ENS Lyon Camp. Day 5. Basic group.
C++.
30 October

Contents

1 Input/Output

1.1 C-style

```c
#include <stdio.h>

int main()
{
    int a;
    FILE *fin, *fout;
    fin = fopen("example.in", "r");
    fout = fopen("example.out", "w");
    fscanf(fin, "%i", &a);
    fprintf(fout, "%i", a);
    fclose(fin);
    fclose(fout);
}
```

There is an issue with long long variables. You should use specifier "%I64d" for Windows and specifier "%lld" for all other operating systems. To prevent errors with this issue you could define the following macros:

```
#ifdef WIN32
    printf("%I64d\n",ans);
#else
    printf("%lld\n",ans);
#endif
```

1.2 C++-style

Another possibility to avoid this issue is usage of C++.

```
#include <fstream>

int main ()
{
    long a;
    ifstream fin ("example.in");
    ofstream fout ("example.out");
    fin >> a;
    fout << a << endl;
    fin.close();
    fout.close();
}
```

2 Stack Overflow Runtime Error

All variables store in two different places in the memory. One of it is called "stack" and another one is called "heap" (In this context there is no connection between them and data structures with the same names). Heap is a region of memory that stores all global variables, memory that allocates using new(), malloc(), etc. functions. Stack is a special region of memory that stores temporary variables created by each function (including the main() function). Stack variables only exist while the function that created them, is running.

You should keep in mind, that there is a limit (varies with OS) on the size of variables that can be store on the stack. So if you declare large array in the main function, you program crash. If you have a lot of recursive calls, you program crash. You can set size of stack manually as follows: #pragma comment(linker, "/STACK:36777216"). The size is specified in bytes.

3 STL containers

3.1 Pair

This class couples together a pair of values, which may be of different types.

```
#include <utility>      // std::pair

int main () {
    // declaration
    std::pair <int,double> p;
```

```
6
7       // create pair
8       p = std::make_pair (10,3.1415);
9
10      // output first and second elements of pair
11      std::cout << p.first << ", " << p.second << '\n';
12  }
```

3.2 Vector

Vectors represent arrays that can change in size. Vectors are very efficient accessing its elements (just like arrays) and relatively efficient adding or removing elements from its end. You can create vectors with the following constructors:

```
1  #include <vector>
2
3  int main ()
4  {
5       std::vector<int> first;                                // empty vector of ints
6       std::vector<int> second (4,100);                       // four ints with value 100
7       std::vector<int> third (second.begin(),second.end());  // iterating second
8       std::vector<int> fourth (third);                       // a copy of third
9
10      int myints[] = {16,2,77,29};
11      std::vector<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );
12 }
```

Here is the examples of vector usage:

```
1  int main ()
2  {
3       std::vector<int> v;
4
5       // Returns whether the vector is empty
6       bool isEmpty = v.empty();
7
8       // Returns the number of elements in the vector.
9       int size = v.size();
10
11      // Adds a new element at the end of the vector, after its current last element.
12      v.push_back(5);
13
14      // Removes the last element in the vector.
15      v.pop_back();
16
17      // Returns a reference to the first element in the vector.
18      int first = v.front();
19
20      // Returns a reference to the last element in the vector.
21      int last = v.back();
22
23      // Returns a reference to the element at position i in the vector container.
```

```
    int x = v[i];
}
```

Time complexity of all operation is $\mathcal{O}(1)$.

3.3 Queue

Queues operates in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other. You can create it in the same way as vector. Here is the examples of queue usage:

```
#include <queue>
int main ()
{
    std::queue<int> q;

    // Adds a new element at the end of the queue, after its current last element.
    q.push(5);

    // Remove the "oldest" element in the queue
    q.pop();

    // Returns whether the queue is empty
    bool isEmpty = q.empty();

    // Returns the number of elements in the queue
    int size = q.size();

    // Returns a reference  "oldest" element in the queue
    int first = q.front();

    // Returns a reference to the the "newest" element in the queue
    int last = q.back();
}

```

3.4 Deque

Deque is a double-ended queue. Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back). It constructs in the same way. Here is the examples of deque usage:

```
#include <deque>
int main ()
{
    std::deque<int> q;

    // Returns whether the queue is empty
    bool isEmpty = q.empty();

```

```
    // Returns the number of elements in the queue
    int size = q.size();

    // Adds a new element at the end of the deque container
    q.push_back(5);

    // Removes the last element in the deque
    q.pop_back();

    // Inserts a new element at the beginning of the deque
    q.push_front(9);

    // Removes the first element in the deque
    q.pop_front();

    // Returns a reference to the first element in the deque
    int first = q.front();

    // Returns a reference to the last element in the container.
    int last = q.back();

    // Returns a reference to the element at position i in the deque.
    int x = v[i];
}

```

3.5 Set

Sets are containers that store unique elements following a specific order (elements must be comparable). Set implementation uses binary search trees. There is two way to redefine comparator of elements

```
bool cmpFunction (int l, int r) {return r < l;}

struct cmpClass {
    bool operator() (const int& l, const int& r) const
    {return r < l;}
};

int main ()
{
   std::set<int> first; // empty set of ints

   int myints[]= {10,20,30,40,50};
   std::set<int> second (myints,myints+5); // range

   std::set<int> third (second);  // a copy of second

   std::set<int> fourth (second.begin(), second.end());  // iterator

   std::set<int,cmpClass> fifth; // redefine comparator using class
```

```
20
21     bool(*fn_pt)(int,int) = cmpFunction; // declate pointer to cmpFunction
22     std::set<int,bool(*)(int,int)> sixth (fn_pt);  // redefine comparator
23 }
24
```

Here is the examples of set usage:

```
1  int main ()
2  {
3      std::set<int> s;
4
5      // Returns whether the set is empty
6      bool isEmpty = s.empty();
7
8      // Returns the number of elements in the queue
9      int size = s.size();
10
11     // Returns a pair, with its member pair::first set to an iterator
12     // pointing to either the newly inserted element
13     // or to the equivalent element already in the set.
14     // The pair::second element in the pair is set to true if a new element
15     // was inserted or false if an equivalent element already existed.
16     s.insert(20);
17
18     int a[]= {5,10,15};
19     s.insert (myints,myints+3);
20
21     // Removes from the set container either a single element
22     // or a range of elements ([first,last)).
23     it = s.begin();
24     s.erase(it); // erasing by value
25     s.erase(20); // erasing by key
26
27     // Searches the container for an element equivalent to val and returns
28     // an iterator to it if found, otherwise it returns an iterator to set::end.
29     s.find(20);
30
31     // Returns an iterator pointing to the first element in the container
32     // which is not considered to go before val
33     // (i.e., either it is equivalent or goes after).
34     std::set<int>::iterator lb = s.lower_bound(20);
35
36     // Returns an iterator pointing to the first element in the container
37     // which is considered to go after val.
38     std::set<int>::iterator ub = s.upper_bound(20);
39 }
40
```

There is similar container multiset, that allows store the equivalent values with the same methods;

```
1 #include <set>
2
3 typedef std::multiset<int>::iterator setIt;
```

```
4
5  int main ()
6  {
7      std::multiset<int> s;
8
9      // Count elements with a specific key
10     int cnt = s.count(42);
11
12     // Get range of equal elements
13     std::pair<setIt,setIt> ret = mymultiset.equal_range(42);
14 }
15
```

Time complexity is $\mathcal{O}(\log n)$. In C++11 there exists unordered set (hash set), that stores unordered value and time complexity is approximately $\mathcal{O}(1)$ (#include <unordered_set>).

3.6 Map

Maps are associative containers that store elements formed by a combination of a key value and a mapped value, following a specific order. You can redefined compare operator in the same way as it was redefined in set. Here is example of usage map.

```
1  #include <map>
2
3  int main ()
4  {
5      std::map<char,int> m;
6
7      // Returns whether the map is empty
8      bool isEmpty = m.empty();
9
10     // Returns the number of elements in the map
11     int size = m.size();
12
13     // Returns a pair, with its member pair::first set to an iterator pointing
14     // to either the newly inserted element
15     // or to the equivalent element already in the map.
16     // The pair::second element in the pair is set to true
17     // if a new element was inserted
18     // or false if an equivalent element already existed.
19     m.insert(std::pair<char,int>('a',100));
20
21     // Removes from the map container either a single element
22     // or a range of elements ([first,last)).
23     it = m.begin();
24     m.erase(it); // erasing by iterator
25     s.erase('a'); // erasing by key
26
27     // Searches the container for an element equivalent to val
28     // and returns an iterator to it if found,
29     // otherwise it returns an iterator to map::end.
```

```
    s.find('a');

    // Returns an iterator pointing to the first element in the container
    // which is not considered to go before val
    std::map<char, int>::iterator lb = m.lower_bound('a');

    // Returns an iterator pointing to the first element in the container
    // which is considered to go after val.
    std::map<char, int>::iterator ub = m.upper_bound('b');

    // If k matches the key of an element in the container,
    // the function returns a reference to its mapped value.
    // If k does not match the key of any element in the container,
    // the function inserts a new element with that key
    // and returns a reference to its mapped value.
    mymap['a'] = 5;
}

```

Similary there exists multimap that allows store the equivalent values with the same methods; In C++11 there exists unordered map (hash map) that stores unordered value and time complexity is approximately $\mathcal{O}(1)$ (#include <unordered_map>).

4 STL sort

```
#include <algorithm>    // std::sort
#include <vector>

bool cmpFunction (int l, int r) { return (r < l); }

struct cmpClass {
    bool operator() (int l, int r) { return (r < l);}
} myobject;

int main () {
    int myints[] = {32,71,12,45,26,80,53,33};
    std::vector<int> myvector (myints, myints+8);

    // using default comparison (operator <):
    std::sort (myvector.begin(), myvector.begin()+4);

    // using function as comp
    std::sort (myvector.begin()+4, myvector.end(), myfunction);

    // using object as comp
    std::sort (myvector.begin(), myvector.end(), myobject);
}

```

There is function std::stable_sort that preserves the relative order of the elements with equivalent values.