תכנות מתקדם ושפת C++
מצגת 2

אלגוריתמים

נושאים

- אלגוריתמים
- back_inserter()
- פרדיקט
- אובייקט פונקציה
- ביטוי למבדה

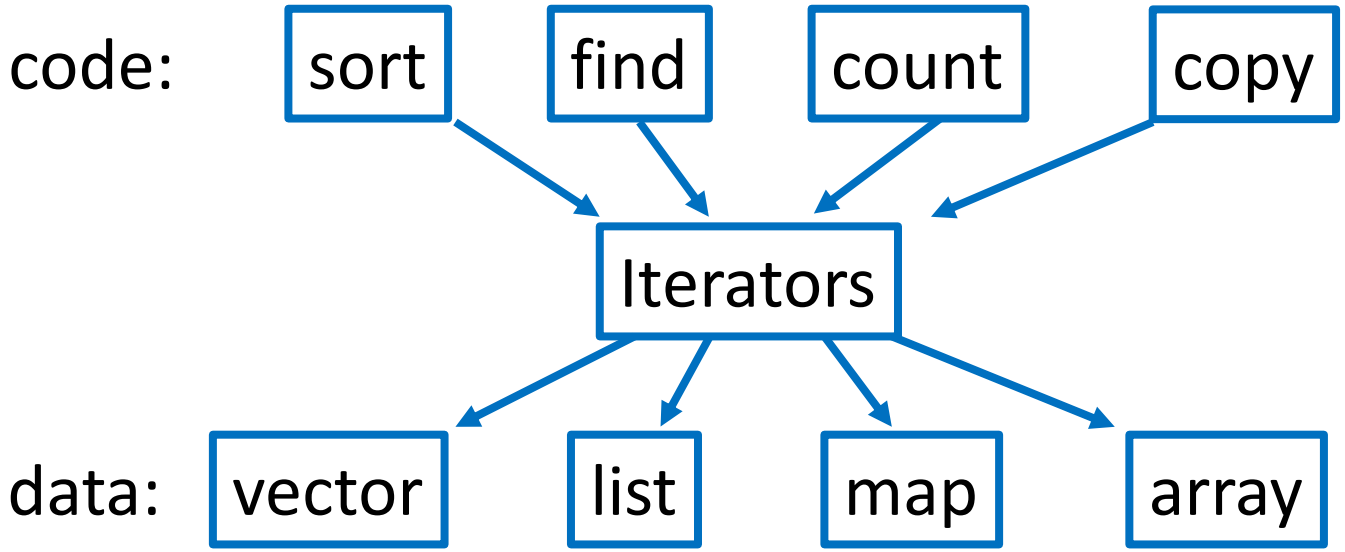
אלגוריתמים

המיכלים מגדירים מספר קטן של פעולות: הוספה, מחיקה, גודל.

ישנם פעולות נוספות שנרצה לעשות: חיפוש איבר, החלפת איבר, סידור מחדש של האיברים.

כדי שלא נצטרך להגדיר את כל הפעולות עבור כל המיכלים, הספרייה הסטנדרטית (STL) הגדירה אלגוריתמים שיכולים לפעול הרבה מיכלים.

המיכל מספק איטרטור, האלגוריתם קורא וכותב את הנתונים שבמיכל באמצעות האיטרטור.



find

find מחפש האם המיכל מכיל ערך מסוים.

האלגוריתם עובר על המיכל בתחום של שני איטרטורים:

```
int val = 42; // value we'll look for
// result will denote the element we want if it's in vec,
// or vec.cend() if not
auto result = find(vec.cbegin(), vec.cend(), val);
cout << "The value " << val
<< (result == vec.cend()
? " is not present" : " is present") << endl;
```

find

האלגוריתם עובר על המיכל בתחום של שני **איטרטורים**,

לכן אפשר להשתמש בו גם כדי לחפש במיכלים אחרים:

```
// look through string elements in a list
string val = "a value";
auto result = find(1st.cbegin(), 1st.cend(), val);
```

מצביעים פועלים כמו **איטרטורים**, לכן אפשר לחפש גם **במערך**:

```
int ia[] = {27, 210, 12, 47, 109};
int val = 83;
int* result = find(ia, &ia[5], val);
// int* result = find(begin(ia), end(ia), val);
```

אפשר לחפש **בחלק** מהמערך:

```
// search from ia[1] up to but not including ia[4]
auto result = find(ia + 1, ia + 4, val);
```

accumulate

accumulate מסכם את האיברים שבמיכל.

עובר על המיכל בתחום של שני **איטרטורים**, ומוסיף אותם לפרמטר השלישי.

הפרמטר השלישי קובע איזו פעולת חיבור תתבצע:

```
// sum the elements in vec starting the summation with 0
int sum = accumulate(vec.cbegin(), vec.cend(), 0);
```

הפעולה + מוגדרת עבור string לכן אפשר לחבר **מחרוזות**:

```
string sum = accumulate(v.cbegin(), v.cend(), string(""));
```

שגיאה, הפעולה + **לא** מוגדרת עבור char *:

```
// error: no + on const char*
string sum = accumulate(v.cbegin(), v.cend(), "");
```

באלגוריתמים כמו find ו- accumulate שרק **קוראים**, עדיף להשתמש ב- cbegin ו- cend

equal

equal בודק האם שני מיכלים הם שווים ומחזיר אמת או שקר.

האלגוריתם מקבל שני איטרטורים עבור המיכל הראשון ואחד עבור המיכל השני:

```
// roster2 should have at least as many elements as roster1
equal(roster1.cbegin(), roster1.cend(), roster2.cbegin());
```

אפשר להשוות בין מיכלים שונים, ואפילו בין מיכלים שלהם אלמנטים שונים ובלבד שאפשר להשוות ביניהם באמצעות ==.

לדוגמה:

```
list<string> roster1 = {"a", "an", "the"};
vector<const char*> roster2 = {"a", "an", "the"};
```

האלגוריתם מניח שהמיכל השני גדול לפחות כמו הראשון.

copy

copy מעתיק איברים ממיכל למיכל.

האלגוריתם מקבל שני איטרטורים עבור המיכל הראשון ואחד עבור המיכל השני:

```
// use copy to copy one built-in array to another
int a1[] = {0,1,2,3,4,5,6,7,8,9};
int a2[sizeof(a1)/sizeof(*a1)]; // a2 has same size as a1
// ret points just past the last element copied into a2
auto ret = copy(begin(a1), end(a1), a2);
```

sort, unique

sort ממיין את המיכל לפי האופרטור <.

unique מסדר את המיכל כך שבתחילת המיכל לא יופיעו איברים כפולים.

האלגוריתמים מקבלים שני איטרטורים:

```
void elimDups(vector<string> &words) {
// sort words alphabetically so we can find the duplicates
    sort(words.begin(), words.end());
// unique reorders the input so that each word appears once
// in the front portion of the range
// and returns an iterator one past the unique range
    auto end_unique = unique(words.begin(), words.end());
// erase uses a vector operation to remove the non-unique
    words.erase(end_unique, words.end());
}
```

sort, unique

דוגמה:

the quick red fox jumps over the slow red turtle

```
sort(words.begin(), words.end());
```

fox	jumps	over	quick	red	red	slow	the	the	turtle

```
auto end_unique = unique(words.begin(), words.end());
```

fox	jumps	over	quick	red	slow	the	turtle	???	???

end_unique
(one past the last unique element)

```
words.erase(end_unique, words.end());
```

fox	jumps	over	quick	red	slow	the	turtle

replace, replace_copy

replace מחליף איבר מסוים באיבר אחר בתוך המיכל.

replace_copy מחליף איבר מסוים באיבר אחר ומעתיק למיכל אחר.

מסדר את המיכל כך שבתחילת המיכל לא יופיעו איברים כפולים.

```
// replace any element with the value 0 with 42
replace(ilst.begin(), ilst.end(), 0, 42);

// leave the original sequence unchanged
// a third iterator is a destination to write the sequence
replace_copy(ilst.cbegin(), ilst.cend(), ivec.begin(), 0, 42);
```

צריך לוודא שבמיכל האחר יש מספיק מקום.

back_inserter

אפשר להשתמש ב- back_inserter כדי להעתיק למיכל שאין בו מספיק מקום או למיכל ריק.

back_inserter מוסיף איברים למיכל ומעתיק לתוכם.

```
vector<int> vec; // empty vector
auto back_it = back_inserter(vec);
// assigning through back_it adds elements to vec
*back_it = 42; // vec now has one element with value 42
```

בדוגמה הקודמת:

```
vector<int> vec; // empty vector
// use back_inserter to grow destination as needed
replace_copy(ilst.cbegin(), ilst.cend(),
     back_inserter(ivec), 0, 42);
```

back_inserter

אפשר להשתמש ב- back_inserter כדי להעתיק למיכל שאין בו מספיק מקום או למיכל ריק.

back_inserter מוסיף איברים למיכל ומעתיק לתוכם.

```
vector<int> vec; // empty vector
auto back_it = back_inserter(vec);
// assigning through back_it adds elements to vec
*back_it = 42; // vec now has one element with value 42
```

בדוגמה הקודמת:

```
vector<int> vec; // empty vector
// use back_inserter to grow destination as needed
replace_copy(ilst.cbegin(), ilst.cend(),
     back_inserter(ivec), 0, 42);
```

Insert Iterators

back_inserter יוצר אוביקט שמתנהג כמו איטרטור ומשתמש ב- push_back

front_inserter יוצר איטרטור שמשתמש ב- push_front

inserter יוצר איטרטור שמשתמש ב- insert

הפעולות המוגדרות עבור Insert Iterators:

```
it = t
// calls c.push_back(t), c.push_front(t)
// or c.insert(t,p) where p is the iterartor given to insert

*it, ++it ,it++
// Each operator returns it, they do nothing
```

back_inserter

copy מעתיק איברים ממיכל למיכל.

האלגוריתם מקבל שני איטרטורים עבור המיכל הראשון ואחד עבור המיכל השני:

```
copy(begin, end, out);
```

מימוש:

```
while(begin != end)
     *out++ = *begin++;
```

אם כתבנו:

```
copy(begin, end, back_inserter(c));
```

ההשמה ל back_inserter תוסיף את כל איבר לסוף המיכל, האופרטורים * ו- ++ לא צריכים לעשות דבר.

מימוש של find

```
template <typename In, typename X>
In find(In begin, In end, const X& x)
{
     while (begin != end && *begin != x)
          ++begin;
     return begin;
}

void f(const string& s)
{
auto p_space = find(s.begin(),s.end(),' ');
auto p_whitespace = find_if(s.begin(),s.end(), isspace);
}
```

מימוש של replace

```
template<typename For, typename X>
void replace(For beg, For end, const X& x, const X& y) {
    while (beg != end){
        if (*beg == x)
            *beg = y;
        ++beg;
    }
}

void f(const string& s) // replace space by underscore
{
    replace(s.begin(),s.end(),' ', '_');
}
```

מימוש של transform

```
template<typename In, typename Out, typename Op>
Out transform(In first, In last, Out res, Op op)
{
    while (first!=last)
        *res++ = op(*first++);
    return res;
}

void toupper(string& s) // remove case
{
    transform(s.begin(),s.end(),s.begin(),toupper);
}
```

העברת פונקציה לאלגוריתמים

הרבה אלגוריתמים משווים איברים, ולצורך זה משתמשים באופרטורים < או ==.

לפעמים נרצה להגדיר בעצמנו מה קטן ומה שווה.

לאלגוריתמים יש גרסאות שמקבלות פונקציה שמחליפה את < או ==.

דוגמה, מיון לפי גודל מילה:

```
// comparison function to be used to sort by word length
bool isShorter(const string &s1, const string &s2)
{
return s1.size() < s2.size();
}
// sort on word length, shortest to longest
sort(words.begin(), words.end(), isShorter);
```

העברת ביטוי למבדה לאלגוריתמים

כשמעבירים פונקציה לאלגוריתמים, צריך להגדיר אותה בנפרד.

מלבד זאת, הפונקציה לא יכולה לקבל יותר מאחד או שני פרמטרים שהאלגוריתם מעביר לה.

ביטוי למדה הוא פונקציה ללא שם שאפשר להגדיר בתוך הקריאה לאלגוריתם, יש לו את הצורה הבאה:

```
[capture list] (parameter list) { function body }
```

דוגמה, הדפסת איברי המיכל עם for_each:

```
// print words, each one followed by a space
for_each(words.begin(), words.end(),
     [](const string &s){cout << s << " ";});
cout << endl;
```

ביטוי למבדה

ביטוי למדה יכול להשתמש במשתנים של הפונקציה שבה הוא מוגדר רק אם הם מועברים ב- **Capture List**.

דוגמה, מציאת האיבר הראשון הגדול מערך מסוים עם find_if:

```
// get an iterator to the first element
// whose size() is >= sz
int sz = 10;
auto wc = find_if(words.begin(), words.end(),
     [sz](const string &a){ return a.size() >= sz; });
```

דוגמה, ספירת האיברים שקטנים מערך מסוים עם count_if:

```
// count number of values less than x
int x = 50;
int c = count_if(vec.begin(), vec.end(),
            [x](int a){ return a < x; });
```

ביטוי למבדה

Capture by Reference מאפשר לביטוי למדה לשנות ערכים מחוץ לביטוי.

דוגמה, חשב את סכום האיברים ושמור אותו במשתנה שהוגדר מחוץ ללמדה:

```
int sum = 0;
for_each(vec.begin(), vec.end(),
     [&sum] (int x) { sum += x; });
```

דוגמה, הדפסת איברי המיכל:

```
// Print words to os separated by c:
void print_words(vector<string> &words,
     ostream &os = cout, char c = ' ')
{
   for_each(words.begin(), words.end(),
      [&os, c](const string &s) { os << s << c; });
} // the only way to capture os is by reference
```

העברת אובייקט פונקציה לאלגוריתמים

מחלקה שמעמיסה את אופרטור הקריאה לפונקציה, מאפשרת להשתמש באובייקטים של אותה מחלקה כאילו הם פונקציה.

דוגמה, אובייקט פונקציה שמחזיר את הערך המוחלט של מספר :

```
struct absInt {
     int operator()(int val) const {
          return val < 0 ? -val : val;
     }
};

int i = -42;
absInt absObj; // object that has a function-call operator
int ui = absObj(i); // passes i to absObj.operator()
```

אובייקט פונקציה

לאובייקט פונקציה יכולים להיות משתנים נוספים:

```
class PrintString {
public:
    PrintString(ostream &o = cout, char c = ' '):
        os(o), sep(c) { } // constructor
    void operator()(const string &s) const
        { os << s << sep; }
private:
    ostream &os; // stream on which to write
    char sep; // character to print after each output
};
```

אובייקט פונקציה

לאובייקט פונקציה יכולים להיות משתנים נוספים:

```
PrintString printer; // uses the defaults
printer(s); // prints s followed by a space on cout
PrintString errors(cerr, '\n');
errors(s); // prints s followed by a newline on cerr
```

אפשר להשתמש באובייקט הפונקציה כארגומנט לאלגוריתם:

```
for_each(vs.begin(), vs.end(), PrintString(cerr, '\n'));
// Third argument is a temporary object of type PrintString
```

אלגוריתמים

`p=find(b,e,x)` p is the first p in [b:e) so that ∗p==x

`p=find_if(b,e,f)` p is the first p in [b:e) so that f(∗p)==true

`n=count(b,e,x)` n is the number of elements ∗q in [b:e) so that ∗q==x

`n=count_if(b,e,f)` n is the number of elements ∗q in [b:e) so that f(∗q)

`replace(b,e,v,v2)` Replace elements ∗q in [b:e) so that ∗q==v by v2

`replace_if(b,e,f,v2)` Replace elements ∗q in [b:e) so that f(∗q) by v2

`p=copy(b,e,out)` Copy [b:e) to [out:p)

`p=copy_if(b,e,out,f)` Copy elements ∗q from [b:e) so that f(∗q) to [out:p)

`p=unique(b,e)` Copy [b:e) to [b:p); discard adjacent duplicates

`accumulate(b,e,init)` Sum elements of [b:e) to init