# ספריית תבניות תקנית - STL

## הקדמה

לפני כ-20 שנה החליטו להוסיף לשפת ++C הבסיסית, ספריה הבנויה מעליה ומשתמשת בעיקר בתבניות (template). הספרייה נקראת STL - Standard Template Library, והיא כיום באה כחלק בלתי נפרד מהשפה.

## מושגים

כל תבנית שאנחנו יוצרים מגיעה לתהליך של קומפילציה רק כאשר אנחנו קוראים לה והגדרנו איזה סוג אובייקטים יכנסו אליה. כדי שהתבנית תתקמפל, הסוגים צריכים לקיים דרישות מסוימות. אוסף של דרישות על סוג נקרא "מושג" - "concept". לדוגמה, נניח שיש לנו תבנית פונקציה המחשבת מינימום. הפונקציה עובדת רק על סוגים שיש להם אופרטור "קטן מ-" (>). המושג " LessThanComparable" מציין כל סוג שיש לו אופרטור "קטן מ-". לכן, אם אנחנו כותבים תבנית של פונקציית מינימום עם פרמטר-סוג T, אנחנו יכולים לכתוב בתיעוד שלה שהסוג T צריך להיות "LessThanComparable".

בתיעוד של הספרייה התקנית, מוגדרים כמה מושגים מרכזיים:

* LessThanComparable - מכילים אופרטור "קטן מ-".
* EqualityComparable - מכילים אופרטור "=="
* Assignable - מכילים בנאי מעתיק ואופרטור השמה. כברירת מחדל כל הסוגים הם כאלה, שהרי מוגדרים אוטומטית על ידי הקומפיילר במקרה שלא הגדרנו אותם, אלא אם כן מחקנו להם את הבנאי המעתיק או אופרטור ההשמה, או שהפכנו אותם לפרטיים.

### השוואה בין מושג ב-C++ לממשק ב-Java

גם בשפת Java יכולנו להגדיר ממשק בשם LessThanComparable עם שיטה מופשטת בשם lessthan, ולהגדיר פונקציית מינימום המקבלת פרמטרים מסוג LessThanComparable. אבל יש שני הבדלים:

1. ב-Java, רק מחלקות יכולות לממש ממשקים. לכן, אם כתבנו פונקציית מינימום על עצמים מסוג LessThanComparable, היא לא תעבוד על סוגים פרימיטיביים כמו int ו-double. בנוסף, אם כתבנו פונקציית מינימום על עצמים מסוג LessThanComparable, היא לא תעבוד על עצמים אחרים מספרייה אחרת שלא מכירים את הממשק LessThanComparable. לעומת זאת, ב ++C המושג "LessThanComparable" משמש לתיעוד בלבד - גם int משתייך למושג הזה כי יש לו אופרטור "קטן מ-". גם מי שאינו מכיר כלל את המושג LessThanComparable יכול להשתייך למושג הזה אם יש לו אופרטור "קטן מ-".
2. החיסרון הוא, שב-++C הודעות השגיאה קשות יותר להבנה. ב-Java הקומפיילר מזהיר אותנו בפירוש כשאנחנו מנסים להפעיל פונקציה עם פרמטר שאינו מממש את הממשק LessThanComparable. ב-++C, הקומפיילר יצעק רק כשיראה שאנחנו מנסים לגשת לאופרטור "קטן מ-" שלא קיים. כתוצאה מכך הודעת השגיאה עלולה להיות קשה יותר להבנה.

## רכיבי הספרייה התקנית

הרכיבים העיקריים של הספרייה התקנית הם: מיכלים, איטרטורים, אלגוריתמים, פונקטורים, מחרוזות וזרמים.

**מיכלים** - אלו מבנים שמכילים אובייקטים כמו: vector, map, queue וכו'.

**איטרטורים** - כל מיכל מגדיר איטרטור להתחלת וסוף המיכל המאפשר לעבור באופן סדרתי על כל האיברים.

**אלגוריתמים** - פועלים על מיכלים. כל אלגוריתם מקבל כקלט איטרטורים המגדירים את התחום שבו האלגוריתם צריך לעבוד. שים לב שאין קשר ישיר בין אלגוריתמים למיכלים. לכאורה, היינו חושבים שהקלט של אלגוריתם צריך להיות מיכל. אולם אז היינו צריכים לכתוב את האלגוריתם מחדש לכל סוג של מיכל. ולכן אם יש n אלגוריתמים ו-m מיכלים, זה יוצא O(mn) עבודה. לעומת זאת, בשיטת האיטרטורים אנחנו צריכים לממש כל אלגוריתם פעם אחת ואיטרטורים לכל מיכל, סך הכל O(m+n) עבודה.

**פונקטורים** - חלק מהאלגוריתמים צריכים לקבל פונקטור שלפיהם הם מבצעים את פעולתם.

**מחרוזת** - מיכל מיוחד המשמש בעיקר לאחסון תווים ולבצע פעולות התאימות למחרוזות.

## הודעות שגיאה

אחד הקשיים העיקריים בעבודה עם STL הוא הודעות השגיאה. למשל, אם ננסה להריץ את אלגוריתם sort על list, לא נקבל הודעה פשוטה שאומרת "אי אפשר להריץ sort על list", אלא הודעה ארוכה ומסובכת הנכנסת לפרטי התבניות בספריה התקנית. כדי לפענח את הודעת השגיאה, צריך לחפש את ה-note המפנה לשורה בקוד שלנו, ומשם לנסות להבין מה הבעיה.

ישנן ספריות המנסות לתת הודעות שגיאה משמעותיות יותר, למשל boost ו-STLFilt.

## ספריות נוספות

בנוסף לספריה התקנית, יש ספריות נוספות. המקובלת ביותר היא boost. ספריה זו נחשבת "כמעט" תקנית. למעשה הרבה מהדברים בספריה התקנית התחילו את דרכם ב-boost, השתכללו והשתפרו, עד שבסוף נכנסו לספריה התקנית. לכן, אם חסר משהו בספריה התקנית, חפשו ב-boost.

# מיכלים - Containers

## סוגי מיכלים

ב-STL יש ארבעה סוגי מיכלים: גודל קבוע (Fixed Size), סדרתיים (Sequential), אסוציאטיביים (associative), ומתאמים (Adaptors). ניתן לראות את כל המיכלים וטבלת השוואה ביניהם [כאן](http://www.cplusplus.com/reference/stl/). כל מיכל יכול להכיל כל סוג של אובייקט.

פרט חשוב שצריך לשים לב אליו במיכלים ב-STL, הוא שמיכלים מחזיקים עותקים של אובייקטים, לא מצביעים או רפרנסים (כמו ב-Java), אלא עותק שלם. יש לכך שתי משמעויות חשובות:

1. אפשר להכניס למיכל רק סוג אובייקט המשתייך למושג Assignable, כלומר יש לו בנאי מעתיק ואופרטור השמה.
2. בכל פעם שמכניסים עצם למיכל, נבנה עצם חדש באמצעות בנאי מעתיק. ובכל פעם שמפרקים מיכל, מתפרקים כל העצמים הנמצאים בו.

## גודל קבוע

אלו הם מיכלים שכל האיברים בהם נשמרים במקום רציף על המחסנית ולא בערימה, ולכן חייב להיות להם גודל קבוע בזמן קומפילציה. כלומר צריך לציין להן גודל כבר בשעת ההצהרה עליהן, ולא ניתן להכניס או למחוק מהם איברים לאחר ההצהרה, אלא רק להחליף ערכים. כל שאר המיכלים שאינם תחת קטגוריה זו נשמרים על הערימה, ולכן גודלם יכול להשתנות תוך כדי ריצה.

### pair

זוהי תבנית שמקבלת שני סוגים ומתייחסת אליהם כיחידה אחת. אפשר לבנות pair באמצעות בנאי רגיל או להכניס שני ערכים בסוגריים מסולסלות. הגישה לאיברים באמצעות שיטות first ו-second ללא סוגריים. כדי לקבל את שני האיברים למשתנים ניתן להשתמש בדרך מקוצרות באמצעות אופרטור [].

pair<int,string> p = {1,"cpp"}; //same as pair<int,string> p{1,"cpp"}

int i = p.first; // i = 1

string s = p.second; // s = "cpp"

auto[a,b] = p; // a = 1, b = "cpp"

שיטה שימושית נוספת היא make\_pair המאפשר לבנות pair ללא שנפרט את סוג האובייקט.

auto p = make\_pair(1,"cpp");

### tuple

זהו הרחבה של pair למספר קבוע של איברים. כדי לגשת לאיבר ב-tuple נשתמש בשיטה get<i>(tuple\_name) המחזירה את האיבר במקום ה-i. כל הקיצורים שהראינו ב-pair קיימים גם ב-tuple.

auto t = tuple{5,'a',"hello"};

int i = get<0>(t); // i = 5

auto[a,b,c] = t; // a=5, b='a', c="hello"

השימוש העיקרי במיכל זה הוא כאשר רוצים להחזיר מספר אובייקטים מפונקציה. במקרה זה ערך החזרה יהיה auto ונחזיר tuple כלשהו.

auto f() {

return tuple{5,'a',"hello"};

{

auto[a,b,c] = f(); // a=5, b='a', c="hello"

### array

זהו המערך הרגיל שאנו מכירים שמכיל אובייקטים **מסוג אחד בלבד**, וגודלו ידוע בזמן קומפילציה.

## מיכלים סדרתיים

אלו הם מיכלים ששומרים את האיברים לפי הסדר שהמתכנת הכניס אותם. כל האיברים במיכלים אלו נשמרים בערימה, ולכן גודל המיכלים יכול להשתנות תוך כדי ריצה. יש ארבעה סוגי מיכלים סדרתיים הנבדלים בעיקר בסיבוכיות הזמן הנדרשת לביצוע פעולות שונות (בסוף הסעיף תמונה מסכמת).

### forward\_list

זוהי רשימה מקושרת שבה כל איבר יודע רק לגשת לאיבר הבא אחריו. הכנסת איבר, לא משנה היכן תהיה, מאוד מהירה אם ידוע איטרטור לאיבר שרוצים להכניס אחריו. כדי למצוא איבר צריך לעבור באופן סדרתי מהאיבר הראשון עד שמגיעים לאיבר הרצוי, לכן סיבוכיות O(n).

### list

כמו forward\_list אלא שכל איבר יודע לגשת קדימה ואחורה, ולא רק קדימה. היתרון היחיד על פני forward\_list הוא שהאיטרציה מהסוף להתחלה הרבה יותר מהירה O(n) ולא O(n2).

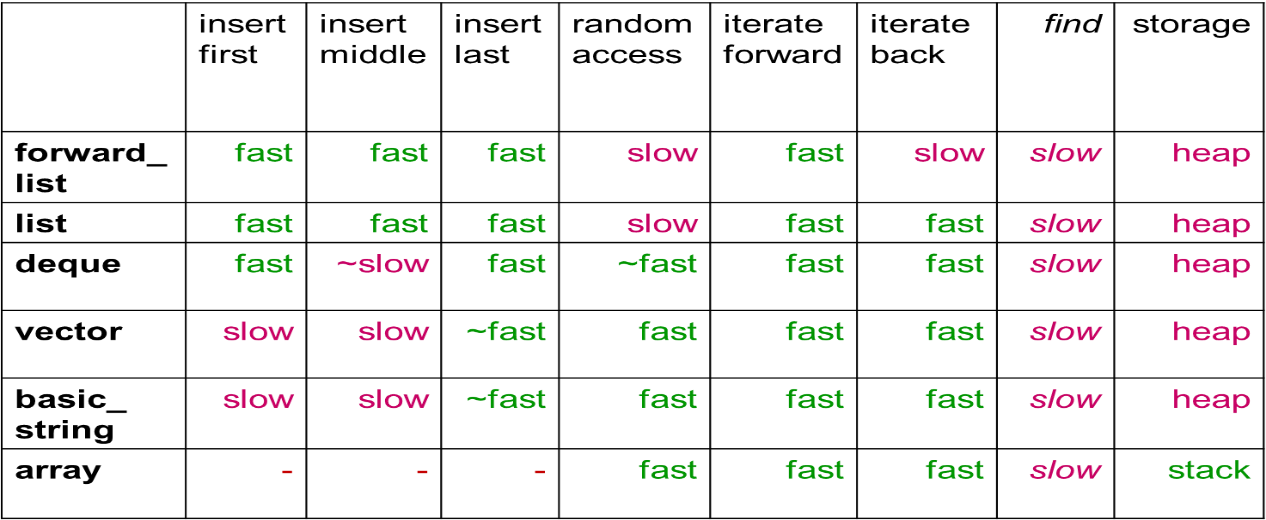
### vector

בדומה למערך הוא גוש זיכרון אחד רציף המכיל את כל האיברים, אלא שבניגוד למערך נשמר על הערימה ולא במחסנית, ולכן ניתן להכניס ולמחוק איברים. היתרונות של וקטור הוא שהחיפוש מאוד מהיר, אך הכנסת איבר בהתחלה או באמצע, ולעיתים גם בסוף, דורשת להזיז את כל הזיכרון, ולכן מאוד איטית.

### deque - תור דו-כיווני

דומה מאוד לוקטור, אך מחלק את הזיכרון למספר גושים רציפים ולא גוש אחד גדול. בדרך כלל ממומש באמצעות וקטור שבו כל איבר הוא מצביע לוקטור אחר. הוקטור הראשי שומר מקום פנוי גם בהתחלה וגם בסוף להכנסת איברים בהמשך.

הכנסת איבר בהתחלה או בסוף בזמן קבוע O(1) אם יש מקום פנוי בווקטור המשני הראשון או האחרון. אחרת, צריך להעתיק את הוקטור הראשון/אחרון ולעדכן את הוקטור הראשי. הכנסת איבר באמצע לעומת זאת איטית יותר אך יותר טובה משל וקטור, שכן לכל היותר צריך להעתיק רק את אותו הבלוק בו מכניסים איבר ולא את כל המיכל.

חיפוש איבר הוא בזמן קבוע O(1). קודם הולכים לוקטור הראשי, משם לוקטור המשני המתאים, ושם מוצאים את האובייקט הרצוי.

## מיכלים אסוציאטיביים

אלו הם מיכלים שבהם ניתן לגשת לנתונים לפי מפתחות. האיברים במיכלים אלו שמורים על הערימה ולכן ניתן לשנות את גודלם תוך כדי ריצה. המיכלים אינם מסודרים לפי סדר ההכנסה אלא בסדר אחר. יש שני סוגים של מיכלים אסוציאטיביים: מסודרים ולא מסודרים.

### מסודרים

בו המפתחות מסודרים כברירת מחדל מהקטן לגדול לפי האופרטור "קטן מ-" (>) של סוג האובייקט שהוכנס אליהם. הסטנדרט של השפה היא שבמיכלים אלו פעולות הכנסה, מחיקה וחיפוש מתבצעים בסיבוכיות של O(log(n)). בדרך כלל המימוש של סוג זה של מיכלים יהיה באמצעות עצים מאוזנים. יכול להיות עץ בינארי, אדום שחור, AVL, B וכו'. יש ארבעה מיכלים בקבוצה זו: set, map, multiset, multimap.

כאמור, סדר "קטן מ-" מוגדר רק כברירת מחדל במיכלים האסוציאטיביים המסודרים. ניתן גם ליצור מיכל אסוציאטיבי עם סדר שונה מהרגיל. לשם כך יש להעביר פונקטור מתאים כפרמטר שני לתבנית המיכל. למשל, כדי ליצור set של int שבו האיברים יסודרו בסדר הפוך מהקטן לגדול לפי אופרטור "גדול מ-" (<), ניצור את התבנית הבאה:

set<int, greater<int> s;

### לא מסודרים

בסוג זה של מיכלים המפתחות מסודרים לפי סדר כלשהו שנוח לקומפיילר. הסטנדרט של השפה הוא שבמיכלים אלו פעולות הכנסה, מחיקה וחיפוש מתבצעים בסיבוכיות של O(1). בדרך כלל המימוש יהיה באמצעות טבלאות גיבוב למיניהם. יש ארבעה מיכלים בקבוצה זו: unordered\_set, unordered\_map, unordered\_multiset, unordered\_multimap. מיכלים אלו מוגדרים באופן זהה למיכלים האסוציאטיביים המסודרים אלא שמבנה הנתונים איתו משתמשים במימוש המיכל שונה, כמו שפירטנו.

### set

קבוצה שמכילה מפתחות בלבד, ואינה יכולה להכיל אותו מפתח פעמיים. מיכל זה שימושי בעיקר כאשר אנו עוברים על קבוצת איברים, ואנו רוצים לזכור אילו איברים כבר ראינו.

### map

במיכל זה כל מפתח ממופה לערך אחד, כלומר כל איבר הוא בעצם pair של מפתח וערך. מיכל זה גם כן אינו מאפשר להכיל שני מפתחות זהים. מיכל זה שימושי כאשר רוצים לשמור נתון נוסף על המפתח. לדוגמה, במעבר על קבוצת איברים נשמור בערך את מספר הפעמים שראינו את המפתח. זהו מיכל מאוד שימושי ומעניין, ולכן נרחיב עליו בהמשך.

### multiset ו-multimap

multiset הוא כמו set רגיל, אלא שמאפשר שיהיו שני מפתחות זהים. multimap גם כן כמו map רגיל, אלא שכל מפתח יכול להוביל למספר ערכים ולא ערך אחד בלבד כמו ב-map.

## מתאמים

מתאם זהו מיכל שיכול להכיל כל סוג נתונים. בנוסף לסוג הנתונים, המתאם מקבל מיכל כלשהו כפרמטר ומוסיף לו שיטות כדי להתאים אותו לממשק רצוי. אפשר להכניס גם מיכל שהוא מחלקה שאנו יצרנו, אך צריך שמחלקה זו תשתייך למושג כלשהו כדי שיהיה ניתן להכניס אותו כפרמטר למתאם. יש שלושה מיכלים מסוג זה: stack, queue, priority\_queue.

### stack

מתאם ההופך כל מיכל סדרתי למחסנית על ידי הוספת שיטות הניגשות לאיבר הכי חדש במחסנית, כמו push, pop. בברירת מחדל ממומש באמצעות deque.

### queue

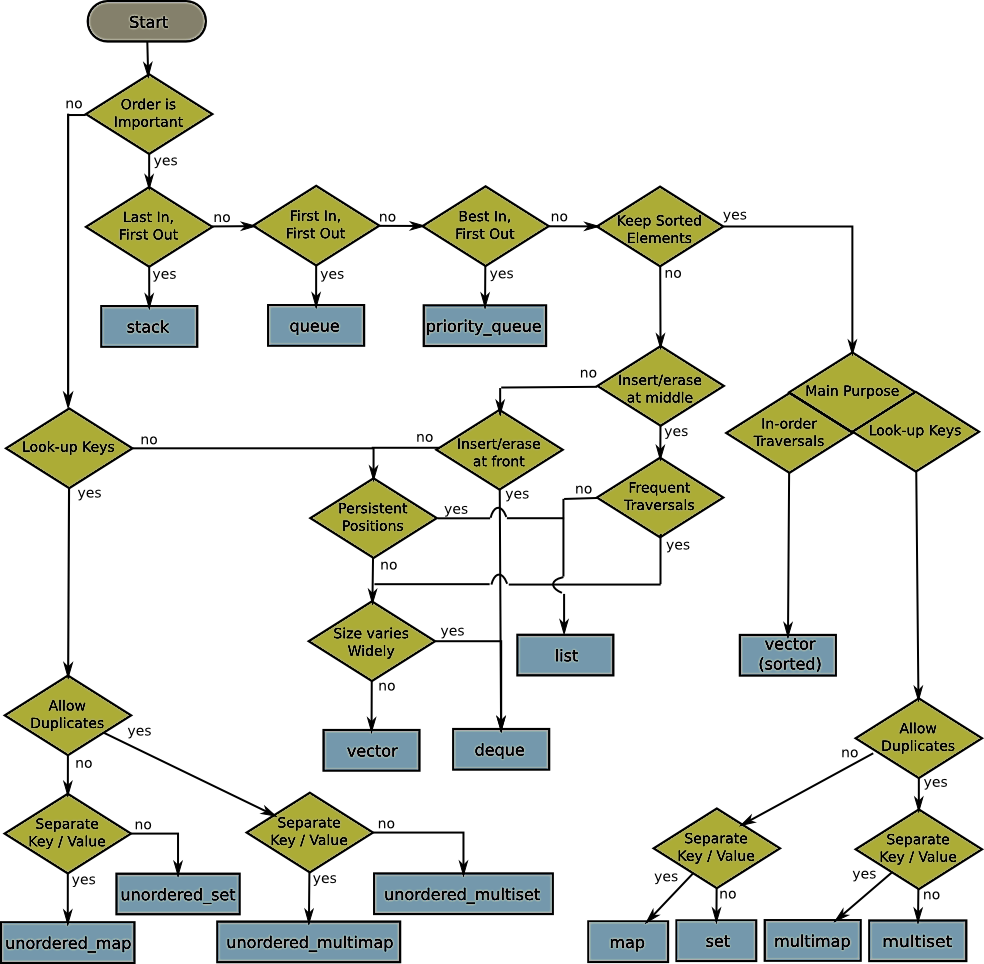
מתאם ההופך כל מיכל סדרתי לתור על ידי הוספת שיטות הניגשות לאיבר הכי ישן בתור. בברירת מחדל ממומש גם כן באמצעות deque.

### priority\_queue

מתאם ההופך כל מיכל סדרתי לתור עדיפויות על ידי הוספת שיטות הניגשות לאיבר הכי טוב בתור. בברירת מחדל ממומש באמצעות vector.

## איזה מיכל לבחור

בבואנו לבחור מיכל שנרצה להשתמש בו בקוד שלנו, נוכל להיעזר בדיאגרמה למטה, שבה נמצאים כל סוגי המיכלים עליהם פירטנו בקצרה בפרק זה. כדי למצוא את המיכל המתאים נתחיל מלמעלה ונענה על השאלות (כן/לא), עד שנגיע למיכל שהוא הכי מתאים לפי התשובות שלנו.



## vector

עבור כל סוג מיכל נרחיב על המיכל הכי שימושי ויעיל מבין המיכלים בקבוצה שלו. וקטור vector<T> בדומה למערך הוא גוש זיכרון אחד רציף המכיל את כל האיברים, ולכן הגישה לאיבר מאוד מהירה אך ההכנסה והמחיקה איטית. הוא נשמר על הערימה ולכן ניתן להכניס ולמחוק ממנו איברים תוך כדי ריצה. ניתן לאחסן בו כל סוג משתנים, כמו מחלקות מורכבות ואף משתנים פרימיטיביים, כמו int ו-char.

### גודל וקיבולת

כדי לייעל את פעולת ההכנסה, לכל וקטור מוקצה זיכרון מעבר למספר האיברים שיש בו, השמור לו למקרה שהמשתמש יוסיף לו עוד איברים. מספר האיברים שיש בוקטור נקרא גודל הוקטור - size. ואילו מספר האיברים שיש להם מקום בוקטור נקרא קיבולת הוקטור - capacity. הגודל תמיד שווה או קטן מהקיבולת. ניתן לגשת לגדלים אלו בכל רגע נתון באמצעות שיטות size() ו-capacity(). שים לב שהזיכרון מאינדקס size() ועד capacity()-1 אמנם מוקצה לוקטור אך אינו מוגדר. כשמוסיפים עצם בסוף הוקטור, יש שתי אפשרויות:

* האפשרות הקלה היא שהגודל לאחר ההוספה עדיין שווה או קטן מהקיבולת. במקרה זה צריך רק לבנות/להעתיק את העצם החדש למקום הפנוי בסוף הוקטור.
* האפשרות הקשה היא שהגודל לאחר ההוספה גדול יותר מהקיבולת. במקרה זה צריך להגדיל את הקיבולת. לשם כך יש לאתחל מקום בזיכרון עם קיבולת גדולה יותר, להעתיק את כל האיברים מהזיכרון הישן אל החדש, ולפרק את כל האיברים הישנים. מקובל להגדיל את הקיבולת פי 2 בכל פעם שמצב כזה מתרחש.

לכן, אם ידוע גודל הוקטור עדיף כבר בשעת ההצהרה לתת אותו לוקטור ולמנוע העתקות כפולות אלו. ניתן גם להקצות ידנית את גודל הקיבולת באמצעות שיטת reserve(uint new\_capacity). אנו יכולים גם לכווץ את הקיבולת כך שתשתווה לגודל באמצעות שיטת shrink\_to\_fit().

### אתחול

יש מספר רב של דרכים לאתחל וקטור:

vector<myclass> vec; // empty vector

vector<myclass> vec(10); // 10 identical objects with default constructor

vector<myclass> vec(10, myclass(5)); // 10 objects with specified constructor

vector<int> vec{{5},{3,0},{1}} // 3 objects with different constructor

vector<int> vec{c.begin(), c.end()} // initialized with exist container

### הוספת איבר

יש שתי שיטות להוסיף עצם לסוף של וקטור ולהרחיב את הגודל על חשבון הקיבולת:

* + push\_back - מקבלת עצם מסוג T ומעתיקה אותו לסוף הוקטור, על ידי שימוש בבנאי מעתיק. אם בבנאי התקבל עצם זמני, אזי עצם זה יפורק בסוף התהליך.
  + emplace\_back - מקבלת פרמטרי אתחול לעצם מסוג T, ומשתמשת בהם כדי לבנות עצם חדש בסוף הוקטור, על ידי שימוש בבנאי המתאים של העצם. שיטה זו יעילה יותר מהראשונה כי היא חוסכת את הצורך ליצור עצם זמני ולפרק אותו, שהרי יוצרת את העצם ישירות במקום שלו.

גם כדי להכניס עצם באמצע הוקטור יש שתי שיטות:

* Insert - מקבלת עצם בנוי מסוג T, ואיטרטור לתוך הוקטור במיקום המתאים. השיטה תכניס את העצם הבנוי במקום שעליו מצביע האיטרטור.
* emplace - מקבלת פרמטרי אתחול לבנאי של T ואיטרטור, ובונה בעזרתם עצם חדש במקום שעליו מצביע האיטרטור. השיטה השנייה יעילה יותר כי היא חוסכת את יצירת העצם הזמני. אמנם שתיהן צריכות להזיז חלק גדול מהאיברים בוקטור ולכן הן לוקחות זמן ליניארי בגודל הוקטור.

### גישה לאיבר

הגישה לאיבר בוקטור יכולה להיעשות באמצעות אופרטור [i], המחזיר את האיבר במקום ה-i, או באמצעות שיטה at(i). ההבדל ביניהם הוא שבאופרטור [] לא מתבצעת בדיקה שהאיבר במקום זה אכן מוגדר במערך, ולכן אם ניגש לאיבר שלא מוגדר נקבל שגיאת זמן ריצה. ואילו בשיטת at מתבצעת בדיקה כזו, ולכן אם ניגש למיקום לא מוגדר, תתקבל שגיאת קומפילציה. אנו תמיד מעדיפים שגיאת קומפילציה על פני שגיאת זמן ריצה.

vector<int> vec(10);

vec[20] = 0; // run time error

vec.at(20) = 0; // compilation time error

### שיטות נוספות

לכל וקטור יש כמה טיפוסים הקשורים אליו, ואפשר לגשת אליהם בעזרת ארבע נקודות '::', ולאחריהם שם הטיפוס.

* value\_type - הסוג של העצמים בוקטור (שווה לסוג T המועבר כפרמטר לוקטור).
* reference - רפרנס לעצם בוקטור (שווה ל T&).
* const\_reference - רפרנס לעצם קבוע (שווה ל const T&).
* iterator - איטרטור על הוקטור.
* const\_iterator - איטרטור קבוע על הוקטור.

## map

במפה כל מפתח ממופה לערך, כלומר כל איבר הוא בעצם pair של מפתח וערך. באמצעות אופרטור סוגריים [] שבתוכו נכניס מפתח כדי לקבל רפרנס של הערך המתאים למפתח שהכנסנו. במידה ואין מפתח כזה במפה, אזי המפתח הזה ייווצר ויוחזר רפרנס של ערך ברירת מחדל כלשהו. לכן אופרטור סוגריים זוהי הדרך הנוחה ביותר ליצירה, כתיבה וקריאה של איברים מתוך המפה.

map<string, int> m;

m["abc"] = 300; // create pair {"abc",300}

כדי להחזיר איבר שלם של המפה, ולא רק את הערך, נשתמש בשיטת find(T key) המקבלת מפתח ומחזירה איטרטור לאיבר שבתוכו נמצא המפתח. אם המפתח הזה לא קיים במפה יוחזר איטרטור השווה לאיטרטור end() המצביע לאיבר אחרי האחרון במפה. לאחר שקיבלנו איטרטור של המפה, נשלוף את האיבר שמצביע אליו באמצעות אופרטור '\*'. כדי לגשת למפתח של האיבר t שהתקבל נרשום t.first, וכדי לגשת לערך נרשום t.second.

auto find\_abc = m.find("abc");

if(find\_abc == m.end())

cout << "there is no such key in the map" << endl;

auto at\_find\_abc = \*find\_abc;

find\_abc->second = 500;

הדרך הנוחה ביותר לעבור בלולאה על כל איברי המפה היא באמצעות לולאת for each שהערכים יתקבלו אל משתנה מסוג auto[key,val].

for(auto [key,val] : m) {

cout << key << ',' << val << endl;

## מיכלים נוספים

### basic\_string

זוהי תבנית של מיכל, המסוגל להכיל כל סוג של אובייקטים. אמנם הוא משמש בעיקר למחרוזות, ולכן מוגדרים בו שיטות רבות ומיוחדות כדי לטפל במחרוזות, כמו חיפוש תת-מחרוזת, שרשור ועוד. למעשה מחלקת string הבסיסית המובנית בשפה, היא לא יותר מאשר מיכל זה שהאובייקט שהוא מכיל הוא char, כלומר string = basic\_string<char>.

basic\_string<char> s; //same as "string s;"

הסיבה שהגדירו basic\_string ולא string אחד, הוא כדי להתאים מחרוזות לשפות קידוד שונות, כמו ASCII, Unicode, utf-8 וכו'.

כדי להפוך ערך כלשהו למחרוזת, אפשר להשתמש בשיטה הגלובלית to\_string, או ב-ostringstream, או בסיומת s. למשל s"abc" עם סיומת s מציין אובייקט string המכיל "abc".

### valarray

זהו מיכל של מספרים המייצג וקטור. המיוחד בו שאפשר לחבר שני valarray ולקבל valarray המייצג את הסכום הוקטורי של שני הוקטורים.

valarray a {1,2,3};

valarray b {4,5,6};

valarray c = a+b; //c = {5,7,9}

# איטרטורים

## הגדרה

איטרטור הוא תבנית עיצוב שמטרתו לעבור על כל האיברים של מיכלים (Containers) באופן סדרתי. בפועל האיטרטור הוא אובייקט מסוים שמצביע על איבר כלשהו במיכל. בפעם הראשונה שניצור אותו יצביע לאיבר הראשון במיכל, ובאמצעות אופרטורים ושיטות המוגדרות על האיטרטור, נוכל לקדם אותו כל עוד לא הגענו לאיבר האחרון במיכל.

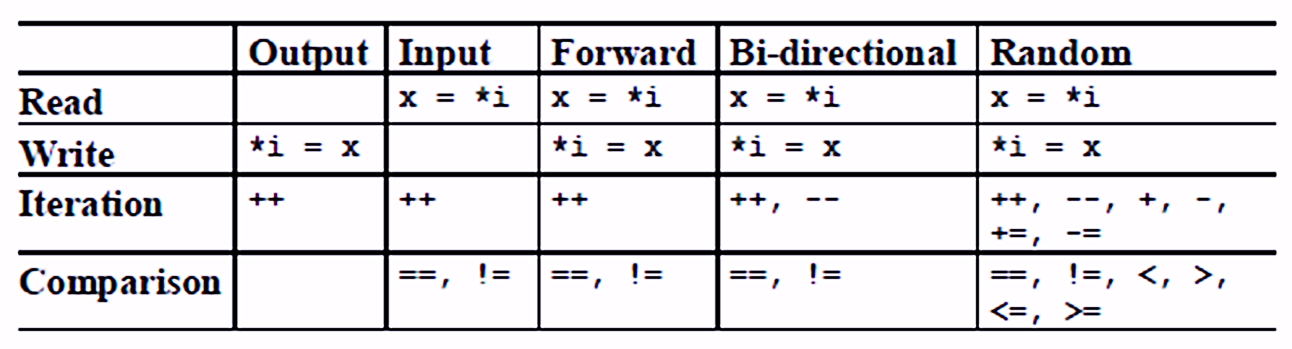
הספרייה התקנית ב- C++ (STL) עובדת בעיקר עם איטרטורים. כל האלגוריתמים בספרייה אינם מקבלים כקלט מיכל אלא איטרטורים המגדירים את התחום שבו האלגוריתם צריך לעבוד. לכאורה, היינו חושבים שהקלט של אלגוריתם צריך להיות מיכל. אולם אז היינו צריכים לכתוב את האלגוריתם מחדש לכל סוג של מיכל. ולכן אם יש n אלגוריתמים ו-m מיכלים, זה יוצא O(mn) עבודה. לעומת זאת, בשיטת האיטרטורים אנחנו צריכים לממש כל אלגוריתם פעם אחת ואיטרטורים לכל מיכל, סך הכל O(m+n) עבודה. בנוסף, באמצעות האיטרטור האלגוריתמים יכולים לעבוד גם על מבנים שאינם מיכלים, כמו range ו-accumulate, ואפילו על ערוצי זרימה. מסיבה זו, כדי להבין את הספרייה התקנית חשוב להבין את סוגי האיטרטורים המוגדרים בה.

## סוגי איטרטורים

לכל אלגוריתם יכולים להיות דרישות שונות מהאיטרטור עליו פועל. ישנם אלגוריתמים שמשתמשים באופרטורים רבים על האיטרטור, וישנם שמשתמשים במעט אופרטורים. כדי לספק תיעוד מדויק של הדרישות מהאיטרטורים, חילקו אותם ל-5 סוגים. ניתן לראות בתמונה למטה אילו אופרטורים מוגדרים עבור כל סוג.

* 1. איטרטור קלט (input iterator) - משמש לקריאה בלבד, אפשר להזיז אותו קדימה בלבד (++).
  2. איטרטור פלט (output iterator) - משמש לכתיבה בלבד, אפשר להזיז אותו קדימה בלבד (++).
  3. איטרטור קדימה (forward iterator) - שילוב של איטרטור קלט ופלט, יכול לשמש לקריאה ולכתיבה.
  4. איטרטור דו-כיווני (bidirectional iterator) - כמו איטרטור קדימה, רק שיכול גם לזוז אחורה (--).
  5. איטרטור גישה אקראית (random access iterator) - מקביל לפוינטר. מאפשר קריאה, כתיבה, הזזה קדימה ואחורה, ובנוסף אפשר לבצע עליו אריתמטיקה באמצעות אופרטורים '+' '-'.

vector<int> v{10,5,7,4,2};

cout << \*(**v.begin()+3**) << endl; //will print 4

המיכלים השונים מציעים איטרטורים ברמות שונות, למשל:

* + זרמים מציעים איטרטורי קלט, פלט וקדימה.
  + list, set, map מציעות איטרטור דו-כיווני.
  + וקטור מציע איטרטור גישה אקראית.
  + String ומערך רגיל מציעים איטרטור שהוא פוינטר, ועל כן גם הם מוגדרים כאיטרטור גישה אקראית.

## לולאות עם איטרטור

לדוגמה, מעבר בלולאה והדפסה של כל האיברים במחסנית באמצעות איטרטור:

for(Stack<int>::iterator i=myStack.begin(); i!=myStack.end(); i++)

cout << \*i << endl;

החל מ C++11, יש צורה קצרה יותר לכתוב את הלולאה עבור מיכלים שממשים את begin ו-end ואת האופרטורים '\*', '=', '=!', '++'. לולאות מסוג זה נקראות לולאות for each. אם נשתמש בהן ללא שהגדרנו את כל האופרטורים והשיטות הנצרכות, נקבל שגיאת קומפילציה.

for(int val : myStack)}

cout << val << endl;

}

## שיטות לקבלת איטרטור

בכל מיכל מובנים מספר שיטות המחזירות איטרטור למיקומים שונים במיכל, וכן שיטות המחזירות סוגי איטרטורים שונים. נתאר את השיטות המרכזיות שמחזירות איטרטור ובאילו מיכלים הן מוגדרות.

* לכל מיכל יש שיטה begin() המחזירה איטרטור לתחילת המיכל ושיטה end() המחזירה איטרטור לאיבר שאחרי האיבר האחרון במיכל. נשתמש בהן כדי לעבור באופן סדרתי על כל איברי המיכל.
* למיכלים המאפשרים הליכה אחורה (כולם, למעט זרמים ו-forward\_list) יש גם שיטה rbegin() המחזירה איטרטור לסוף המיכל ושיטה rend() המחזירה איטרטור לאיבר שלפני תחילת המיכל. נשתמש בהן כדי לעבור באיטרציה על כל איברי המיכל בסדר הפוך.
* לכל איטרטור במיקומים שהצגנו לעיל יש גירסה שהיא const, המאפשרת לקרוא את האיברים אך לא לשנות אותם. אפשר לגשת לאיטרטורים אלו ע"י השיטות: cbegin, cend, crbegin, crend. בסעיף הבא נפרט על סוג איטרטורים אלו יותר.

## const\_iterator

const\_iterator משמש במקרים שלא רוצים לשנות את המיכל אלא רק לעבור על כל האיברים ולבצע פעולה כלשהי. שימוש נוסף הוא כאשר אנו מקבלים מיכל קבוע ואנו רוצים לקבל איטרציה שלו. מכיוון שהמיכל קבוע לא נוכל להפעיל עליו שיטות begin() ו-end() המחזירות איטרטור רגיל, אלא נצטרך לקרוא לשיטות המחזירות const\_iterator.

המחלקות זהות לחלוטין, אלא ש-const\_iterator באופרטורים '\*' ו-'<-' מחזיר const של האובייקט אליו מצביע. מקובל שהשיטות שמחזירות const\_iterator התחלה וסוף הן cbegin ו-cend. לא ניתן להשתמש בלולאת for each ב-const\_iterator.

## איטרטורי הכנסה

ישנם כל מיני איטרטורים מיוחדים מסוג פלט (output iterator) שמקבלים כפרמטר מיכל. וכאשר מפעילים עליהם אופרטור השמה '=' עם אובייקט המתאים למיכל, יכניסו אובייקט זה למיכל שקיבלו בבנאי לפי הגדרתם. כאשר יש מספר איברים להכניס למיכל, מאוד מומלץ להשתמש באלגוריתם copy. יש שלושה סוגים של אופרטורי הכנסה.

### insert\_iterator

תבנית של איטרטור המקבלת מיכל ואיטרטור לאיבר כלשהו במיכל. כאשר נבצע על איטרטור זה השמה עם אובייקט המתאים למיכל, האיטרטור ישתמש בשיטת insert על המיכל ויכניס אותו במיקום של האיטרטור שהתקבל כפרמטר.

set<int> s;

auto itr = insert\_iterator(s, s.begin());

\*itr = 0;

vector<int> v{5,1,3};

copy(v.begin(), v.end(), insert\_iterator(s, s.begin())); //s = {5,1,3,0}

### back\_insert\_iterator

תבנית של איטרטור המקבלת מיכל בבנאי. כאשר נבצע על איטרטור זה השמה עם אובייקט המתאים למיכל, האיטרטור ישתמש בשיטת push\_back על המיכל, שיצור איבר חדש בסוף המיכל עם האובייקט שהוכנס. אם לא מוגדרת במיכל שהתקבל בבנאי שיטת push\_back, תתקבל שגיאת קומפילציה. איטרטור זה משמש בעיקר כאשר אנו לא יודעים מה גודל הקלט שצריך להיכנס למיכל.

vector<int> v1;

copy(s.begin(), s.end(), back\_insert\_iterator(v1)); //v1 = {5,1,3,0}

### front\_insert\_iterator

תבנית של איטרטור המקבלת מיכל. כאשר נבצע על איטרטור זה השמה עם אובייקט המתאים למיכל, האיטרטור ישתמש בשיטת front\_push להוספת האובייקט בראש המיכל. אם לא מוגדרת במיכל שהתקבל בבנאי שיטת front\_push תתקבל שגיאת קומפילציה.

vector<int> v2;

copy(s.begin(), s.end(), front\_insert\_iterator(v2)); //v2 = {0,3,1,5}

## איטרטורי זרימה

אלו הם איטרטורים המקושרים לערוץ פלט/קלט, ומאפשרים באמצעות פעולות אופרטורים עליהם לקרוא/לכתוב אליהם. יש שני סוגים של איטרטורי זרימה: ostream\_iterator עבור פלט ו-istream\_iterator עבור קלט.

### ostream\_iterator

זוהי תבנית של מחלקת איטרטור המקבלת בבנאי ערוץ פלט (כמו cout, קובץ טקסט, וכו'), ומחרוזת שתפריד בין ההדפסות של האיטרטור זרימה. באמצעות אופרטור '\*' ו'=' אפשר לכתוב דרך האיטרטור אל ערוץ הפלט.

ostream\_iterator<double> outItr(cout," - ");

for(double i=0; i<5; ++i) {

(\*outItr) = i; //will print "1 – 2 – 3 – 4"

outItr++;

}

שימוש נפוץ ב-ostream\_iterator הוא באלגוריתם copy, כאשר מכניסים לאלגוריתם טווח של איברים באמצעות איטרטורי התחלה וסוף של מיכל, ועוד ostream\_iterator שיכניס אוטומטית את כל האיברים בטווח זה אל ערוץ הפלט המקושר ל-ostream\_iterator.

set<int> s = {1,7,3,55};

copy(s.begin(), s.end(), ostream\_iterator<int>(cout, ", "));

### istream\_iterator

זוהי תבנית של מחלקת איטרטור המקבלת בבנאי ערוץ קלט (כמו cin, istringstream, קובץ טקסט, וכו'). הפעלת אופרטור '\*' על האיטרטור יגרום לו להחזיר את האיבר הבא בקלט. אם בונים istream\_iterator באמצעות בנאי ברירת מחדל, אזי האיטרטור יצביע לערך המציין את סוף הקלט. נשתמש באיטרטור ברירת מחדל זה בלולאה בתור תנאי העצירה.

vector<char> vec(4);

istringstream str("a b c d");

istream\_iterator<char> inItr(str);

istream\_iterator<char> end();

for(; inItr!=end; inItr++) {

vec.at(i) = \*inItr; //vec = {a,b,c,d);

}

שימוש נפוץ ב-istream\_iterator הוא באלגוריתם copy, כאשר מכניסים לאלגוריתם שני איטרטורים מסוג istream\_iterator, המציינים התחלה וסוף של קלט בערוץ קלט כלשהו. ובנוסף עוד איטרטור התחלה של מיכל כלשהו. האלגוריתם יכניס את כל האיברים מערוץ הקלט אל המיכל. אם ערוץ הקלט הוא cin, נקבל קוד שמאפשר למלא מיכל באמצעות הכנסת איברים מהטרמינל בזמן ריצה. במקרים אלו עדיף שהאיטרטור של המיכל יהיה איטרטור הכנסה למקרה שהקלט גדול מהמיכל.

vector<int> v;

copy(istream\_iterator<int>(cin),istream\_iterator<int>{},back\_insert\_iterator(v));

## הכנסה ומחיקה בעזרת איטרטורים

נראה כיצד ניתן להכניס ולמחוק איבר בכל סוג מיכל בעזרת איטרטורים.

### הכנסה

כדי להכניס איבר לכל סוג מיכל, נשתמש בשיטת insert או emplace, המוגדרות בכל מיכל ב-STL, ומקבלות איטרטור itr ואובייקט x, ומכניסות את x לפני itr. ההבדל בין השיטות הוא ש-emplace יכולה לקבל פרמטרים של הבנאי של האובייקט המוכל במיכל, ובכך לבנות את האובייקט בתוך המיכל ולמנוע העתקה מיותרת.

ב-insert נוכל גם להכניס קבוצת איברים על ידי שנכניס לשיטה איטרטור להתחלה ולסוף שלהם. הקבוצה תיכנס לפני האיטרטור של המיכל. אפשר גם להכניס קבוצת איברים באמצעות emplace, על ידי שנכניס את כל האובייקטים לשיטה.

set<int> s;

vector<int> v{1,3,5};

s.insert(s.begin(), 10); //s = {10}

s.insert(s.begin(), v.begin(), v.end()); //s = {1,3,5,10}

s.emplace(s.begin(), 2, 4); //s = {2,4,1,3,5,10}

### מחיקה

כדי למחוק איבר מכל סוג מיכל, נשתמש בשיטת erase, המוגדרת בכל מיכל ב-STL. erase מקבלת איטרטור ומוחקת את האיבר שעליו מצביע האיטרטור. לאחר מכן מעדכנת את האיטרטור שיצביע לאיבר הבא. ניתן להכניס ל-erase גם שני איטרטורים והיא תמחק את כל האיברים ביניהם, לא כולל האחרון.

במקרה שעוברים בלולאה על איברי המיכל, אם נמחק איבר ולאחר מכן נפעיל אופרטור '++', אזי במיכלים השמורים כגוש רציף בזיכרון נקבל שדילגנו על איבר, ובמיכלים אחרים יכול להיות שנקבל שגיאת segmentation fault. לכן חשוב מאוד להתנות שאם מוחקים איבר לא מקדמים את האיטרטור.

template<typename container, typename iterator>

void erase\_odd\_elements(container& c, iterator b, iterator e) }

for (; b!=e;) }

bool is\_odd = (\*b)%2 != 0;

if (is\_odd)

b = c.erase(b);

else

b++;

}

}

# פונקטורים - Functors

## הגדרה

פונקטור הוא אובייקט שיש לו אופרטור סוגריים. יש שלושה סוגים של פונקטורים:

1. מצביע לפונקציה.
2. מחלקה שיש לה אופרטור סוגריים. לסוג זה יש יתרון בכך שאפשר להגדיר בהם שדות שיעזרו לשכלל את החישוב בפונקטור. בנוסף אפשר לשלב בהם ירושה.
3. ביטוי למדא.

## פונקטורים מובנים

יש מספר תבניות של פונקטורים בספריה הסטנדרטית שמאוד נוח לעבוד איתם.

* less - זהו פונקטור שמקבל שני פרמטרים מסוג T, ומחזיר true אם הפרמטר הראשון קטן יותר מהשני לפי אופרטור "קטן מ-" המוגדר ב-T. אחרת, מחזיר false.
* greater - פונקטור שמקבל שני פרמטרים מסוג T, ומחזיר true אם הפרמטר הראשון גדול יותר מהשני לפי אופרטור "גדול מ-" המוגדר ב-T. אחרת, מחזיר false.
* מנוע מספרים אקראיים

# אלגוריתמים

## הגדרה

כל אלגוריתם בספריה התקנית אינו מקבל כקלט מיכלים, אלא זוג איטרטורים המייצגים טווח איברים במיכל, על ידי איטרטור להתחלה ואיטרטור לאחרי הסוף. כתוצאה מכך, כל אלגוריתם ממומש פעם אחת בלבד ואינו מותאם לסוג המיכל. בנוסף, האלגוריתמים אינם עובדים רק על מיכלים בספרייה התקנית, אלא על כל מבנה שמוגדר בו איטרטור. לכן הם יכולים לעבוד גם על מיכלים שאנו יצרנו, וכן יכולים לעבוד על מבנים שאינם מיכלים, כמו ערוצי זרימה.

ישנם מעל 100 אלגוריתמים בספרייה התקנית. כל מתכנת טוב צריך לדעת את כולם. הסיבה לכך היא שהאלגוריתמים פותרים בעיות מאד נפוצות בתכנות. ולכן אם לא נכיר את כולם, אנחנו עלולים לנסות לממש אותם בעצמנו בפרויקט אחר. ואז, אנחנו כנראה נממש אותם בחיפזון, בלי בדיקות, בצורה לא יעילה, ועם באגים. בנוסף, האלגוריתמים נכנסו לתקן של השפה, ולכן מתכנתים בכל העולם משתמשים בהם ומבינים אותם, ומצפים שגם מתכנתים חדשים יכירו אותם וישתמשו באותה שפה. כדי לזכור את כל האלגוריתמים נחלק אותם ל-7 קבוצות הגיוניות לפי אופי הפעולה שהם מבצעים.

## סוגי אלגוריתמים

### שאילתות (queries)

בקבוצת השאילתות נמצאים אלגוריתמים רבים המקבלים כקלט איטרטור התחלה ואיטרטור סוף, המייצגים טווח איברים במיכל, ומחזירים ערך כלשהו על הפרטים הנמצאים בטווח. כל אלגוריתמים מקבוצה זו אינה משנה שום דבר במיכל הנוכחי.

### אלגוריתמים על קבוצות

בקבוצה זו נמצאים אלגוריתמים המקבלים כקלט שתי קבוצות ומחזירים קבוצה שלישית. כל אחת מקבוצות הקלט מועברת על ידי שני איטרטורים - התחלה וסוף. תנאי מקדים לאלגוריתמים האלה הוא, שהאיברים בכל אחת מקבוצות הקלט מסודרים לפי אופרטור "קטן מ-" (<) או לפי פונקטור אחר המועבר כקלט. תנאי זה מתקיים עבור אסוציאטיביים מסודרים, אבל גם עבור וקטור או מערך שסידרנו אותו בעזרת sort. הפלט מיוצר בעזרת איטרטור נוסף - איטרטור פלט, המצביע לתחילת הקבוצה.

### פרמוטציות

בקבוצת הפרמוטציות נמצאים אלגוריתמים המקבלים כקלט טווח איברים (איטרטור התחלה סוף), ומשנים את סדר האיברים בסדרה הנמצאת בין האיטרטורים, אך אינם משנים את ערכי האיברים עצמם.

### מעבירים (movers)

בקבוצה זו נמצאים אלגוריתמים המעתיקים או מעבירים פרטים בין טווח אחד לטווח אחר.

### משני ערך (value modifiers)

בקבוצה זו נמצאים אלגוריתמים המשנים את הערכים בטווח נתון.

### משני מבנה - structure changers

בקבוצה זו נמצאים שני אלגוריתמים שמטרתם למחוק פרטים מתוך טווחים.

### מיליות (runes)

לרוב האלגוריתמים שראינו עד כה יש כמה גרסאות, שאפשר ליצור על ידי הוספות "מיליות" לשם האלגוריתם. לדוגמה, המילית is מציינת שאילתה: is\_sorted - האם הטווח מסודר, is\_sorted\_until - עד איפה בדיוק הטווח מסודר, וכו'. **אפשר לשלב בין מיליות שונות.**

## שאילתות מספריות

**count** - מקבל טווח איברים באמצעות איטרטורי התחלה וסוף, ואובייקט אותו אנו מעוניינים לחשב כמה פעמים נמצא במיכל. האלגוריתם מחזיר את כמות הפעמים שאובייקט זה נמצא בטווח האיברים. נוכל גם לספור כמה פעמים מחרוזת נמצאת בתוך קובץ טקסט, על ידי איטרטורי זרימה המקושר לקובץ.

vector<int> v{4,6,8,5,4,7};

count(v.begin(),v.end(),4) //returns 2

**accumulate** - מקבל טווח של איברים באמצעות איטרטורי התחלה וסוף, ומחזיר את הסכום של כל האיברים בטווח זה. בנוסף מקבל ערך התחלתי ממנו מתחיל לסכום. הפונקטור הברירת מחדל באלגוריתם הוא אופרטור חיבור, אמנם אנו יכולים להכניס פונקטור אחר שלפיו האיברים יסכמו, כמו כפל וחילוק.

vector<int> v{4,6,8,5};

accumulate(v.begin(),v.end(),0); //returns 23

accumulate(v.begin(),v.end(),1,[](int a,int b){return a\*b}); //returns 960

**partial\_sum** - מקבל טווח איברים ועוד איטרטור, ומכניסה סכומים חלקיים של כל האיברים מהמיכל הראשון לאיטרטור במיכל השני. גם כאן הפונקטור ברירת מחדל הוא אופרטור חיבור, אמנם ניתן להכניס פקטור אחר שלפיו האיברים יסכמו.

vector<int> v{4,6,8,5};

vector<int> v2;

partial\_sum(v.begin(),v.end(),back\_insert\_iterator(v2)); //v2={4,10,18,23}

**adjacent\_difference** - זוהי פעולה הפוכה של partial\_sum. מקבל את אותו קלט ומכניס לאיטרטור במיכל השני את ההפרש בין כל שני איברים סמוכים. אם נמלא מיכל באמצעות partial\_sum ולאחר מכן נפעיל עליו adjacent\_difference, נקבל את המיכל המקורי ממנו התחלנו.

vector<int> v{4,10,18,23};

vector<int> v2{v.size()};

adjacent\_difference(v.begin(),v.end(),v2.begin()); //v2={4,6,8,5}

**inner\_product -** מקבל טווח איברים של שני מיכלים ועוד מספר התחלתי ממנו מתחיל לסכום, ומחזיר את המכפלה הסקלרית בין שני המיכלים. האלגוריתם מתחשב רק בכמות האיברים של המיכל הראשון שהוכנס. הוא מניח שגם כמות האיברים במיכל השני באותו גדול, ולכן מספיק להכניס לאלגוריתם איטרטור התחלה למיכל השני, אך ניתן גם להכניס איטרטור לסוף המיכל.

vector<int> v1{1,2,3,4};

vector<int> v2{5,6,7,8};

inner\_product(v1.begin(),v1.end(),v2.begin(),0); //returns 1\*5+2\*6+3\*7+4\*8

## שאילתות תכונות

אלו אלגוריתמים שמקבלים טווח של איברים באמצעות איטרטורי התחלה וסוף, ועוד פונקטור בוליאני שיופעל על כל איבר בטווח. האלגוריתם יחזירו true או false בהתאם לתכונה שהם מייצגים.

**all\_of** - מחזיר true אם כל האיברים מקיימים את התנאי.

**any\_of** - מחזיר true אם לפחות אחד מהאיברים מקיים את התנאי.

**none\_of** - מחזיר true רק אם אף אחד מהאיברים מקיים את התנאי.

vector<int> v{4,6,8,5,4,1,3,1,4,7};

all\_of(v.begin(),v.end(), [](int x){return x<10;}) //true

any\_of(v.begin(),v.end(), [](int x){return x>10;}) //false

none\_of(v.begin(),v.end(), [](int x){return x<8;}) //false

## שאילתות חיפוש

אלגוריתמים אלו מקבלים טווח של איברים באמצעות איטרטורי התחלה וסוף, ועוד איבר או מספר איברים שאותם אנו מחפשים בטווח. אם האיבר שאנו מחפשים נמצא בטווח, הם יחזירו איטרטור למיקום איבר זה, אחרת יחזירו איטרטור המייצג את סוף הטווח שהתקבל בקלט.

**find** - כמתואר לעיל, מקבלים טווח ואובייקט לחיפוש, ומחזירים איטרטור למיקום הראשון של אובייקט זה בטווח. אם האובייקט לא נמצא בטווח יחזירו איטרטור לסוף הטווח.

vector<int> v{4,2,8,5,2,0,1,1,9,7,5,2,0};

auto find8 = find(v.begin(), v.end(), 8); //find8-v.begin=2

**binary\_search** - כמו find אלא שכאן החיפוש אינו סדרתי אלא בינארי.

auto bs8 = binary\_search(v.begin(), v.end(), 8); //bs8-v.begin=2

**adjacent\_find** - מקבל טווח של איברים, ובודק האם בטווח קיימים שני איברים סמוכים זהים. אם כן, יחזיר איטרטור לאיבר הראשון הזהה, אחרת יחזיר איטרטור לסוף הטווח שהתקבל בקלט.

vector<int> v{4,2,8,5,2,0,1,1,9,7,5,2,0};

auto af = adjacent\_find(v.begin(), v.end()) ;//af-v.begin()=6

**search** - מקבל טווח של איברים ועוד טווח קטן יותר, ובודק אם הטווח השני מוכל בתוך הטווח הראשון. אם כן, מחזיר איטרטור לאיבר הראשון בטווח השני.

vector<int> v{4,2,8,5,2,0,1,1,9,7,5,2,0};

array<int,3> a{5,2,0}; //search520-v.begin()=3;

auto search520 = search(v.begin(), v.end(), a.begin(), a.end());

**find\_end** - כמו search רק שמתחיל לחפש מהסוף להתחלה.

auto findend = find\_end(v.begin(), v.end(), a.begin(), a.end()); //findend-v.begin()=10

**min\_element** - מקבל טווח של איברים ומחזיר איטרטור לאיבר הכי קטן לפי הגדרת אופרטור "קטן מ-" בהגדרת מחלקת האובייקט. ניתן גם להכניס פונקטור שלפיו יתבצע החישוב מי הכי קטן.

auto minelement\_itt = min\_element(begin(v), end(v));

**max\_element** - מקבל טווח של איברים ומחזיר איטרטור לאיבר הכי גדול לפי הגדרת אופרטור "גדול מ-" בהגדרת מחלקת האובייקט. ניתן גם להכניס פונקטור שלפיו יתבצע החישוב מי הכי גדול.

auto maxelement\_itr = max\_element(begin(v), end(v));

**minmax\_element** - מקבל טווח של איברים ומחזיר pair של זוג איטרטורים. הראשון מצביע לאיבר המינימלי והשני מצביע לאיבר המקסימלי.

auto [min1,max1] = minmax\_element(begin(v), end(v));

**equal\_range** - מקבל טווח של איברים ממוין בסדר עולה ועוד אובייקט. האלגוריתם יחזיר pair של איטרטורים המצביעים להתחלה וסוף של התת-טווח שבו מופיע אובייקט זה. אם אובייקט זה אינו מופיע כלל בטווח שני האיטרטורים שיחזיר יהיו זהים ויצביעו למיקום בטווח שבו צריך להוסיף את האובייקט.

vector<int> v{0,1,1,1,2,6,7};

auto [first1,last1] = equal\_range(v.begin(), v.end(), 1);

//first1-v.begin()=1, last1-v.begin()=4

auto [first3,last3] = equal\_range(v.begin(), v.end(), 3);

//first3-v.begin()=5, last3-v.begin()=5

## אלגוריתמים על קבוצות

**set\_union** - מקבל שתי קבוצות של איברים ממוינים, ואיטרטור התחלה לקבוצה שלישית. סך הכל חמישה איטרטורים. האלגוריתם יבצע פעולת **איחוד** בין שתי הקבוצות, ויכניס את כל האיברים בצורה ממוינת לקבוצה השלישית החל ממיקום האיטרטור החמישי. אם יש שני איברים זהים בשני הקבוצות הם יופיעו פעם אחת בקבוצה השלישית. אלגוריתם זה מייצג את פעולת על שתי קבוצות.

vector<int> v{10,1,1,2,3,4,4,5,6,7,8,9};

set<int> v{22,4,6,8,10,12,14,16,18};

vector<int> v3;

set\_union(begin(v1),end(v1),begin(v2),end(v2),back\_inserter(v3));

**set\_intersection** - מקבל שתי קבוצות של איברים ממוינים, ואיטרטור התחלה לקבוצה שלישית. סך הכל חמישה איטרטורים. האלגוריתם יבצע פעולת **חיתוך** בין שתי הקבוצות, ויכניס את כל האיברים בצורה ממוינת לקבוצה השלישית החל ממיקום האיטרטור החמישי. אלגוריתם זה מייצג את פעולת על שתי קבוצות.

set\_intersection(begin(v1),end(v1),begin(v2),end(v2),back\_inserter(v3));

**set\_difference** - מקבל שתי קבוצות של איברים ממוינים, ואיטרטור התחלה לקבוצה שלישית. סך הכל חמישה איטרטורים. האלגוריתם יבצע פעולת **חיסור** בין הקבוצה הראשונה לשנייה, כלומר ייקח רק את האיברים שיש בראשונה שאין בשנייה, ויכניס את כל האיברים בצורה ממוינת לקבוצה השלישית החל ממיקום האיטרטור החמישי. נשים לב שהאלגוריתם לא אסוציאטיבי, כלומר v1-v2 שונה מ-v2-v1.

set\_difference(begin(v1),end(v1),begin(v2),end(v2),back\_inserter(v3));

**set\_symmetric\_difference**- מקבל שתי קבוצות של איברים ממוינים, ואיטרטור התחלה לקבוצה שלישית. סך הכל חמישה איטרטורים. האלגוריתם ייקח את כל האיברים שיש רק באחת הקבוצות ולא בשניהם, ויכניס את כל האיברים בצורה ממוינת לקבוצה השלישית החל ממיקום האיטרטור החמישי. אלגוריתם זה מייצג את פעולת על שתי קבוצות.

set\_symmetric\_difference(begin(v2),end(v2),begin(v1),end(v1),back\_inserter(v3));

**includes** - מקבל שני קבוצות של איברים ממוינים, סך הכל 4 איטרטורים. האלגוריתם יחזיר true אם הקבוצה הראשונה מכיל את השנייה, אחרת יחזיר false.

includes(begin(v1), end(v1), begin(v2), end(v2))

## פרמוטציות

מטרת אלגוריתמים אלו לעבור על כל הפרמוטציות של סדר האיברים. האלגוריתמים יקבלו קבוצת איברים (איטרטור התחלה וסוף), ויסדרו את איברי הקבוצה כך שתתאים לפרמוטציה הבאה. סדר הפרמוטציות מוגדר באופן לקסיקוגרפי. כדי לעבור על כל האיברים נשתמש בלולאת do while. האלגוריתם יחזיר true כל עוד לא הגענו לפרמוטציה האחרונה, אחרת יחזיר false כדי להפסיק את הלולאה.

**next\_permutation** - מקבל קבוצת איברים באמצעות איטרטור התחלה וסוף, ומסדר את איברי הקבוצה כך שתתאים לפרמוטציה הבאה. כדי שנעבור בלולאה על כל הפרמוטציות חשוב שהסידור הראשון שממנו מתחילים יהיה ממוין מהקטן לגדול לפי אלגוריתם sort.

string v {"abcd"};

do {

//do something

{while (next\_permutation(v.begin(),v.end());

**prev\_permutation** - כמו האלגוריתם הקודם אלא שמחזירה פרמוטציה קודמת לפי סדר לקסיקוגרפי. כדי שנעבור בלולאה על כל הפרמוטציות חשוב שהסידור הראשון שממנו מתחילים יהיה מהגדול לקטן.

string v {"dcba"};

do {

//do something

{while (prev\_permutation(v.begin(),v.end());

## פרמוטציות סידור

אלגוריתמים שמטרתם לסדר איברים של מיכלים. בכל האלגוריתמים ברירת המחדל היא למיין לפי אופרטור "קטן מ", אך אפשר להכניס פונקטור בינארי משלנו שמקבל שני איברים ויחזיר true אם האיבר הראשון צריך להיות לפני האיבר השני. אחרת, אם האיבר השני צריך להיות לפני הראשון, מחזיר false.

**sort** - מקבל קבוצת איברים באמצעות איטרטור התחלה וסוף, וממיין אותם. אלגוריתם זה פועל רק על מיכלים שמוגדר בהם איטרטור גישה אקראית (vector, deque, string, array וכו'). הסיבה לכך היא שאחרת המיון לא יהיה יעיל.

vector<int> v{19,18,16,17,13,12,10,14,15,,11};

sort(begin(v), end(v)); // v=10,11,12,13,14,15,16,17,18,19

sort(begin(v), end(v), [](int a, int b){return a>b;}); //big to small

**nth-element** - מקבל קבוצת איברים ועוד מצביע לאיבר כלשהו במקום ה-n בקבוצה, באמצעות שלושה איטרטורים: התחלה, אמצע, סוף. האלגוריתם ממיין את הקבוצה כך שהאיבר במקום ה-n יהיה במקום המתאים לו אילו מיינו את הקבוצה כולה. בצד שמאל של האיבר יהיו כל האיברים שקטנים ממנו, ובצד ימין כל האיברים שגדולים ממנו, לא בהכרח ממוינים. אלגוריתם זה שימושי כאשר רוצים למצוא רוצים למצוא רק את האיבר ה-n בקוטנו בצורה מהירה. דוגמה נפוצה לשימוש באלגוריתם זה הוא במיון מהיר.

nth\_element(begin(v), begin(v)+5, end(v)); //14,10,11,12,13,**15**,16,19,17,18

**partial\_sort** - כמו nth\_element אלא שבנוסף דואג שעד האיבר ה-n כל האיברים יהיו במקום המתאים להם כמו מיון רגיל על הקבוצה כולה. מהאיבר ה-n והלאה לא בהכרח ממוין.

partial\_sort(begin(v), begin(v)+5, end(v)) //**10,11,12,13,14,15**,16,19,17,18;

**reverse** - מקבל קבוצת איברים באמצעות איטרטור התחלה וסוף. האלגוריתם והופך את סדר האיברים.

reverse(begin(v), end(v));

**inplace\_merge** - מקבל קבוצת איברים המחולקת לשניים וממזג שני חלקים אלו לקבוצה ממוינת. האלגוריתם מקבל שלושה איטרטורים: התחלה, איטרטור להיכן שהקבוצות מחולקות, ואיטרטור לסוף. דוגמה נפוצה לשימוש באלגוריתם זה הוא במיון מיזוג.

array<int,10> a{11,13,15,17,19, 12,14,16,18,20};

inplace\_merge(begin(a), begin(a)+5, end(a)); //11-20 sorted

## פרמוטציות ערבוב

**shuffle** - מקבל קבוצת איברים, באמצעות איטרטור התחלה וסוף, ועוד מנוע ערבול אליו צריך להכניס מספר הנקרא "זרע". האלגוריתם מערבב את סדר האיברים בקבוצה לפי המספר שהוכנס למנוע הערבול. הפעלת האלגוריתם עם זרע זהה תניב את אותו ערבוב, אך עבור זרע שונה נקבל ערבוב אחר.

shuffle(begin(v), end(v), default\_random\_engine{1});

**rotate** - מקבל קבוצת איברים ועוד איטרטור למיקום כלשהו בקבוצה. האלגוריתם יחלק את האיברים לשניים: אלו שמימין לאיבר אמצע, כולל עצמו, ואילו שמשמאלו. לאחר מכן יחליף בין שני קבוצות אלו, כך שכעת האיבר אמצע יהיה ראשון בקבוצה.

vector<int> v{9,8,6,7,4,5,2};

rotate(begin(v), begin(v)+3, end(v)); // v = 7,4,5,2,9,8,6

**partition** - מקבל קבוצת איברים ועוד פונקטור בוליאני. האלגוריתם ימקם את כל האיברים שמקיימים את התנאי בפונקטור ראשונים ולאחריהם כל האיברים שאינם מקיימים את התנאי. האיברים בכל קבוצה אינם ממוינים. האלגוריתם מחזיר איטרטור לאיבר הראשון בסדר האיברים שאינו מקיים את התנאי. ניתן לשלב אלגוריתם זה עם אלגוריתמים sort ו-copy, כדי ליצור ולמיין קבוצת איברים שמקיימים את התנאי.

vector<int> v{9,8,6,7,4,5,2,0,3,1};

auto middle = partition(begin(v),end(v),[](int x){return x%2==0;});//v = 6,4,2,0,8,1,5,3,9,7

**partition\_point** - מקבל קבוצה שבוצע עליה partition, ומחזיר איטרטור לאיבר אמצע הראשון שלא מקיים את התנאי. שימושי למקרה שחייבים לשנות את האיטרטור שהוחזר מ-partition, ורוצים לקבל אותו שוב..

auto middle2 = partition\_point(begin(v), end(v), [](int x){return x%2==0;});

## מעבירים

**copy** - מקבל קבוצת איברים ועוד איטרטור פלט (output iterator) למיכל נוסף. האלגוריתם יעתיק את כל האיברים מהמיכל הראשון לאיטרטור של המיכל השני. צריך לשים לב שאכן ניתן להכניס את כל האיברים במיכל הראשון למיכל השני. כלומר, שהמיכל השני מספיק גדול להכיל את האיברים שמכניסים אליו. אם לא ידוע מספר האיברים צריך להשתמש באיטרטורי הכנסה במיכל השני.

vector<int> v{0,1,2,3,4,5,6,7,8,9};

vector<int> v2;

copy(v.begin(), v.end(), back\_insert\_iterator(v2));

ניתן להשתמש באלגוריתם זה גם עבור איטרטורי זרימה. כך נוכל בדרך מאוד נוחה לקלוט מערוץ קלט אובייקטים לתוך מיכל, וכן נוכל לפלוט אובייקטים לתוך ערוץ פלט.

vector<int> v;

copy(istream\_iterator<int>(cin),istream\_iterator<int>{},back\_insert\_iterator(v));

set<int> s = {1,7,3,55};

copy(s.begin(), s.end(), ostream\_iterator<int>(cout, ", "));

**copy\_backwards** - כמו copy, אלא שבאיטרטור של הקבוצה השנייה שאליה מעתיקים, האלגוריתם לא יקבל איטרטור התחלה שממנו יתחיל להעתיק, אלא יקבל איטרטור סוף שיעתיק עד אליו.

copy\_backward(begin(v), begin(v)+4, begin(v2)+7);

**swap\_ranges** - מקבל שני קבוצות איברים, איטרטור התחלה וסוף של הקבוצה הראשונה ועוד איטרטור התחלה של הקבוצה השנייה. האלגוריתם מניח ששניהם באותו גודל. האלגוריתם יחליף את האיברים בין שני הקבוצות. אם קבוצה אחת גדולה יותר, אזי היא תקבל את כל האיברים של הקבוצה הקטנה, ועוד ערכי זבל.

swap\_ranges(begin(v), end(v), begin(v2))

**sample** - מאפשר לדגום באופן אקראי מספר איברים קבוע מקבוצת איברים ולשים אותם בקבוצה שנייה. האלגוריתם מקבל איטרטור התחלה וסוף של הקבוצה הראשונה, איטרטור התחלה של הקבוצה השנייה שממנו ימקם את האיברים, מספר איברים לדגום, ומנוע ערבול עם זרע שלפיו ידגום את האיברים מהקבוצה.

sample(v.begin(),v.end(),back\_inserter(cout), 5, default\_random\_engine(1));

## משני ערך

**fill** - מקבל קבוצת איברים, באמצעות איטרטור התחלה וסוף, ועוד ערך כלשהו. האלגוריתם ישנה את כל האיברים בטווח אל הערך שהתקבל.

vector<char> v(4);

fill(begin(v), end(v), 'b'); //v = b, b, b, b

**iota** - מקבל קבוצת איברים, ועוד ערך כלשהו val. האלגוריתם יאתחל את האיבר הראשון ב-val. עבור כל איבר לאחר מכן, קודם יעדכן את val באמצעות אופרטור ++, ולאחר מכן יעדכן את האיבר לערך שהתקבל לאחר הקידום.

iota(begin(v), end(v), 'b'); //v = b, c, d, e

**generate** - כמו fill, אלא שבמקום ערך מקבל פונקטור. עבור כל איבר האלגוריתם יפעיל את הפונקטור ויכניס את הפלט לתוך האיבר. אם הפונקטור הוא מחלקה עם שדות או פונקציה עם משתנה סטטי, נוכל למלא את הקבוצה בערכים שונים לפי הגדרת הפונקטור. אם נרצה להפעיל פונקטור שמקבל ערך, לא נוכל לעשות זאת עם פונקטור שהוא פונקציה רגילה, אלא נוכל לעשות זאת באחת משני דרכים:

* הפונקטור מסוג מחלקה. ואז נוכל להכניס לתוך הפונקטור גם ערך התלוי באיבר.
* נשתמש בפונקציה bind המקבלת פונקטור וערך x, ומחזירה פונקטור חדש שהוא בעצם הפונקטור הקודם שמקבל את x כפרמטר.

generate(v.begin(), v.end(), powers\_of\_two);

generate(v.begin(), v.end(), bind(powers\_of\_k, 3));

**replace** - מקבל קבוצת איברים ושני ערכים x ו-y. האלגוריתם יחליף את כל איברים x באיברי y.

vector<int> v {1,8,4,1};

replace(begin(v), end(v), 1, 3); // v = 3,8,4,3

## משני מבנה

**remove** - מקבל טווח וערך, ו"מוחק" את כל האיברים בטווח השווים לערך הנתון. האלגוריתם לא באמת יכול למחוק אותם, שהרי הוא הרי לא מקבל רפרנס לאוסף, אלא רק שני איטרטורים. במקום הוא מעביר כל האיברים שאמורים להישאר בטווח (שאינם שווים לערך הנמחק) לתחילת הטווח, ואז מחזיר איטרטור איבר שממנו עד סוף הטווח צריך למחוק. לאחר מכן, אפשר להשתמש בשיטה אחרת של האוסף (למשל vector::erase) כדי לבצע את המחיקה עצמה.

vector<int> v{11,8,4,6,11,9,3,5,11,6};

auto new\_end = remove(begin(v), end(v), 11);

v.erase(new\_end, end(v));

**unique** - מטרתו למחוק את כל האיברים הכפולים. פועל בדומה ל-remove. מקבל טווח איברים, ומזיז את כל האיברים הכפולים לסוף, ומחזיר איטרטור לאיבר הראשון שצריך למחוק.

vector<int> v {11,8,5,6,11,11,11,9,3,5,5,11,6};

auto new\_end = unique(begin(v), end(v));

v.erase(new\_end, end(v));

## מיליות

אלו מילים שאם נוסיף אותם לאלגוריתמים מסוימים יתקבל אלגוריתם שונה בהתאם למשעות המילה.

**stable\_x** - נוסיפם לאלגוריתמי מיון שמסדרים רק חלק מהאיברים. הוספת מילה זו יוצרת אלגוריתם שבכל איברים אלו שומר על ההסדר ההתחלתי. לדוגמה: stable\_partition.

**x\_If**- יוצרת אלגוריתם שאפשר להכניס לו פונקטור בוליאני, ופעולת האלגוריתם המקורי תתבצע רק עבור איברים המקיימים את התנאי. לדוגמה: remove\_if, copy\_if, replace\_if.

**is\_x** - יוצרת אלגוריתם שמחזיר ערך בוליאני לשאלה האם האלגוריתם המקורי התבצע. לדוגמה: is\_sorted.

**Is\_x\_until** - כמו is אלא שבודק האם התנאי מתקיים, ומחזיר איטרטור לאיבר הראשון שלא מקיים את התנאי. לדוגמה: is\_sorted\_until.

**x\_copy** - יוצר אלגוריתם שמצבע את פעולת האלגוריתם המקורי ומעתיק את התוצאה אתל קבוצה אחרת. האלגוריתם לא משנה את הקבוצה המקורית שהתקבלה. לדוגמה: partition\_copy, reverse\_copy.

## אלגוריתמים נוספים

**for\_each** - מקבל קבוצת איברים באמצעות איטרטור התחלה וסוף, ועוד פונקטור. האלגוריתם מפעיל את הפונקטור על כל אחד מהאיברים באופן סדרתי.

for\_each(v.begin(), v.end(), print); //print all elements

for\_each(v.begin(), v.end(), [](int &n){ n++; }); //promote all elements by 1

**transform** - מקבל טווח איברים, איטרטור למיכל נוסף, ופונקטור. האלגוריתם יפעיל על כל איבר בטווח את הפונקטור ויכניס את הפלט לאיטרטור של המיכל הנוסף. גרסה נוספת היא שמקבל שני טווחים, איטרטור נוסף, ופונקטור בינארי. האלגוריתם יפעיל את הפונקטור על איבר מכל טווח ויכניס את הפלט לאיטרטור הנוסף.

string s("hello");

vector<int> ordinals; //insert the ASCII value of the string to this vector

transform(s.begin(),s.end(),back\_inserter(ordinals),[](unsigned char c){return c;});

vector<int> v1{1,2,3,4,5,6};

vector<int> v2{6,5,4,3,2,1};

vector<int> v3;

transform(begin(v1),end(v1),begin(v2),back\_inserter(v3),[](int x,int y){return x\*y;});