Tentamen C++-programmering

2010–04–13, 14.00–19.00

Hjälpmedel: En valfri C++-bok. OH-bilderna från föreläsningarna är *inte* tillåtna.

Du ska i dina lösningar visa att du behärskar C++ och att du kan använda C++ standardklasser. "C-lösningar" ger inga poäng, även om de är korrekta.

Uppgifterna ger preliminärt $18 + 12 + 10 + 10 = 50$ poäng. För godkänt krävs 25 poäng (3/25, 4/33, 5/42 respektive G/25, VG/38).

---

1. C++ har en standardiserad vektorklass men inte någon matrisklass. En sådan klass skulle man kunna använda så här:

```
Matrix<int> a(3, 4);
for (size_t i = 0; i != 3; ++i) {
    for (size_t j = 0; j != 4; ++j) {
        a[i][j] = (i + 1) * (j + 1);
    }
}
cout << a << endl;
Matrix<int> b = a;
b += a;
cout << b << endl;
cout << a + b << endl;
int x[] = {1, 2, 3, 4};
vector<int> v(x, x + 4);
vector<int> w = a * v;
for (size_t i = 0; i != w.size(); ++i) {
    cout << w[i] << " ";
}
cout << endl;
```

Man ska alltså kunna 1) skapa en matris med element av godtycklig typ, 2) komma åt elementen i en matris (naturligtvis också i en konstant matris), 3) skriva ut en matris (i radvis ordning), 4) addera två matriser, 5) multiplicera en matris och en vektor. Implementera klassen `Matrix` enligt följande anvisningar:

- Du ska bara implementera de konstruktioner som är nödvändiga för att exempelprogrammet ska fungera.
- Matriselementen *ska* lagras i en medlemsvariabel av typen `vector< vector<T> >` (elementen i en rad lagras i en vektor, och matrisen är en vektor av rader).
- Du kan förutsätta att alla index och alla matris- och vektorstorlekar är korrekta.
- Addition av matriser: $sum_{ij} = a_{ij} + b_{ij}$.
- Multiplikation av matris och vektor: $w_i = \sum_j a_{ij} v_j$.

2. Klassen `Matrix` från uppgift 1 ska förses med en iterator så att man kan skriva satser liknande dessa (nollställning av hela matrisen):

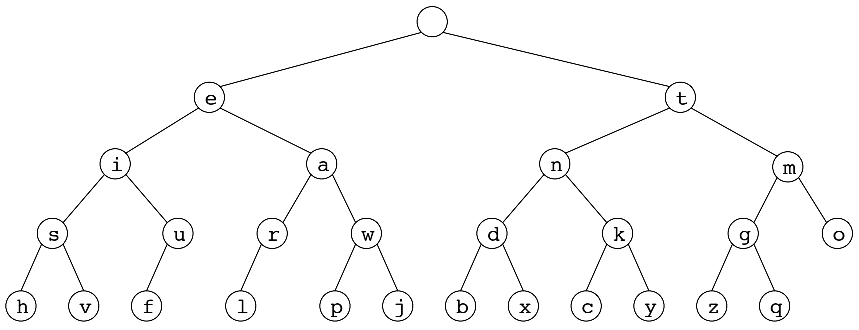
```
Matrix<int> a(3, 4);
for (Matrix<int>::iterator it = a.begin(); it != a.end(); ++it) {
    *it = 0;
}
```

   Implementera iteratorn (bara de konstruktioner som används i exemplet). Iteratorn ska traversera matrisen radvis. Ange också vilka ändringar eller tillägg som måste göras i `Matrix`.

3. I Morse-alfabetet, som inte används mycket nuförtiden, representeras varje tecken av en följd av punkter och streck. Början av alfabetet har följande utseende:

a	`.-`
b	`-...`
c	`-.-.`
d	`-..`

   Med hjälp av trädet i nedanstående figur kan man översätta en följd av punkter och streck till en bokstav a–z. Man börjar i trädets rot och flyttar sig nedåt: till vänster om man har en punkt, till höger om man har ett streck. Exempel: `.-.` ger vänster–höger–vänster det vill säga tecknet `'r'`.



   I ett program representeras trädet av följande klass:

```
class MorseTable {
public:
    MorseTable(); // creates the Morse table
    string translate(const string& code) const;
private:
    struct Node {
        Node* left;  // the node to the left under this node (0 if none)
        Node* right; // - to the right
        char symbol; // the character in the node
        Node(char s) : symbol(s), left(0), right(0) {}
    };
    Node* root; // pointer to the root of the tree
};
```

   Implementera operationen `translate`. Strängen `code` innehåller ett antal Morsekoder (följder av punkter och streck) åtskilda av blanka, som ska översättas till klartext. En Morsekod som inte finns i tabellen ska översättas till `'?'`.

4. Implementera STL-funktionen `set_intersection` (den andra varianten där elementen jämförs med ett funktionsobjekt) som beskrivs i bifogade dokumentation från SGI. Implementera sedan den första varianten genom att utnyttja implementeringen av den andra varianten.

set_intersection

Algorithms

Category: algorithms

Function

Component type: function

Prototype

`Set_intersection` is an overloaded name; there are actually two `set_intersection` functions.

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                                InputIterator2 first2, InputIterator2 last2,
                                OutputIterator result);

template <class InputIterator1, class InputIterator2, class OutputIterator,
          class StrictWeakOrdering>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                                InputIterator2 first2, InputIterator2 last2,
                                OutputIterator result,
                                StrictWeakOrdering comp);
```

Description

`Set_intersection` constructs a sorted range that is the intersection of the sorted ranges `[first1, last1)` and `[first2, last2)`. The return value is the end of the output range.

In the simplest case, `set_intersection` performs the "intersection" operation from set theory: the output range contains a copy of every element that is contained in both `[first1, last1)` and `[first2, last2)`. The general case is more complicated, because the input ranges may contain duplicate elements. The generalization is that if a value appears `m` times in `[first1, last1)` and `n` times in `[first2, last2)` (where `m` or `n` may be zero), then it appears `min(m,n)` times in the output range. [1] `Set_intersection` is stable, meaning both that elements are copied from the first range rather than the second, and that the relative order of elements in the output range is the same as in the first input range.

The two versions of `set_intersection` differ in how they define whether one element is less than another. The first version compares objects using `operator<`, and the second compares objects using a function object `comp`.

Definition

Defined in the standard header algorithm, and in the nonstandard backward-compatibility header algo.h.

Requirements on types

For the first version:

- `InputIterator1` is a model of Input Iterator.
- `InputIterator2` is a model of Input Iterator.
- `OutputIterator` is a model of Output Iterator.
- `InputIterator1` and `InputIterator2` have the same value type.
- `InputIterator`'s value type is a model of LessThan Comparable.
- The ordering on objects of `InputIterator1`'s value type is a *strict weak ordering*, as defined in the LessThan Comparable requirements.
- `InputIterator`'s value type is convertible to a type in `OutputIterator`'s set of value types.

For the second version:

- `InputIterator1` is a model of Input Iterator.
- `InputIterator2` is a model of Input Iterator.
- `OutputIterator` is a model of Output Iterator.
- `StrictWeakOrdering` is a model of Strict Weak Ordering.
- `InputIterator1` and `InputIterator2` have the same value type.
- `InputIterator1`'s value type is convertible to `StrictWeakOrdering`'s argument type.
- `InputIterator`'s value type is convertible to a type in `OutputIterator`'s set of value types.

Preconditions

For the first version:

- `[first1, last1)` is a valid range.
- `[first2, last2)` is a valid range.
- `[first1, last1)` is ordered in ascending order according to `operator<`. That is, for every pair of iterators `i` and `j` in `[first1, last1)` such that `i` precedes `j`, `*j < *i` is `false`.
- `[first2, last2)` is ordered in ascending order according to `operator<`. That is, for every pair of iterators `i` and `j` in `[first2, last2)` such that `i` precedes `j`, `*j < *i` is `false`.
- There is enough space to hold all of the elements being copied. More formally, the requirement is that `[result, result + n)` is a valid range, where `n` is the number of elements in the intersection of the two input ranges.
- `[first1, last1)` and `[result, result + n)` do not overlap.
- `[first2, last2)` and `[result, result + n)` do not overlap.

For the second version:

- `[first1, last1)` is a valid range.
- `[first2, last2)` is a valid range.
- `[first1, last1)` is ordered in ascending order according to `comp`. That is, for every pair of iterators `i` and `j` in `[first1, last1)` such that `i` precedes `j`, `comp(*j, *i)` is `false`.
- `[first2, last2)` is ordered in ascending order according to `comp`. That is, for every pair of iterators `i` and `j` in `[first2, last2)` such that `i` precedes `j`, `comp(*j, *i)` is `false`.
- There is enough space to hold all of the elements being copied. More formally, the requirement is that `[result, result + n)` is a valid range, where `n` is the number of elements in the intersection of the two input ranges.
- `[first1, last1)` and `[result, result + n)` do not overlap.
- `[first2, last2)` and `[result, result + n)` do not overlap.

Complexity

Linear. Zero comparisons if either `[first1, last1)` or `[first2, last2)` is empty, otherwise at most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

Example

```
inline bool lt_nocase(char c1, char c2) { return tolower(c1) < tolower(c2); }

int main()
{
  int A1[] = {1, 3, 5, 7, 9, 11};
  int A2[] = {1, 1, 2, 3, 5, 8, 13};
  char A3[] = {'a', 'b', 'b', 'B', 'B', 'f', 'h', 'H'};
  char A4[] = {'A', 'B', 'B', 'C', 'D', 'F', 'F', 'H' };

  const int N1 = sizeof(A1) / sizeof(int);
  const int N2 = sizeof(A2) / sizeof(int);
  const int N3 = sizeof(A3);
  const int N4 = sizeof(A4);

  cout << "Intersection of A1 and A2: ";
  set_intersection(A1, A1 + N1, A2, A2 + N2,
                   ostream_iterator<int>(cout, " "));
  cout << endl
       << "Intersection of A3 and A4: ";
  set_intersection(A3, A3 + N3, A4, A4 + N4,
                   ostream_iterator<char>(cout, " "),
                   lt_nocase);
  cout << endl;
}
```

The output is

```
Intersection of A1 and A2: 1 3 5
Intersection of A3 and A4: a b b f h
```

Notes

[1] Even this is not a completely precise description, because the ordering by which the input ranges are sorted is permitted to be a strict weak ordering that is not a total ordering: there might be values `x` and `y` that are equivalent (that is, neither `x < y` nor `y < x`) but not equal. See the LessThan Comparable requirements for a fuller discussion. The output range consists of those elements from `[first1, last1)` for which equivalent elements exist in `[first2, last2)`. Specifically, if the range `[first1, last1)` contains `n` elements that are equivalent to each other and the range `[first1, last1)` contains `m` elements from that equivalence class (where either `m` or `n` may be zero), then the output range contains the first `min(m, n)` of these elements from `[first1, last1)`. Note that this precision is only important if elements can be equivalent but not equal. If you're using a total ordering (if you're using `strcmp`, for example, or if you're using ordinary arithmetic comparison on integers), then you can ignore this technical distinction: for a total ordering, equality and equivalence are the same.

See also

`includes`, `set_union`, `set_difference`, `set_symmetric_difference`, `sort`

STL Main Page

Contact Us | Site Map | Trademarks | Privacy | Using this site means you accept its Terms of Use
Copyright © 2009 - 2010 Silicon Graphics International. All rights reserved.