Data Structures

Ready Made
Data Structure Libraries

Ready Made Data Structures

- Today, many programming languages contain ready made data structures.
- Programmers can easily use these ready made data structures to solve the problems they want to solve independently of the content (code) of the data structures they will use.
- In C++, these ready made data structures are in the **S**tandard **T**emplate **L**ibrary (STL).

STL

- The main idea of STL is making the complicated parts (coding) of data structures ready and presenting an easy-to-use interface to the programmer.
- For example, a programmer wanting to use a stack of integers can simply create this stack by writing the code below:

```
stack<int> s;
```

- Then, using the methods the STL provides for this data structure, the programmer can carry out the desired operations:

```
s.push(3);
```

Three Types of Data Structures in the STL

- Sequence containers
  (linear data types such as vector and linked list)
  - C++ Vectors
  - C++ Lists
  - C++ Double-Ended Queues
- Container adapters
  (versions of linear data structures with some properties restricted: stack, queue)
  - C++ Stacks
  - C++ Queues
  - C++ Priority Queues
- Associative containers
  (data structures that provide direct access to their elements via search keys)
  - C++ Bitsets
  - C++ Maps
  - C++ Multimaps
  - C++ Sets
  - C++ Multisets

STL Libraries

- STL containers are located in different libraries.
- To use a container the related library should be added to the code:

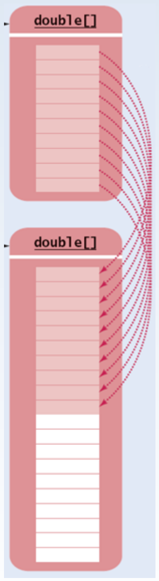
```
<vector>
<list>
<deque>
<queue>
<stack>
<map>
<set>
<valarray>
<bitset>
```

Sequence Containers

- In C++, there are 3 sequence containers:
  - vector – a more efficient array structure
  - list – linked list
  - deque – a type of array structure
- Common methods in sequence containers
  - front → returns the reference of the first element
  - back → returns the reference of the last element
  - push_back → adds an element to the end
  - pop_back → removes the last element

Vector

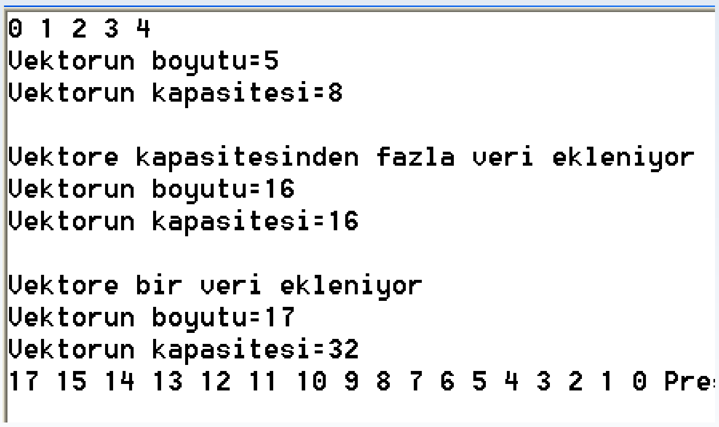
- Is a more reliable array realization.
- Similar to normal arrays, is made up of elements that are located consecutively in memory. The [] operator may be used. Fast access is to data is provided.
- Can dynamically increase its size:
  - When an array with a larger size is needed, it allocates a larger space from memory, and the content of the old array gets copied into this space.
  - The memory allocated from the memory for the old array can be returned to the memory.



Vector Example

```
vector<int> myvec;
int i;
for(i=0;i<5;i++)
      myvec.push_back(i);
for(i=0;i<5;i++)
      cout<<myvec[i]<<" ";
cout<<endl<<“Vector size="<<myvec.size()<<endl;
cout<<"Vector capacity="<<myvec.capacity()<<endl;
cout<<endl<<“Data to be added exceeds vector capacity"<<endl;
int exsize=myvec.size();
int excap=myvec.capacity();
for(i=exsize;i<2*excap;i++)
      myvec.push_back(i);
cout<<"Vector size="<<myvec.size()<<endl;
cout<<"Vector capacity="<<myvec.capacity()<<endl;
myvec.push_back(i+1);
cout<<endl<<“Data is being added to vector"<<endl;
cout<<"Vector size="<<myvec.size()<<endl;
cout<<"Vector capacity="<<myvec.capacity()<<endl;
while(!myvec.empty()){
      cout<<myvec.back()<<" ";
      myvec.pop_back();
}
```

Vector Example



```
0 1 2 3 4
Vektorun boyutu=5
Vektorun kapasitesi=8

Vektore kapasitesinden fazla veri ekleniyor
Vektorun boyutu=16
Vektorun kapasitesi=16

Vektore bir veri ekleniyor
Vektorun boyutu=17
Vektorun kapasitesi=32
17 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 Pre
```

Vector Member Functions

In addition to common methods, there are many other methods that can be used:

- Methods related to capacity: size, max_size, resize, capacity, empty, reserve
- Methods related to access to elements: [] , at, front, back
- Methods that make changes to the vector: assign, push_back, pop_back, insert, erase, swap, clear

You can refer to STL guides for usage details of methods belonging to containers.

List

- The STL container "list" presents an efficient implementation of adding to and removing from the linked list. If addition and removal operations are generally taking place at the end of the list, the "deque" container is more suitable.
- The "list" container is implemented as a doubly linked list. Therefore, we can traverse the list quickly in both directions.

Deque

- "double-ended queue"
- The deque is a structure that has been designed to use the best of the vector and list structures together.
- Like the vector, it provides fast access to elements using an index [] (this is implemented on the array.)
- As in the list, operations such as fast addition and removal of elements at the front and the back have been implemented in an efficient manner (But, it is also possible to insert in between elements of the list.)
- It is the default data structure for a queue structure.
- In addition to the basic operations in a vector, contains the push_front and pop_front functions.
- Note: push_back exists in all containers, but push_front exists in only list and deque.

Container Adapters

- In C++, there are 3 container adapters:
  - stack
  - queue
  - priority_queue

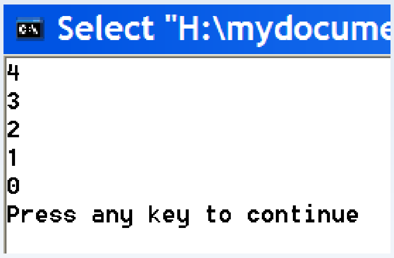
Stack Member Functions

- empty → isempty()
- pop → its difference from our pop() method is that it does not have a return value. void pop(); removes the element at the top of the stack but does not provide access to the element.
- push → push(…)
- size → returns the number of elements in the stack
- top → provides access to the topmost element of the stack but does not remove the element from the stack.

Stack Example

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

int main(){
  stack<int> stack;
  for(int i=0;i<5;i++)
      stack.push(i);
  while(!stack.empty()){
      cout<<stack.top()<<endl;
      stack.pop();
  }
  return EXIT_SUCCESS;
}
```



```
4
3
2
1
0
Press any key to continue
```

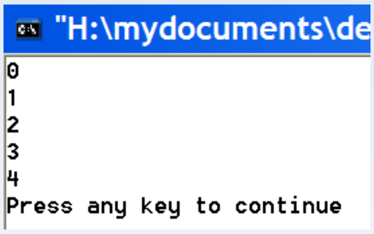
Queue Member Functions

- back→ provides access to the last element of queue
- empty → isempty()
- front → provides access to the first element of queue
- pop → its difference from our pop() method is that it does not have a return value. void pop(); removes the first element of the queue but does not provide access to the element.
- push → push()
- size → returns the number of elements in the queue

Queue Example

```
#include <iostream>
#include <stack>
#include <queue>
using namespace std;

int main(){
  queue<int> myq;
  for(int i=0;i<5;i++)
      myq.push(i);
  while(!myq.empty()){
      cout<<myq.front()<<endl;
      myq.pop();
  }
  return EXIT_SUCCESS;
}
```


"H:\mydocuments\de

```
0
1
2
3
4
Press any key to continue
```

Priority Queue

- By default, it is ordered from largest to smallest.
- Largest gets processed first.
- It is also possible to specify your own priority type, but more advanced programming knowledge is needed.

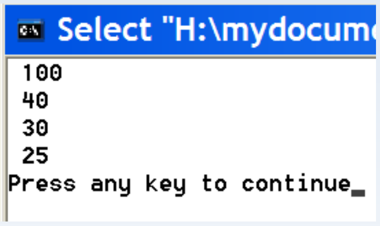
empty
size
top → provides access to the first element
push → add element based on priority
pop → removes the first element.

Priority Queue Example

```
#include <iostream>
#include <stack>
#include <queue>
using namespace std;

int main(){
   priority_queue<int> mypq;

   mypq.push(30);
   mypq.push(100);
   mypq.push(25);
   mypq.push(40);
      while (!mypq.empty())
      {    cout << " " << mypq.top()<<endl;
           mypq.pop();
      }
      return EXIT_SUCCESS;
}
```



```
 100
 40
 30
 25
Press any key to continue_
```

Container Adapters

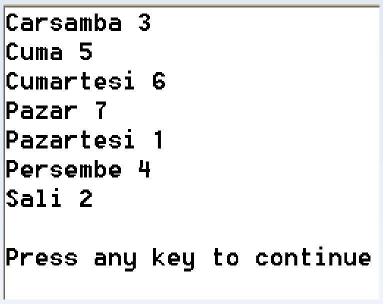
- The advantage of a container adapter is that the user can choose the underlying data structure.
- For example, the underlying data structure for the stack structure is the deque by default. It has been implemented using the deque.
- If we want to use the version of the stack realized on the linked list:
  - `stack<int> stack;` is replaced with
  - `stack<int, list<int> > stack;`
- If we want to do this using a vector, we write
  - `stack<int, vector<int> > stack;`

Associative Containers

- Associative containers:
  - provide direct access to elements via keys
  - store search keys in order
- There are 5 associative containers:
  - multiset – holds only keys, repeats are possible
  - set – holds only keys, repeats are not possible
  - bitset – set used for bit operations
  - multimap – holds keys and associated values, repeats are possible.
  - map – holds keys and associated values, repeats are not possible.
- Common functions:
  - find, lower_bound, upper_bound, count

Example of Using Keys (map)

```
map<string, string> strMap;
strMap[“Monday"]    = "1";
strMap[“Tuesday"]   = "2";
strMap[“Wednesday"] = "3";
strMap[“Thursday"]  = "4";
strMap[“Friday"]    = "5";
strMap[“Saturday"]  = "6";
strMap.insert(pair<string, string>(“Sunday", "7"));
while(!strMap.empty()){
    cout<<strMap.begin()->first<<" “
          <<strMap.begin()->second<<endl;
    strMap.erase(strMap.begin());
}
```



```
Carsamba 3
Cuma 5
Cumartesi 6
Pazar 7
Pazartesi 1
Persembe 4
Sali 2

Press any key to continue
```

Iterators

- When we use the ready made libraries of C++ for data structures, we use structures called iterators to iterate over the elements of a data structure.
- Iterators resemble pointers.
- Just as we use pointers when traversing a linked list,
  - Ex: Node *ptr=head;
  - while(ptr!=NULL) ptr=ptr->next;

  when traversing the list structure of the STL, we use iterators. (We do not know what the node structure of the "list" is. We are only processing the data part.)

Iterators

```
list <int> myl;
for(int i=0;i<5;i++)
   myl.push_back(i);
list <int>::iterator myit;
for(myit=myl.begin();myit!=myl.end();myit++)
   cout<<*myit<<endl;
```

Iterators

- It is not possible to use iterators on container adapters. This is because the way their elements can be accessed is predefined by the adapter structure.
- Ex: It is not possible to move between the elements of a stack. This is because access to the stack is possible only from a single point and this is implemented using the stack structure's own methods.

Iterator types	Direction	Capability
iterator	forward	read/write
const_iterator	forward	read
reverse_iterator	backward	read/write
reverse_const_iterator	backward	read

Iterators

```
list <int> myl;
for(int i=0;i<5;i++)
    myl.push_back(i);
list <int>::reverse_iterator myit;
for(myit=myl.rbegin();myit!=myl.rend();myit++)
    cout<<*myit<<endl;
```

Example

To measure the frequency of words in a file containing English plaintext, we will write a program that uses the "map" data type.

Topic: STL and sequential access files

As output, the program will display the 10 most frequently used words in the file.

Example: Word Frequency

- First, words read in order from the file should be put into a map structure. In this way, the frequency of each word will be calculated.

```
FILE *myfile= fopen( "english.txt", "r" );
if(!myfile){  ... }
char word[100];
map <string,int> freq;
while(!feof(myfile)){
    fscanf(myfile,"%s",word);
    freq[word]++;
}
fclose(myfile);
```

Example: Word Frequency

- The map data structure is ordered only by key.
- It is not possible to order by value.
- For that reason, the map should be assigned to another map such that not the words, but the value fields are the keys.
- Since there will be more than one word with the same frequency value, the structure has to be a multimap:

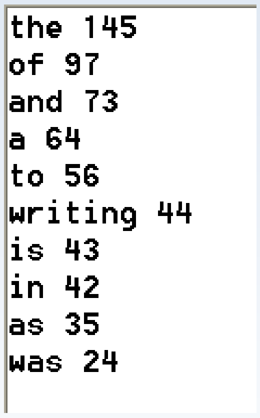
```
multimap <int,string > freq_rev;
map<string,int>::iterator it;
for(it=freq.begin();it!=freq.end();it++)
    freq_rev.insert(make_pair(it->second,it->first));
```

Example: Word Frequency

- In the new multimap, values are ordered from smallest to largest.
- To print the most frequent 10 words to the screen, the 10 records at the end of the structure have to be found.

```
multimap <int,string>::reverse_iterator myit;
int count;
for (myit=freq_rev.rbegin(),count=0;count<10;myit++,count++)
      cout<<(*myit).second<<" "<<(*myit).first<<endl;
```

Example: Word Frequency



```
the 145
of 97
and 73
a 64
to 56
writing 44
is 43
in 42
as 35
was 24
```