functions II

- הגדרת מתודה כ- const מהווה חלק מחתימתה
- יתכנו שתי מתודות עם אותו שם ואותם פרמטרים, אך אחת const והשנייה לא

```
class C { …
   int f();     // will be called for non const objects
   int f() const; //will be called for const objects
};
```

inline functions I

- קריאה לפונקציות הנה פעולה בעלת תקורה יקרה
- עבור פונקציות פשוטות חבל לשלם מחיר זה
- ב-C היינו נעזרים ב MACRO. ב-++C נעזר ב- inline function
- **inline function** מוגדרת כפונקציה לכל דבר, אולם קומפיילר חכם יבחר לממש אותה בצורה שתמנע את התקורה של פונקציה רגילה. למשל, על ידי השתלת גוף הפונקציה בנקודת הקריאה בדומה למקרו. הדבר עלול לגרום לניפוח קוד ולא תמיד אפשרי (רקורסיה ...) אך לעתים קרובות מייעל את הקריאה לפונקציה

inline functions II

ניתן להגדיר **inline function** בשתי דרכים

- כחלק מהגדרת המחלקה:

```
class Stack {
   int _size, top_index; int* array;
public:
   int size() { return _size ;}  …
```

- להוסיף inline לפני המימוש. בכל מקרה על פונקציה זו להיות ממומשת ב-**header file**.

```
class Stack {
   int _size, top_index; int* array;
public:
   int size()  };
inline int Stack::size() { return _size ;} …
```

קלט / פלט ב C++

ב ++ C קלט ופלט מתבצע ע"י "הזרמת" נתונים באמצעות האופרטורים >> ו-<<.

Input / Output in C++:

```
#include <iostream>
using namespace std;

int j , k = 123;
double d;
cout << "A string " << k;
cin >> j >> d;
```

Input / Output in C:

```
#include <stdio.h>

int  j,k = 123;
double d;
fprintf (stdout,"%s %d", "A string" , k);
fscanf (stdin, "%d  %lf", &j, &d);
```

יתרונות C++:

- הקומפיילר מזהה את טיפוסי המשתנים לבד (חוסך טעויות).
- בקלט, משתמשים במשתנים - לא בכתובותיהם (חוסך טעויות וגם יותר "נקי").

ערוצים סטנדרטים ב-C ++

- הקלט והפלט ב-C++ נעשה ע"י ערוצים (streams). ערוצים הם **מחלקות** המוגדרות כחלק מהספרייה הסטנדרטית של C++. הקלט והפלט של עצם מתבצעים ע"י שימוש בפונקציות **ואופרטורים** המוגדרים עבור מחלקה זו. מחלקות אלו מוגדרות בקובץ **iostream**, ולכן על מנת להשתמש בהם צריכה להופיע בתחילת הקובץ השורה:

#include <iostream>

- אופרטורי הקלט והפלט חכמים יותר מפקודות הקלט/פלט של שפת C בכך שאין צורך לציין את סוג המשתנים.
- ערוצי הפלט מוגדרים במחלקה **ostream**. קיימים שני ערוצי פלט סטנדרטיים:
  - ערוץ הפלט הסטנדרטי - cout
  - ערוץ הודעות השגיאה הסטנדרטי - cerr
- ערוצי הקלט מוגדרים ע"י המחלקה **istream**.
  - קיים ערוץ קלט סטנדרטי אחד - cin .

קלט פלט ב C++

פלט האופרטור המשמש להזרמת נתונים לתוך ערוץ פלט הוא "<<":

<ostream> << <data>

- ניתן גם להשתמש בפונקציה put על מנת לפלוט נתונים, בצורה הבאה:

<ostream>.put(<data>);

- לדוגמא, נניח כי x הוא משתנה שלם וערכו 3, ו- y הוא משתנה ממשי שערכו 2.45

cout << “Hello " << x << " " << (x+y);

Hello 3 5.45

קלט האופרטור המשמש לשאיבת נתונים מתוך ערוץ קלט הוא "<<":

<istream> >> <variable name>

- ניתן גם להשתמש בפונקציה get על מנת "לשאוב" נתונים, בצורה הבאה:

<istream>.get(<variable name>);

- דוגמא:

cin >> x >> y;

- קורא לתוך x את הנתון הראשון המופיע בקלט ולתוך y את הנתון השני המופיע בקלט.

דוגמא לקלט ופלט

- ערוצי הקלט וערוצי הפלט הם אובייקטים. ישנן מתודות נוספות אשר ניתן להפעיל על אובייקטים אלו. למשל:
- הפונקציה **eof()**, הבודקת אם בערוץ מופיע סימן סוף קובץ.
- דוגמא נוספת היא הפונקציה **width()** הקובעת את מספר התווים המינימלי אשר יודפסו בהדפסה הבאה של מחרוזת ומספר, והפונקציה **fill()** אשר קובעת את התו שיודפס במידה והמחרוזת/מספר קצרים מהנדרש. לדוגמא:

```
cout.width(4);
cout.fill('#');
cout << "(" << 12 << "), (" << 12 << endl;
```

הפלט המתקבל:

###(12),(12)

דוגמא: תוכנית זאת קוראת את מה שמופיע בקלט תו אחר תו ומדפיסה את התוים עד שמופיע הסימן לסוף קובץ.

```
#include <iostream>
using namespace std;
void main()
{
    char x;
    while (!(cin.eof())) {
        cin >> x;
        cout << x;
    }
}
```

ערוצי קבצים

- שימוש בערוצי קבצים דומה למדי לשימוש בערוצי ק/פ. ערוצי הקבצים הם מחלקות **הנורשות** מערוצי הק/פ הרגילים ולכן כל הפעולות המוגדרות על ערוצים רגילים מוגדרים גם על ערוצי קבצים. מחלקות אלו מוגדרות בקובץ **fstream.h**, ולכן על מנת להשתמש בהם צריכה להופיע בתחילת הקובץ השורה:

#include <fstream>

- ערוצי הקלט מקבצים מוגדרים ע"י המחלקה **ifstream**. למחלקה זו יש **constructor** המקבל כקלט מחרוזת ופותח את הקובץ ששמו ערך המחרוזת

- ערוצי הפלט לקבצים מוגדרים ע"י המחלקה **ofstream**. למחלקה זו יש **constructor** המקבל כקלט מחרוזת ופותח את הקובץ ששמו ערך המחרוזת

דוגמא לעבודה עם קבצים

```
#include <iostream>
#include <fstream>
using namespace std;

void copy_file(char* a, char* b)
{
        ifstream from(a);    // Open the file a
        if(!from.is_open()) { // if(!from){
            cerr << "cannot open file " << a << endl;
            exit(1);
        }
        ofstream to(b);      // Open the file b
        if(!to.is_open()) { // if(!to){
            cerr << "cannot open file " << b << endl;
            exit(1);
        }
        char c;
        while (!(from.eof())) {
            from.get(c);
            to.put(c);
        }
}
```

namespace

- בעייה ב-C:
  - פרוייקט חולק בין מספר מתכנתים וכעת רוצים לחבר אותו לתוכנית אחת
  - מתכנתים שונים השתמשו באותו שם לפונקציות שונות

```
/* my.h */
….
void print_results(int, int);
….
```

```
/* your.h */
….
void print_results(int, int);
….
```

namespace

- פתרון ב-C++:
  - נעטוף כל חלק בתוכנית ברמת היררכיה נוספת

```
/* my.h */
namespace my{
….
void print_results(int, int);
}
```

```
/* your.h */
namespace yours{
….
void print_results(int, int);
}
```

```
int main(){
        my::print_results(4,5);
        …..
```

namespace

- ב C++ נהוג לכלול מחלקות סטנדרטיות בשמן בלבד, ללא h. בסוף.
- אם כוללים ספרייה סטנדרטית באופן זה, נכללת ספרייה מ-namespace הנקרא std.
- על מנת להשתמש ב-namespace זה עלינו להביא אותו במפורש ל-namespace הנוכחי

```
#include <iostream>
using std::ostream;
```