הספרייה הסטנדרטית של C++

מהי הספרייה הסטנדרטית ?
הספרייה הסטנדרטית היא אוסף קבצים המגדירים שירותים שונים בהם ניתן להשתמש מתוך התוכנית. השירותים הללו אמורים להיות מסופקים ע"י כל ספק של מהדר C++ ולכן קוד המשתמש בספריה הסטנדרטית אמור לעבור קומפילציה בצורה תקינה גם כאשר הוא מועבר בין פלטפורמות שונות.

מהם השיקולים לבחירת השירותים הכלולים בספריה סטנדרטית ?
ספריה סטנדרטית אמורה לכלול שירותים שימושיים לחלק גדול מהמימושים. הרעיון הוא לא לכלול כלים אזוטריים המתאימים לצרכים מיוחדים, אלא לחפש דווקא את המכנה המשותף לצרכים של רב המתכנתים. מכיוון שספרייה סטנדרטית צריכה להיות ממומשת בכל מהדר של השפה, אסור לה להיות גדולה מדי. ספריה סטנדרטית טובה צריכה, אם כן, לספק את אוסף השירותים הכלליים השימושיים ולא יותר מזה. במידה ויש צורך בשירותים מיוחדים, על המתכנת להתקין ספרית שירותים מיוחדת שאיננה חלק מהסטנדרט.

בספריה הסטנדרטית של C++, הושם דגש מיוחד על יעילות. למרות שלא לכל המימושים היעילות היא קריטית, הספרייה הסטנדרטית חייבת לספק שירותים יעילים. ניתן לבנות קוד לא יעיל ע"י שימוש בקוד יעיל אך את ההפך לא ניתן לעשות. אם הספרייה הסטנדרטית הייתה נותנת שירותים לא יעילים, לא היה ניתן לכתוב בעזרתה קודים יעילים, והשימושיות שלה הייתה נפגעת.
הספרייה הסטנדרטית בנויה כך שהיא מכוונת את המשתמש לעשות בה שימוש יעיל.

מה כוללת הספרייה הסטנדרטית של C++ ?
השירותים הכלולים בספריה הסטנדרטית הם:

1. מספר מבני נתונים בסיסיים (containers) והאיטרטורים שלהם.
2. מספר אלגוריתמים כלליים המנוסחים במושגים של איטרטורים.
3. שירותי קלט\פלט.
4. "עטיפות" למבני נתונים (Adaptors).
5. שירותים להקצעת זיכרון.
6. שירותים לחישובים נומריים.
7. שירותים של מחרוזות (strings).
8. שירותים נוספים: הגדרות Exceptions, תמיכה בשפה, localizations.

הערה: החלק של מבני הנתונים, האיטרטורים והאובייקטים להקצאת זיכרון, פותח בהתחלה כספריה נפרדת שנקראה standard templates library – STL. מאוחר יותר אומצה הספרייה והייתה חלק מהספרייה הסטנדרטית.

קבצי ה – header של הספרייה הסטנדרטית
ה – headers של הספרייה הסטנדרטית הם קבצים ללא extension. לדוגמה, כאשר נרצה לכלל את ה – header של זרמי ה – קלט\פלט, נכתוב:

```
#include <iostream>
```

אם נכתוב, כפי שהיינו רגילים:

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

יכלול ה – preprocessor את הקובץ iostream.h אשר איננו חלק מהספרייה הסטנדרטית החדשה אלא קובץ השייך לגרסאות מוקדמות יותר של השפה והושאר למטרות תאימות. ההגדרות בקובץ istream.h שונות מההגדרות בקובץ iostream של הספרייה הסטנדרטית.
קבצי header אחדים מ – c זכו אף הם להיכלל בספריה הסטנדרטית, שמם נותר כשהיה רק בתוספת האות c בתחילתם. לדוגמה: <cstdio>, <cstdlib>, <ctime>, <cstring> ועד.

מרחב השם std
כל המחלקות, האובייקטים והפונקציות של הספרייה הסטנדרטית, נמצאים בתוך מרחב שם הנקרא std. התוכנית 'hello world' המשתמשת בספריה הסטנדרטית, תיראה לכן כך:

```
#include <iostream>
int main() {
```

```
	std::cout << "hello world!"<<std::endl;
}
```

או:

```
#include <iostream>
using namespace std;
int main() { // write the same way we are used to
	cout << "hello world!\n";
}
```

קוד כזה מבטל למעשה את מרחב השם std ביחידת הקומפילציה הזאת. הדבר עלול לגרום להתנגשות שמות ולכן בדרך כלל איננו מומלץ. אפשר לבטל את מעטפת השם רק עבור אלמנטים מסוימים:

```
using std::cout;
```

המחלקה string

string היא מחלקה המייצגת מחרוזת של תווי char. המחלקה תומכת בצורה נוחה בפעולות הבסיסיות של עבודה עם מחרוזות. לדוגמה:

```
#include <iostream>
#include <string>
using namespace std;
int main() {
	string str("hi dude!");
	string str1 = "z" + str;
	str += "\n what's up ?\n";
	if (str1 < str)
		cout << str1;
	else
		cout << str;
	str[2] = 'g';
	// …

}
```

מבני הנתונים הבסיסיים:

כל מבני הנתונים הם templates של טיפוסים כלליים, הם מכונים containers כיוון שחושבים עליהם ככלים לאחסנת אובייקטים. ה - containers הכלולים בספריה הסטנדרטית הם:

vector – מערך גמיש של טיפוס כללי. פניה לאבר כללי בזמן קבוע והוספת אבר בזמן ממוצע השואף לקבוע.

list – רשימה משורשרת דו כיוונית של טיפוסים כלליים.

deque - תור בעל שני קצוות. פעולות על הקצוות אמורות להיות יעילות כמו ברשימה וגישה לאבר שרירותי אמורה להיות יעילה כמו ב vector.

queue – תור.

priority_queue – תור עדיפויות (נימצא בתוך ה – header: <queue>).

stack – מחסנית.

map – מערך אסוציאטיבי בעל מופע יחיד של כל אלמנט המאוכסן בו.

multimap – מערך אסוציאטיבי עם אפשרות לריבוי מופעים של אלמנט מאוכסן. (מופיע ב - <map>).

set – קבוצה של אברים, מופע יחיד לכל אבר.

multiset – קבוצה עם ריבוי מופעים (נימצא ב - <set>).

bitset – מערך של ערכים בולאניים.

חלק ממבני הנתונים שתוארו (כמו stack ו queue) הם בעצם עטיפות או ממשקים המשתמשים במבני נתונים אחרים. "עטיפות" אלה נקראות "Adaptors" כיוון שהן לוקחות מבנה נתונים מסוים ומתאימות אותו לפעול כמבנה נתונים אחר. אפשר להגדיר מחסנית שתשתמש ב vector, deque או list. ה – adaptors לא מספקים איטרטורים כיוון שהכוונה היא שישתמשו בהם רק דרך הממשק הנוקשה שהם מגדירים.

מבני הנתונים השונים שומרים על ממשק דומה במידת האפשר. לדוגמה, פונקציה בשם push_back שמשמעותה – דחיפת אבר חדש בסוף מבנה נתונים, תהיה ממומשת באותו השם הן ל – vector הן ל – list והן ל – deque:

```
#include <iostream>
#include <vector>
#include <list>
#include <deque>

int main() {
	std::vector<int> v;
	std::list<double> l;
	std::deque<float> d;
	int i;
	for(i = 0; i<20; i++){
		v.push_back(i);
		l.push_back(i);
		d.push_back(i);

	}
	for(i = 0; i<20; i++) {
		v[i]+=20;
		std::cout << v[i] << ",";
	}
}
```

הרשימה הבאה היא רשימה חלקית של שירותים הניתנים ע"י מבני נתונים:

front() - האבר הראשון.
back() – האבר האחרון.
operator[] – subscripting: פניה לאבר כללי. (לא ממומש ל – list).
at() – פניה לאבר כללי עם בדיקת חריגה. (לא ממומש ל – list).
push_back() – הוספה לסוף.
pop_back() – הסרת האבר האחרון (ללא החזרתו).
push_front() – הוספה להתחלה.
pop_front() – הסרה מההתחלה.
clear() – ריקון מבנה הנתונים.
size() – החזרת מספר האברים המאוכסנים.
empty() – האם המבנה ריק?

שימו לב ש list איננה תומכת ב – subscripting כיוון שאיננה יכולה לממש זאת באופן יעיל.
הממשק האחיד מאפשר לכתוב פונקציות ל – container כללי. אפשר למשל להוסיף את הקוד הבא לדוגמה האחרונה:

```
template <class T,class Container>
void push_many(Container& c, T& val, int n) {
	for(int i=0; i<n; i++)
```

```
        c.push_back(val);
}
int main() {
//…
    push_many(v,7,10);
    push_many(l,8,20);
    push_many(d,9,30);
}
```

איטרטורים

כל מבני הנתונים מלבד ה – Adaptors מספקים איטרטורים. השירותים הניתנים ע״י האיטרטורים תלויים במבנה הנתונים עליהם הם פועלים. באופן כללי, איטרטור נותן את כל השירותים שניתן לספק **בצורה יעילה**. לדוגמה, איטרטור של list יאפשר את קידומו לאבר הבא (++operator) , את החזרתו לאבר הקודם (--operator) אך לא יאפשר הזזה שלו במספר שרירותי של אברים (+ operator), כיוון שפעולה זו עלולה להיות יקרה. לאיטרטור של vector לעומת זאת, כן ימומשו האופרטורים הבינאריים '+' ו '-', כיוון שניתן לקדם איטרטור שלו במספר שרירותי של אברים בצורה יעילה.

האיטרטורים בספריה הסטנדרטית מתחלקים לקטגוריות הבאות:

output iterator - איטרטור המאפשר מעבר סידרתי רק לכתיבה. האופרטורים הממומשים: ++ ו - * לכתיבה.

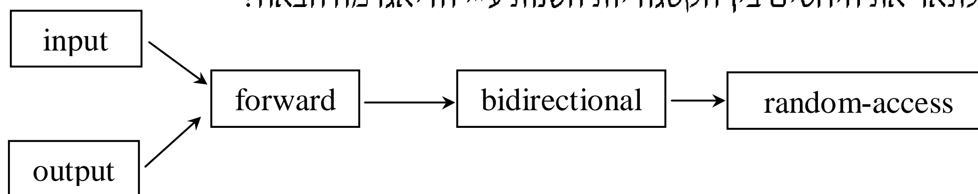
input iterator – איטרטור המאפשר מעבר סידרתי רק לקריאה. ממש את האופרטורים: ++ ==, =! ואת * ,<- לקריאה בלבד.

forward iterator – איטרטור המאפשר כתיבה וקריאה במעבר סידרתי קדימה. אופרטורים: ++, ==, =!, ו- *,<- לקריאה וכתיבה.

bidirectional iterator – איטרטור המאפשר כתיבה וקריאה תוך מעבר קדימה ואחורה. אופרטורים: כל אלה של **forward** ובנוסף --.

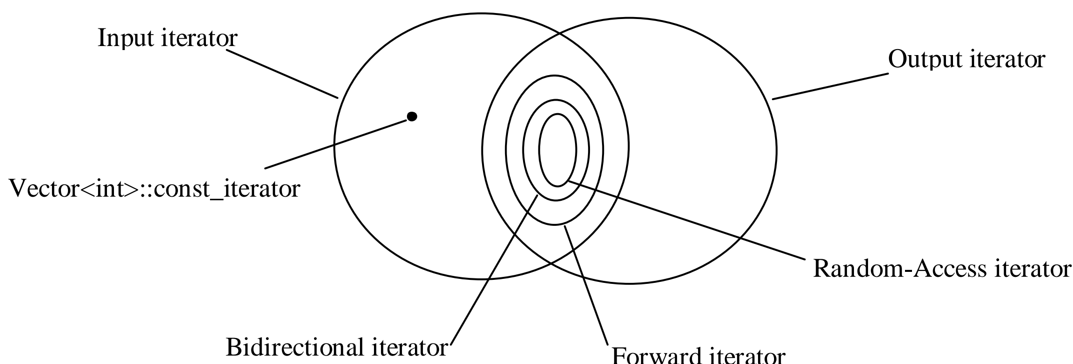
random-access iterator – איטרטור המאפשר כתיבה וקריאה תוך מעבר של קפיצות שרירותיות. אופרטורים: כל אלה של **bidirectional** ובנוסף: =+, =-, +, -, >, <, =>, =< ו [].

ניתן לתאר את היחסים בין הקטגוריות השונות ע״י הדיאגרמה הבאה:



חשוב לשים לב שהקטגוריות אינן מייצגות מחלקות עם יחסי ירושה ביניהן, כל איטרטור הוא מטיפוס שונה והחלוקה לקבוצות היא לפי השירותים שהוא מציע. בכל קטגוריה יכולים להיות מספר איטרטורים שאין שום קשר ביניהם.

נצייר Venn diagram של כלל האיטרטורים בספרייה:



מבני הנתונים מחזירים איטרטורים להתחלה ולאבר אחד אחרי הסוף כאשר מפעילים את הפונקציות begin() ו – end() בהתאמה. לכל מבנה נתונים יש מספר סוגי איטרטורים בסיסיים: iterator, const_iterator ו- reverse_iterator. הראשון יכל לאפשר שינוי של מבנה הנתונים, השני רק יכל להשתמש באינפורמציה השמורה בו והשלישי רץ על מבנה הנתונים בסדר הפוך.

דוגמה לשימוש:

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

int main() {
  list<double> l;
  l.push_back(12);
  l.push_front(15);
  l.push_back(122);
  for(list<double>::const_iterator p = l.begin(); p!=l.end(); p++)
    cout << *p << "<->"; // use of const_iterator
  cout << endl;
  list<double>::iterator p = l.begin();
  // p += 2; // will not compile: p is only a bidirectional iterator
           //not a random access iterator
  for(list<double>::iterator p = l.begin(); p!=l.end(); p++)
    (*p)++; // use of non-const iterator
  for(list<double>::reverse_iterator p = l.rbegin(); p!=l.rend(); p++)
    cout << *p << "<->"; // print the list backwards
  cout << endl;
  p++; // the p that was declared before
  l.insert(p,777); // insert 777 as a second element.
  for(list<double>::reverse_iterator p = --l.rend(); p!=--l.rbegin(); p--)
    cout << *p << "<->"; // this is a twisted way of programing !
  cout << endl;
  l.erase(l.begin(),l.end()); // equal to l.clear()
  cout << l.size() << endl;
}
```

בדוגמה זו נעשה שימוש בפונקציות של מבנה נתונים המשתמשות באיטרטורים (insert ו- erase), גם פונקציות אלה הן חלק מהממשק שממשים רב מבני הנתונים.

אלגוריתמים

הספרייה הסטנדרטית כוללת מספר אלגוריתמים בסיסיים הפועלים על איטרטורים של מבני הנתונים. בכדי להשתמש באלגוריתמים יש לכלול את ה – header: <algorithm>.
נראה דוגמה לשימוש באלגוריתם המחליף ערכים של מבנה נתונים ובאלגוריתם הסופר מופעים של ערך מסוים:

```
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;
int main() {
	deque<double> d;
	d.push_front(2.2);
	d.push_back(2.3);
```

```
        d.push_back(2.4);
        d.push_back(2.3);
        replace(d.begin(),d.end(),2.3,2.4);
        cout << count(d.begin(),d.end(),2.4) << endl;
}
```

פלט התוכנית יהיה : 3.

איטרטורים על זרמי קלט\פלט

ניתן להשתמש באיטרטורים של זרמי הקלט והפלט. הדוגמה הבאה מראה שימוש כזה :

```
#include <iostream>
#include <deque>
#include <algorithm>
#include <string>
using namespace std;
int main() {
  //type '.' to end input
  deque<int> d;
  ostream_iterator<int> oo(cout,"|"); // a '|' will be printed after each element printed
                                              //  through this iterator
  istream_iterator<int> ii(cin);
  istream_iterator<int> eostr;
  copy(ii,eostr,back_inserter(d));
  sort(d.begin(),d.end());
  copy(d.begin(),d.end(),oo);
  cout << endl;

  ostream_iterator<string> os(cout,"!\n");
  *os = "this is a twisted way to say : hello world";
}
```

כאן יש שימוש בשני אלגוריתמים נוספים : copy – המעתיק טווח בין איטרטורים למקום המוצבע ע"י איטרטור אחר, ו – sort הממיין טווח בין איטרטורים. ה – back_inserter משמש להקצעת מקום עבור האלמנטים החדשים שמוכנסים, אלמלא היינו משתמשים בו היינו דורסים זיכרון. בהעתקה לזרם הפלט, לא היינו צריכים back_inserter כיוון שאין צורך בהקצאה.

ה – default ctor של istream_iterator נותן איטרטור המצביע לסוף זרם פלט (כמו 'end()' של istream).

למבני הנתונים של הספרייה הסטנדרטית אין פונקציות הדפסה מובנות אך ניתן להדפיס אותם בקלות כמו שעשינו כאן.

דוגמאות לאלגוריתמים נוספים

הקוד הבא מדגים שימוש במספר אלגוריתמים נוספים :

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
typedef vector<int> Vint;
typedef Vint::iterator iterator;
void print(const int& a) {
  cout << "<*" << a << "*>";
}
bool devide3(const int& n) {
  return n%3 == 0;
```

```
}
int main() {
  Vint v;
  istream_iterator<int> is(cin);
  istream_iterator<int> eos;
  ostream_iterator<int> os(cout," ");
  copy(is,eos,back_inserter(v));
  for_each(v.begin(),v.end(),print); // invokes 'print' for each element
  cout << endl;
  iterator p  = find(v.begin(),v.end(),7); // find the first appearance of '7'
  copy(p,v.end(),os); // from the first 7 to the end
  cout << endl;
  iterator p1  = find_if(v.begin(),v.end(),devide3); // find the first element
                                                      // that satisfy devide3
  copy(p1,v.end(),os); // from the p1 to the end
  cout << endl;
  if(p < p1) // possible for random-access iterator
    cout << count(p,p1,1); // count the number of '1' between p and p1
  else
    cout << count(p1,p,1);
  cout << endl;
  replace(v.begin(),v.end(),7,7777);
  copy(v.begin(),v.end(),os);
  cout << endl;
  unique_copy(v.begin(),v.end(),os); // copy element that are not duplicates
  cout << endl;
}
```

האופן בו מוגדרת האינטראקציה בין האלגוריתמים, האיטרטורים ומבני הנתונים – מבטיחה מימוש יעיל.

נתבונן בדוגמה הבאה:

```
#include <iostream>
#include <deque>
#include <list>
#include <algorithm>
using namespace std;
int main() {
        deque<int> d;
        list<int> l;
        int array[5] = {4,5,2,3,1};
        ostream_iterator<int> os(cout,"|");
        ostream_iterator<int> osl(cout,"<->");
        copy(array,array+5,front_inserter(d));
        copy(array,array+5,front_inserter(l));
        copy(d.begin(),d.end(),os);
        cout << endl;
        copy(l.begin(),l.end(),osl);
        cout << endl;
        sort(d.begin(),d.end());
        // sort(l.begin(),l.end());  will not compile !
        l.sort();
```

```
        copy(d.begin(),d.end(),os);
        cout << endl;
        copy(l.begin(),l.end(),osl);
        cout << endl;
}
```

(שימו לב כיצד האלגוריתמים הכלליים, פועלים גם על איטרטורים פרימיטיביים – כלומר מצביעים)

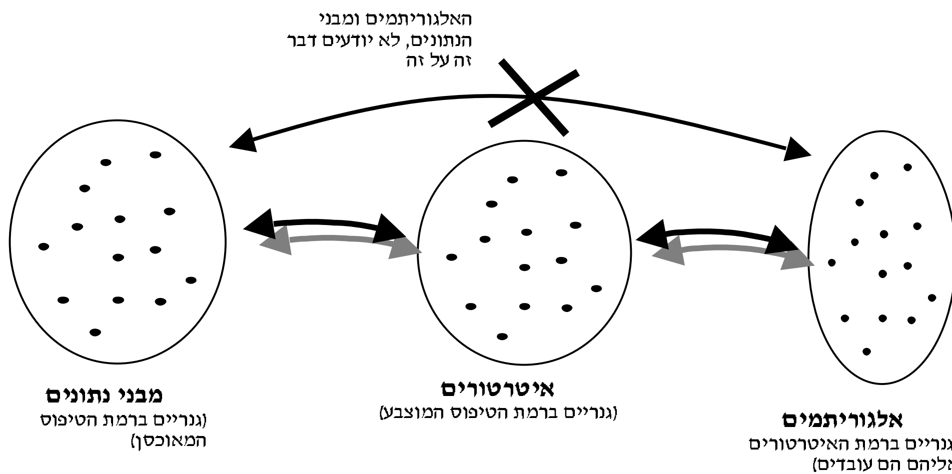
האלגוריתם הכללי sort לא יכול לפעול על האיטרטור שמחזיר list כיוון שהאיטרטור הוא מסוג bidirectional ולא מסוג random-access כפי שהוא מצפה. האלגוריתם sort הוא אלגוריתם מיון יעיל (O(nlogn)) ולכן הוא זקוק לגישה ישירה לאברים של מבנה הנתונים. אם האלגוריתם היה פועל על list כרגיל (כלומר מבצע את הגישה לאלמנט הכללי ע"י מעבר סידרתי) הוא היה מאד לא יעיל. זאת הסיבה שלמבנה הנתונים list יש אלגוריתם מיון משלו הפועל ביעילות תחת המגבלות של list.
דוגמה זו ממחישה עקרון מרכזי של הארכיטקטורה של הספרייה הסטנדרטית (למעשה עקרון של STL) :

- אלגוריתם פועל על סוג מסוים של איטרטור, אותו לא מעניין לאיזה מבנה נתונים האיטרטור שייך, מעניין אותו רק אם לאיטרטור יש את סט הפעולות שהוא צריך ממנו. האלגוריתם מבצע את מלאכתו בצורה יעילה.

- כל מבנה נתונים מחזיר איטרטור הממש רק את אותן פעולות שאותן הוא יכל לבצע בצורה יעילה.

- באופן זה, כל מניפולציה סטנדרטית על מבני הנתונים מובטחת להיות יעילה.

ציור :



האופן בו בנויה הספרייה מאפשר הרחבה נוחה:
על כל מבנה נתונים חדש שנוסיף יפעלו כל האלגוריתמים הקיימים היכולים לפעול עליו בצורה יעילה.
כל אלגוריתם גנרי שנוסיף, יוכל לפעול על כל מבני הנתונים הקיימים שעליהם הוא יכל לעבוד בצורה יעילה.

ניתן דוגמה למימוש אלגוריתם כללי ושימוש בו עם שני סוגים שונים של איטרטורים :

```
#include <list>
#include <iostream>
```

```
template <typename InputIterator, typename OutputIterator>
void mycopy(InputIterator b, InputIterator e, OutputIterator t) {
	while(b!=e)
		*(t++) = *(b++);
}

int main() {
	int array[] = {1,6,8,9,0};
	std::ostream_iterator<int> os(std::cout,"<->");
	std::list<int> l;
	mycopy(array,array+5,std::back_inserter(l));
	mycopy(l.begin(),l.end(),os);
	std::cout << std::endl;
	mycopy(l.rbegin(), l.rend(),os);
	std::cout << std::endl;
}
```

שימו לב כי שמות טיפוסי האיטרטורים ב – mycopy מתאימים לשמות קטגוריות אותם הגדרנו כאשר דיברנו על איטרטורים. כאשר כותבים אלגוריתם גנרי כדאי לתת את השם המתאים לטיפוס מכיוון שאז ברור על איזה סוגי איטרטורים יכל האלגוריתם לעבוד. כדאי לבחור את הקטגוריה הגדולה ביותר האפשרית (הגבלת כלליות מינימלית או הנחת ממשק מינימלית)

עזרים לחישובים נומריים

הספרייה הסטנדרטית מספקת מספר עזרים לחישובים נומריים. דוגמה אחת היא המחלקה המייצגת מספר מרוכב:

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

int main() {
  complex<double> c(1,2);
  cout << c+c << "   " << c*c << endl;
  cout << pow(c,2) << endl;
}
```

דוגמה לשימוש ב – map

מבנה הנתונים map מאפשר למפות אובייקטים מטיפוסים מסוג אחד לאובייקטים מטיפוסים מסוג אחר, לדוגמה:

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
int main() {
  map<string,int> ages;
  ages["Abraham"] =120;
  ages["Issac"] =80;
  ages["Jacov"] =40;
  cout << ages["Abraham"] << endl;
  if(ages.find("Josef")==ages.end())
	cout << "Josef is not in the map\n";
}
```

הערה בקשר לשגיאות קומפילציה

מכיוון שרוב השירותים של הספרייה הסטנדרטית, ניתנים כ – templates (מחלקות או פונקציות), הרבה פעמים קשה להבין את הערות הקומפילציה. ההערות מפנות לקוד של הספרייה עצמה שאותו אסור לנו לשנות. לפעמים ההודעות מתייחסות לפונקציות שאינן ממומשות בטיפוס הכללי עליו עובד ה – template. לדוגמה:

```
class A {};
int main() {
	set<A> s;
	A a;
	s.insert(a);
}
```

טעויות קומפילציה בקוד זה, נגרמות מהעובדה ש – A לא ממש את operator < כפי שצריך טיפוס המאוכסן ב – set לממש. מימוש הפונקציה יפתור את בעיית הקומפילציה:

```
class A {
public:
	bool operator <(const A& other) const {…}
};
```

הערה כללית

כל מה שנאמר כאן יכל לשמש רק כמבוא כללי לספריה הסטנדרטית, את שאר החומר אפשר ללמוד מהספר של Stroustrup (פרקים 3 ו 16-22) או\וגם תוך כדי עבודה עם שימוש בתיעוד online כמו זה הנמצא באתר.