THE STANDARD TEMPLATE LIBRARY (STL)

THE STANDARD TEMPLATE LIBRARY (STL)

IS A POWERFUL SET OF COLLECTION CLASSES IN C++

WHEN THE STL WAS ADDED TO C++, IT WAS

AMAZING! REVOLUTIONARY!

WHEN THE STL WAS ADDED TO C++, IT WAS AMAZING! REVOLUTIONARY!

JAVA AND C# HAVE BUILT ON THIS, SO IT NOW SEEMS POWERFUL, BUT NOTHING OUT OF THE ORDINARY

SUCH IS LIFE, SUCH IS PROGRESS:-)

THE STL HAS 3 IMPORTANT SETS OF FUNCTIONALITY

CONTAINERS

CLASSES THAT HOLD STUFF

THE STL HAS 3 IMPORTANT SETS OF FUNCTIONALITY

CONTAINERS

CLASSES THAT HOLD STUFF

ITERATORS

CLASSES THAT DO ELEMENT-BY-ELEMENT STUFF WITH CONTAINERS

THE STL HAS 3 IMPORTANT SETS OF FUNCTIONALITY

CONTAINERS
CLASSES THAT HOLD STUFF

ITERATORS
CLASSES THAT DO ELEMENT-BY-ELEMENT STUFF WITH CONTAINERS

ALGORITHMS
"FUNCTION OBJECTS", CLASSES THAT ENCAPSULATE ALGORITHMS

THE STL HAS 3 IMPORTANT SETS OF FUNCTIONALITY

CONTAINERS

CLASSES THAT HOLD STUFF

ITERATORS

CLASSES THAT DO ELEMENT-BY-ELEMENT STUFF WITH CONTAINERS

ALGORITHMS

"FUNCTION OBJECTS", CLASSES THAT ENCAPSULATE ALGORITHMS

EXAMPLE 67: USE `vector` THE MOST COMMON STL CONTAINER

EXAMPLE 67: USE `vector` THE MOST COMMON STL CONTAINER

THE STL HAS 3 IMPORTANT SETS OF FUNCTIONALITY

CONTAINERS
CLASSES THAT HOLD STUFF

ITERATORS
CLASSES THAT DO ELEMENT-BY-ELEMENT STUFF WITH CONTAINERS

ALGORITHMS
"FUNCTION OBJECTS", CLASSES THAT ENCAPSULATE ALGORITHMS

CONTAINERS

CLASSES THAT HOLD STUFF

BEFORE THE STL, FOLKS EITHER HAD TO RELY ON ARRAYS, OR WRITE THEIR OWN CONTAINERS

CONTAINERS

CLASSES THAT HOLD STUFF

BEFORE THE STL, FOLKS EITHER HAD TO RELY ON ARRAYS, OR WRITE THEIR OWN CONTAINERS

ARRAYS ARE A REAL PAIN:

- THEY CAN'T BE RESIZED DYNAMICALLY
- THEY REQUIRE AN UNDERSTANDING OF POINTERS
- THEY ARE NOT GENERIC

CONTAINERS

CLASSES THAT HOLD STUFF

BEFORE THE STL, FOLKS EITHER HAD TO RELY ON ARRAYS, OR WRITE THEIR OWN CONTAINERS

EVER WONDER WHY LINKED LISTS ARE SUCH A FAVOURITE TOPIC OF PROGRAMMERS A CERTAIN AGE? THIS IS THE ANSWER.

CONTAINERS

CLASSES THAT HOLD STUFF

BEFORE THE STL, FOLKS EITHER HAD TO RELY ON ARRAYS, OR WRITE THEIR OWN CONTAINERS

CONTAINERS

CLASSES THAT HOLD STUFF

BEFORE THE STL, FOLKS EITHER HAD TO RELY ON ARRAYS, OR WRITE THEIR OWN CONTAINERS

THE STL HAS A HOST OF CONTAINERS - VECTORS, MAPS, LISTS, SETS,...

CONTAINERS

CLASSES THAT HOLD STUFF

THE STL HAS A HOST OF CONTAINERS
- VECTORS, MAPS, LISTS, SETS,...

THE COMMONEST AND MOST WIDELY USED OF WHICH IS `vector`

vector

IS EVERYTHING THAT AN ARRAY IS NOT

- THEY CAN BE RESIZED DYNAMICALLY
- THEY REQUIRE NO UNDERSTANDING OF POINTERS OR DYNAMIC MEMORY ALLOCATION
- THEY ARE GENERIC

```
size_t size = 4;
vector<string> stringVector(size);
stringVector[0] = "Janani";
stringVector[1] = "Swetha";
stringVector[2] = "Navdeep";
stringVector[3] = "Vitthal";


stringVector.push_back("Ahmad");
cout<<stringVector.capacity()<<std::endl;
cout<<stringVector.size()<<endl;

for(int i=0; i<stringVector.size(); ++i){
  cout << "Element #" << i << " has value " << stringVector[i] << endl;
}
```

GENERIC

```
size_t size = 4;
vector<string> stringVector(size);
stringVector[0] = "Janani";
stringVector[1] = "Swetha";
stringVector[2] = "Navdeep";
stringVector[3] = "Vitthal";


stringVector.push_back("Ahmad");
cout<<stringVector.capacity()<<std::endl;
cout<<stringVector.size()<<endl;

for(int i=0; i<stringVector.size(); ++i){
  cout << "Element #" << i << " has value " << stringVector[i] << endl;
}
```

GENERIC

SPECIFY THE ELEMENT TO BE CONTAINED AS THE TEMPLATE PARAMETER

```
size_t size = 4;
vector<string> stringVector(size);
stringVector[0] = "Janani";
stringVector[1] = "Swetha";
stringVector[2] = "Navdeep";
stringVector[3] = "Vitthal";


stringVector.push_back("Ahmad");
cout<<stringVector.capacity()<<std::endl;
cout<<stringVector.size()<<endl;

for(int i=0; i<stringVector.size(); ++i){
  cout << "Element #" << i << " has value " << stringVector[i] << endl;
}
```

GENERIC

SPECIFY THE ELEMENT TO BE CONTAINED AS THE TEMPLATE PARAMETER

CAN BE RESIZED DYNAMICALLY

```
size_t size = 4;
vector<string> stringVector(size);
stringVector[0] = "Janani";
stringVector[1] = "Swetha";
stringVector[2] = "Navdeep";
stringVector[3] = "Vitthal";
```

INITIALLY CREATE IT TO HOLD 4 ELEMENTS..

```
stringVector.push_back("Ahmad");
cout<<stringVector.capacity()<<std::endl;
cout<<stringVector.size()<<endl;

for(int i=0; i<stringVector.size(); ++i){
  cout << "Element #" << i << " has value " << stringVector[i] << endl;
}
```

CAN BE RESIZED DYNAMICALLY

```
size_t size = 4;
vector<string> stringVector(size);
stringVector[0] = "Janani";
stringVector[1] = "Swetha";
stringVector[2] = "Navdeep";
stringVector[3] = "Vitthal";

stringVector.push_back("Ahmad");
cout<<stringVector.capacity()<<std::endl;
cout<<stringVector.size()<<endl;

for(int i=0; i<stringVector.size(); ++i){
  cout << "Element #" << i << " has value " << stringVector[i] << endl;
}
```

INITIALLY CREATE IT TO HOLD 4 ELEMENTS..

ADD AN ELEMENT IF YOU NEED TO!

REQUIRE NO UNDERSTANDING OF POINTERS OR DYNAMIC MEMORY ALLOCATION

```
size_t size = 4;
vector<string> stringVector(size);
stringVector[0] = "Janani";
stringVector[1] = "Swetha";
stringVector[2] = "Navdeep";
stringVector[3] = "Vitthal";


stringVector.push_back("Ahmad");
cout<<stringVector.capacity()<<std::endl;
cout<<stringVector.size()<<endl;

for(int i=0; i<stringVector.size(); ++i){
  cout << "Element #" << i << " has value " << stringVector[i] << endl;
}
```

REQUIRE NO UNDERSTANDING OF POINTERS OR DYNAMIC MEMORY ALLOCATION

VECTOR WILL, BY DEFAULT, CALL THE NO-ARGUMENT CONSTRUCTOR OF EACH ELEMENT WHEN IT IS CREATED

VECTOR WILL, BY DEFAULT, CALL THE DESTRUCTOR OF EACH ELEMENT WHEN IT GOES OUT OF SCOPE

(IF YOUR VECTOR HOLDS POINTERS, THOUGH, YOU STILL NEED TO CALL THE DESTRUCTORS OF THE "POINTED-TO" OBJECTS THOUGH)

REQUIRE NO UNDERSTANDING OF POINTERS OR DYNAMIC MEMORY ALLOCATION

VECTOR WILL BY DEFAULT CALL THE NO ARGUMENT CONSTRUCTOR OF EACH ELEMENT WHEN IT IS CREATED

VECTOR WILL, BY DEFAULT, CALL THE DESTRUCTOR OF EACH ELEMENT WHEN IT GOES OUT OF SCOPE

(IF YOUR VECTOR HOLDS POINTERS, THOUGH, YOU STILL NEED TO CALL THE DESTRUCTORS OF THE "POINTED-TO" OBJECTS THOUGH)

WHAT WILL THIS CODE PRINT?

```
vector<Person> personVector(size);
personVector[0] = Person("Janani");
personVector[1] = Person("Swetha");
personVector[2] = Person("Navdeep");
personVector[3] = Person("Vitthal");


personVector.push_back(Person("Ahmad"));
cout<<personVector.capacity()<<std::endl;
cout<<personVector.size()<<endl;

for(int i=0; i<personVector.size(); ++i){
  cout << "Element #" << i << " has value "
 << personVector[i].getName() << endl;
}



}
```

WHAT WILL THIS CODE PRINT?

```
class Person
{
private:
  string name;
public:
  Person(string s) : name(s)
  {
    cout << "In the 1-argument constructor" << endl;
  }
  Person()
  {
    cout << "In the no-argument constructor" << endl;
  }
  ~Person()
  {
    cout << "In the destructor" << endl;
  }
  string getName()
  {
    return name;
  }

};
```

```
vector<Person> personVector(size);
personVector[0] = Person("Janani");
personVector[1] = Person("Swetha");
personVector[2] = Person("Navdeep");
personVector[3] = Person("Vitthal");


personVector.push_back(Person("Ahmad"));
cout<<personVector.capacity()<<std::endl;
cout<<personVector.size()<<endl;

for(int i=0; i<personVector.size(); ++i){
  cout << "Element #" << i << " has value "
 << personVector[i].getName() << endl;
}
```

WHAT WILL THIS CODE PRINT?

```
  vector<Person> personVector(size);
  personVector[0] = Person("Janani");
  personVector[1] = Person("Swetha");
  personVector[2] = Person("Navdeep");
  personVector[3] = Person("Vitthal");


  personVector.push_back(Person("Ahmad"));
  cout<<personVector.capacity()<<std::endl;
  cout<<personVector.size()<<endl;

  for(int i=0; i<personVector.size(); ++i){
    cout << "Element #" << i << " has value "
   << personVector[i].getName() << endl;
  }



}
```

```
In the no-argument constructor
In the no-argument constructor
In the no-argument constructor
In the no-argument constructor
In the 1-argument constructor
In the destructor
In the 1-argument constructor
In the destructor
In the 1-argument constructor
In the destructor
In the 1-argument constructor
In the destructor
In the 1-argument constructor
In the destructor
In the destructor
In the destructor
In the destructor
In the destructor
8
5
Element #0 has value Janani
Element #1 has value Swetha
Element #2 has value Navdeep
Element #3 has value Vitthal
Element #4 has value Ahmad
In the destructor
In the destructor
In the destructor
In the destructor
In the destructor
```

WHAT WILL THIS CODE PRINT?

```
vector<Person> personVector(size);
personVector[0] = Person("Janani");
personVector[1] = Person("Swetha");
personVector[2] = Person("Navdeep");
personVector[3] = Person("Vitthal");
```

```
In the no-argument constructor
In the no-argument constructor
In the no-argument constructor
In the no-argument constructor
In the 1-argument constructor
In the destructor
In the 1-argument constructor
```

4 OBJECTS WILL BE CONSTRUCTED WITH THE NO-ARGUMENT CONSTRUCTOR (BY DEFAULT)

```
Element #0 has value Janani
Element #1 has value Swetha
Element #2 has value Navdeep
Element #3 has value Vitthal
Element #4 has value Ahmad
In the destructor
In the destructor
In the destructor
In the destructor
In the destructor
```

WHAT WILL THIS CODE PRINT?

```
vector<Person> personVector(size);
personVector[0] = Person("Janani");
personVector[1] = Person("Swetha");
personVector[2] = Person("Navdeep");
personVector[3] = Person("Vitthal");

personVector.push_back(Person("Ahmad"));
cout<<personVector.capacity()<<std::endl;
cout<<personVector.size()<<endl;
}
```

```
In the no-argument constructor
In the no-argument constructor
In the no-argument constructor
In the no-argument constructor
In the 1-argument constructor
In the destructor
In the 1-argument constructor
In the destructor
In the 1-argument constructor
In the destructor
In the 1-argument constructor
In the destructor
In the 1-argument constructor
uctor
uctor
uctor
uctor
uctor

as value Janani
as value Swetha
as value Navdeep
as value Vitthal
as value Ahmad
uctor
uctor
uctor
uctor
In the destructor
```

THE ORIGINAL OBJECTS ARE DESTRUCTED AND THE NEW OBJECTS CREATED, THIS TIME WITH THE 1 ARGUMENT CONSTRUCTR.

WHAT WILL THIS CODE PRINT?

```
vector<Person> personVector(size);
personVector[0] = Person("Janani");
personVector[1] = Person("Swetha");
personVector[2] = Person("Navdeep");
personVector[3] = Person("Vitthal");

personVector.push_back(Person("Ahmad"));
cout<<personVector.capacity()<<endl;
cout<<personVector.size()<<endl;

fo

 <

}

}
```

```
In the no-argument constructor
In the no-argument constructor
In the no-argument constructor
In the no-argument constructor
In the 1-argument constructor
In the destructor
In the 1-argument constructor
In the destructor
In the 1-argument constructor
In the destructor
In the 1-argument constructor
In the destructor
In the 1-argument constructor
r
r
r
r
r


alue Janani
alue Swetha
alue Navdeep
alue Vitthal
alue Ahmad
r
r
r
r
In the destructor
```

1 MORE OBJECT IS CREATED AND ADDED TO THE VECTOR

WHAT WILL THIS CODE PRINT?

```
vector<Person> personVector(size);
personVector[0] = Person("Janani");
personVector[1] = Person("Swetha");
personVector[2] = Person("Navdeep");
personVector[3] = Person("Vitthal");


personVector.push_back(Person("Ahmad"));
cout<<personVector.capacity()<<std::endl;
cout<<personVector.size()<<endl;

for(int i=0; i<personVector.size(); ++i){
  cout << "Element #" << i << " has value "
 << personVector[i].getName() << endl;
}

}
```

PRINT THE CAPACITY, THE SIZE, AND VALUE OF EACH ELEMENT

```
In the no-argument constructor
In the destructor
8
5
Element #0 has value Janani
Element #1 has value Swetha
Element #2 has value Navdeep
Element #3 has value Vitthal
Element #4 has value Ahmad
In the destructor
In the destructor
In the destructor
In the destructor
In the destructor
```

WHAT WILL THIS CODE PRINT?

```
vector<Person> personVector(size);
personVector[0] = Person("Janani");
personVector[1] = Person("Swetha");
personVector[2] = Person("Navdeep");
personVector[3] = Person("Vitthal");


personVector.push_back(Person("Ahmad"));
cout<<personVector.capacity()<<std::endl;
cout<<personVector.size()<<endl;

for(int i=0; i<personVector.size(); ++i){
  cout << "Element #" << i << " has value "
 << personVector[i].getName() << endl;
}


}
```

THE VECTOR GOES OUT OF SCOPE, AND DESTRUCTORS ARE CALLED

```
In the no-argument constructor
In the destructor
8
5
Element #0 has value Janani
Element #1 has value Swetha
Element #2 has value Navdeep
Element #3 has value Vitthal
Element #4 has value Ahmad
In the destructor
In the destructor
In the destructor
In the destructor
In the destructor
```

EXAMPLE 68: UNDERSTAND THE BASICS OF ITERATORS

EXAMPLE 68: UNDERSTAND THE BASICS OF ITERATORS

IN THE PREVIOUS EXAMPLE, WE ITERATED OVER ALL OF THE ELEMENTS IN A VECTOR LIKE THIS

```
for(int i=0; i<personVector.size(); ++i){
  cout << "Element #" << i << " has value "
 << personVector[i].getName() << endl;
}
```

WE RELIED ON THE FACT THAT WE CAN ACCESS VECTOR ELEMENTS USING THE [] OPERATOR

EXAMPLE 68: UNDERSTAND THE BASICS OF ITERATORS

IN THE PREVIOUS EXAMPLE, WE ITERATED OVER ALL OF THE ELEMENTS IN A VECTOR LIKE THIS

```
for(int i=0; i<personVector.size(); ++i){
    cout << "Element " << i << " value "
    << personVector[i].getName() << endl;
}
```

THIS IS NOT THE IDEAL WAY OF ITERATING OVER ELEMENTS OF AN STL CONTAINER

WE RELIED ON THE FACT THAT WE CAN ACCESS VECTOR ELEMENTS USING THE [] OPERATOR

EXAMPLE 68: UNDERSTAND THE BASICS OF ITERATORS

IN THE PREVIOUS EXAMPLE, WE ITERATED OVER ALL OF THE ELEMENTS IN A VECTOR LIKE THIS

```
for(int i=0; i<personVector.size(); ++i){
  cout << "Element #" << i << " has value "
 << personVector[i].getName() << endl;
}
```

WE RELIED ON THE FACT THAT WE CAN ACCESS VECTOR ELEMENTS USING THE [] OPERATOR

THE STL HAS AN ELABORATE SET OF ITERATOR CLASSES THAT SHOULD BE USED INSTEAD.

```
vector<Person>::iterator personIterator;


for(personIterator = personVector.begin();
    personIterator != personVector.end();
    personIterator++)
  {
    cout << personIterator->getName() << endl;
    cout << (*personIterator).getName() << endl;
  }
```

```
vector<Person>::iterator personIterator;
```

TO GET AN ITERATOR OVER A VECTOR CONTAINING PERSON OBJECTS -

```
for(personIterator = personVector.begin();
    personIterator != personVector.end();
    personIterator++)
  {
    cout << personIterator->getName() << endl;
    cout << (*personIterator).getName() << endl;
  }
```

```
vector<Person>::iterator personIterator;
```

TO GET AN ITERATOR OVER A VECTOR CONTAINING PERSON OBJECTS -

```
vector<Person>::iterator personIterator;
```

TO GET AN ITERATOR OVER A VECTOR CONTAINING PERSON OBJECTS -

```
vector<Person>::iterator personIterator;
```

TO GET AN ITERATOR OVER A VECTOR CONTAINING PERSON OBJECTS -

```
vector<Person>::iterator personIterator;

for(personIterator = personVector.begin();
    personIterator != personVector.end();
    personIterator++)
{
  cout << personIterator->getName() << endl;
  cout << (*personIterator).getName() << endl;
}
```

A SPECIAL ITERATOR MARKING THE START OF THE CONTAINER

```
vector<Person>::iterator personIterator;


for(personIterator = personVector.begin();
    personIterator != personVector.end();
    personIterator++)
  {
    cout << (*personIterator).getName() << endl;
  }
```

A SPECIAL ITERATOR MARKING THE END OF THE CONTAINER

```
vector<Person>::iterator personIterator;


for(personIterator = personVector.begin();
    personIterator != personVector.end();
    personIterator++)
{
  cout << personIterator->getName() << endl;
  cout << (*personIterator).getName() << endl;
}
```

AN ITERATOR CAN BE INCREMENTED JUST LIKE A LOOP VARIABLE

```
vector<Person>::iterator personIterator;

for(personIterator = personVector.begin();
    personIterator != personVector.end();
    personIterator++)
  {
    cout << personIterator->getName() << endl;
    cout << (*personIterator).getName() << endl;
  }
```

JUST ANOTHER FOR LOOP, BUT ENTIRELY USING ITERATORS!!

TO GET THE CURRENT VALUE OF THE ITERATOR, DEREFERENCE IT LIKE YOU WOULD A POINTER!

```
vector<Person>::iterator personIterator;


for(personIterator = personVector.begin();
    personIterator != personVector.end();
    personIterator++)
  {
    cout << personIterator->getName() << endl;
    cout << (*personIterator).getName() << endl;
  }
```

TO GET THE CURRENT VALUE OF THE ITERATOR, DEREFERENCE IT LIKE YOU WOULD A POINTER!

```
vector<Person>::iterator personIterator;


for(personIterator = personVector.begin();
    personIterator != personVector.end();
    personIterator++)
  {
    cout << personIterator->getName() << endl;
    cout << (*personIterator).getName() << endl;
  }
```

BTW, TO ITERATE OVER A CONST VECTOR, JUST USE A CONST ITERATOR!

```
vector<Person>::iterator personIterator;

vector<Person>::const_iterator personIterator;
```

THIS IS GENERALLY TRUE - USE A CONST ITERATOR FOR A CONST CONTAINER :-)

ITERATORS ARE USED IN ALL SORTS OF FANCY WAYS IN THE STL

```
personVector.erase(personVector.begin(),personVector.end());
```

THIS IS HOW A HARDCORE STL PROGRAMMER WOULD EMPTY A VECTOR - USING 2 ITERATORS

ITERATORS ARE USED IN ALL SORTS OF FANCY WAYS IN THE STL

WE'VE ONLY SCRATCHED THE SURFACE (AS WITH ALL OF THE STL)

WE'VE ONLY SCRATCHED THE SURFACE (AS WITH ALL OF THE STL)

IN ADDITION TO THE REGULAR (STANDARD) ITERATORS WE JUST SAW -

STREAM ITERATORS: TREAT FILES AND STREAMS AS IF THEY WERE ITERATORS - ALLOWING FILES AND I/O DEVICES AS ARGUMENTS TO ALGORITHMS.

ITERATOR ADAPTORS: REVERSE ITERATORS (THEY WALK BACKWARDS RATHER THAN FORWARD), RAW MEDIA ITERATORS (NEVER MIND WHAT THESE ARE:-))

EXAMPLE 69: UNDERSTAND THE MAP, AN ASSOCIATIVE CONTAINER

EXAMPLE 69: UNDERSTAND THE MAP, AN ASSOCIATIVE CONTAINER

THE MAP OFFERS A SIMPLE WAY TO STORE KEY-VALUE PAIRS, AND QUICKLY FIND THE VALUE CORRESPONDING TO A KEY

(ASSOCIATIVE CONTAINERS LIKE MAPS ARE EXTREMELY POPULAR IN JAVA, C# AND PYTHON)

THE MAP OFFERS A SIMPLE WAY TO STORE KEY-VALUE PAIRS, AND QUICKLY FIND THE VALUE CORRESPONDING TO A KEY

THE MAP OFFERS A SIMPLE WAY TO STORE KEY-VALUE PAIRS, AND QUICKLY FIND THE VALUE CORRESPONDING TO A KEY

```
map <string,string> personMap;

personMap.insert(std::pair<string,string>("Janani","Ravi"));
personMap.insert(std::pair<string,string>("Swetha","Kolalapudi"));
personMap["Navdeep"] = "Singh";
personMap["Vitthal"] = "Srinivasan";
```

THE MAP OFFERS A SIMPLE WAY TO STORE KEY-VALUE PAIRS, AND QUICKLY FIND THE VALUE CORRESPONDING TO A KEY

```
map <string,string> personMap;
```

2 TEMPLATE PARAMETERS, 1 FOR KEY-TYPE AND 1 FOR VALUE-TYPE

```
personMap.insert(std::pair<string,string>("Sudhanshu","Gaddi"));
personMap["Navdeep"] = "Singh";
personMap["Vitthal"] = "Srinivasan";
```

THE MAP OFFERS A SIMPLE WAY TO STORE KEY-VALUE PAIRS, AND QUICKLY FIND THE VALUE CORRESPONDING TO A KEY

```
map <string,string> personMap;

personMap.insert(std::pair<string,string>("Janani","Ravi"));
personMap.insert(std::pair<string,string>("Swetha","Kotalapudi"));
personMap["Navaneeth"] = "Kumar";
personMap["Vitthal"] = "Srinivasan";
```

INSERT A KEY-VALUE PAIR

THE MAP OFFERS A SIMPLE WAY TO STORE KEY-VALUE PAIRS, AND QUICKLY FIND THE VALUE CORRESPONDING TO A KEY

```
map <string,string> personMap;

personMap.insert(std::pair<string,string>("Janani","Ravi"));
personMap.insert(std::pair<string,string>("Swetha","Kolalapudi"));
personMap["Vidhya"] = "Singh";
personMap["Vittal"] = "Srinivasan";
```

PAIR ITSELF IS AN STL CONTAINER

THE MAP OFFERS A SIMPLE WAY TO STORE KEY-VALUE PAIRS, AND QUICKLY FIND THE VALUE CORRESPONDING TO A KEY

```
map <string,string> personMap;

personMap.insert(std::pair<string,string>("Janani","Ravi"));
personMap.insert(std::pair<string,string>("Swetha","Kolalapudi"));
personMap["Vidya"] = "Singh";
personMap["Vitthal"] = "Srinivasan";
```

KEY TYPE AND VALUE TYPE (AGAIN)

THE MAP OFFERS A SIMPLE WAY TO STORE KEY-VALUE PAIRS, AND QUICKLY FIND THE VALUE CORRESPONDING TO A KEY

CAN ALSO INSERT USING THE [] OPERATOR

```
personMap["Navdeep"] = "Singh";
```

THE MAP OFFERS A SIMPLE WAY TO STORE KEY-VALUE PAIRS, AND QUICKLY FIND THE VALUE CORRESPONDING TO A KEY

```
if(personMap.find("Ahmad") == personMap.end())
{
  std::cout<<"Ahmad is not in the map!"<<endl;
}
```

THE MAP OFFERS A SIMPLE WAY TO STORE KEY-VALUE PAIRS, AND QUICKLY FIND THE VALUE CORRESPONDING TO A KEY

```
if(personMap.find("Ahmad") == personMap.end())
{
   std::cout<<"Ahmad is not in the map!"<<endl;
}
```

USE THE FIND METHOD TO GET AN ITERATOR POINTING TO THE ELEMENT

THE MAP OFFERS A SIMPLE WAY TO STORE KEY-VALUE PAIRS, AND QUICKLY FIND THE VALUE CORRESPONDING TO A KEY

```
if(personMap.find("Ahmad") == personMap.end())
{
   std::cout<<"Ahmad is not in the map!"<<endl;
}
```

IF THE KEY IS NOT PRESENT, THE ITERATOR HAS THE SPECIAL VALUE "END"

MAP ITERATORS HAVE TWO VALUES - CALLED FIRST AND SECOND

MAP ITERATORS HAVE TWO VALUES - CALLED FIRST AND SECOND

```
vector<string> personVector;
vector<string> lastNameVector;
for(map<string,string>::iterator
     mapIterator = personMap.begin();
     mapIterator != personMap.end();
    ++mapIterator)
{
  personVector.push_back(mapIterator->first);
  lastNameVector.push_back(mapIterator->second);
}
```

MAP ITERATORS HAVE TWO VALUES - CALLED FIRST AND SECOND

```
vector<string> personVector;
vector<string> lastNameVector;
for(map<string,string>::iterator
    mapIterator = personMap.begin();
    mapIterator != personMap.end();
    ++mapIterator)
{
  personVector.push_back(mapIterator->first);
  lastNameVector.push_back(mapIterator->second);
}
```

SPECIFY THE TYPE OF MAP THAT WE WANT AN ITERATOR FOR

MAP ITERATORS HAVE TWO VALUES - CALLED FIRST AND SECOND

```
vector<string> personVector;
vector<string> lastNameVector;
for(map<string,string>::iterator
    mapIterator = personMap.begin();
    mapIterator != personMap.end();
  ++mapIterator)
{
  personVector.push_back(mapIterator->first);
  lastNameVector.push_back(mapIterator->second);
}
```

THIS IS HOW THE FIRST AND SECOND VALUES IN THE PAIR ARE ACCESSED

EXAMPLE 70: UNDERSTAND THE BASIC IDEA OF STL ALGORITHMS

EXAMPLE 70: UNDERSTAND THE BASIC IDEA OF STL ALGORITHMS

THE STL HAS 3 IMPORTANT SETS OF FUNCTIONALITY

CONTAINERS
CLASSES THAT HOLD STUFF

ITERATORS
CLASSES THAT DO ELEMENT-BY-ELEMENT STUFF WITH CONTAINERS

ALGORITHMS
"FUNCTION OBJECTS", CLASSES THAT ENCAPSULATE ALGORITHMS

ALGORITHMS

"FUNCTION OBJECTS", CLASSES THAT ENCAPSULATE ALGORITHMS

```
find    count    sort

search    merge    for_each

transform
```

ALGORITHMS

"FUNCTION OBJECTS", CLASSES THAT ENCAPSULATE ALGORITHMS

```
find    count    sort

search    merge    for_each

transform
```

find

```
vector<string> names;
names.push_back("Janani");
names.push_back("Ahmad");
names.push_back("Swetha");
names.push_back("Navdeep");
names.push_back("Vitthal");
names.push_back("Ahmad");

vector<string>::iterator findIt = find(names.begin(), names.end(), "Vitthal");
cout << "Find algorithm" << endl << endl;
if(findIt == names.end())
  cout << "Not found!!" << endl;
else
   cout << "First object with value Vitthal found at offset "
     << (findIt-names.begin()) << endl;
```

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
Find algorithm

First object with value Vitthal found at offset 4
```

find

```
vector<string> names;
names.push_back("Janani");
names.push_back("Ahmad");
names.push_back("Swetha");
names.push_back("Navdeep");
names.push_back("Vitthal");
names.push_back("Ahmad");
```

```
vector<string>::iterator findIt = find(names.begin(), names.end(), "Vitthal");
cout << "Find algorithm" << endl << endl;
if(findIt == names.end())
  cout << "Not found!!" << endl;
else
   cout << "First object with value Vitthal found at offset "
     << (findIt-names.begin()) << endl;
```

SAY WE HAVE A VECTOR OF STRINGS

```
[Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
 Find algorithm

 First object with value Vitthal found at offset 4
```

find

```
vector<string> names;
names.push_back("Janani");
names.push_back("Ahmad");
names.push_back("Swetha");
names.push_back("Navdeep");
names.push_back("Vitthal");
names.push_back("Ahmad");

vector<string>::iterator findIt = find(names.begin(), names.end(),
"Vitthal");
```

AND WE WANT TO FIND THE POSITION OF A SPECIFIC VALUE IN THAT LIST

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
Find algorithm

First object with value Vitthal found at offset 4
```

find

```
vector<string> names;
names.push_back("Janani");
names.push_back("Ahmad");
names.push_back("Swetha");
names.push_back("Navdeep");
names.push_back("Vitthal");
names.push_back("Ahmad");

vector<string>::iterator findIt = find(names.begin(), names.end(),
"Vitthal");
cout << "Find algorithm" << endl << endl;
if(findIt == names.end())
    cout << "No found" << endl;
else
    cout << "First object with value Vitthal found at offset "
        << (findIt - names.begin()) << endl;
```

HERE FIND LOOKS LIKE A FUNCTION, BUT ITS ACTUALLY AN OBJECT (A FUNCTION OBJECT IN FACT :-))

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
Find algorithm

First object with value Vitthal found at offset 4
```

find

```
vector<string> names;
names.push_back("Janani");
names.push_back("Ahmad");
names.push_back("Swetha");
names.push_back("Navdeep");
names.push_back("Vitthal");
names.push_back("Ahmad");

vector<string>::iterator findIt = find(names.begin(), names.end(), "Vitthal");
```

LIKE MOST ALGORITHM OBJECTS IT DEALS MOSTLY WITH ITERATORS

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
Find algorithm

First object with value Vitthal found at offset 4
```

find

```
vector<string> names;
names.push_back("Janani");
names.push_back("Ahmad");
names.push_back("Swetha");
names.push_back("Navdeep");
names.push_back("Vitthal");
names.push_back("Ahmad");

vector<string>::iterator findIt = find(names.begin(), names.end(), "Vitthal");
cout << "Find algorithm" << endl << endl;
if(findIt == names.end())
  cout << "Not found" << endl;
else
  cout << "First object with value Vitthal found at offset "
       << (findIt - names.begin()) << endl;
```

LIKE MOST ALGORITHM OBJECTS IT DEALS MOSTLY WITH ITERATORS

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
Find algorithm

First object with value Vitthal found at offset 4
```

find

```
vector<string> names;
names.push_back("Janani");
names.push_back("Ahmad");
names.push_back("Swetha");
names.push_back("Navdeep");
names.push_back("Vitthal");
names.push_back("Ahmad");

vector<string>::iterator findIt = find(names.begin(), names.end(), "Vitthal");
cout << "Find algorithm" << endl << endl;
if(findIt == names.end())
  cout << "Not found" << endl;
else
  cout << "First object with value Vitthal found at offset "
       << (findIt - names.begin()) << endl;
```

LIKE MOST ALGORITHM OBJECTS IT DEALS MOSTLY WITH ITERATORS

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
Find algorithm

First object with value Vitthal found at offset 4
```

find

IF IT DOES NOT FIND THE VALUE, IT RETURNS THE END ITERATOR

```
vector<string> names;
names.push_back("Janani");
names.push_back("Dheeraj");
names.push_back("Bhanu");
names.push_back("Navdeep");
names.push_back("Siddhartha");
names.push_back("Ambuj");

vector<string>::iterator findIt = find(names.begin(), names.end(), "Vitthal");
cout << "Find algorithm" << endl << endl;

if(findIt == names.end())
  cout << "Not found!!" << endl;
else
   cout << "First object with value Vitthal found at offset "
     << (findIt-names.begin()) << endl;
```

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
Find algorithm

First object with value Vitthal found at offset 4
```

find

ELSE YOU CAN FIND THE POSITION OF THE VALUE USING ITERATOR ARITHMETIC

```
(findIt-names.begin())
```

```
Vitthals-MacBook-Pro:~ vitthalsrinivasan$ ./a.out
Find algorithm

First object with value Vitthal found at offset 4
```

ALGORITHMS

"FUNCTION OBJECTS", CLASSES THAT ENCAPSULATE ALGORITHMS

```
find    count    sort

search    merge    for_each

transform
```

ALGORITHMS
"FUNCTION OBJECTS", CLASSES THAT ENCAPSULATE ALGORITHMS

THE CODE ATTACHED WITH THIS EXAMPLE USES ALL OF THESE OPERATORS, BUT LET'S FOCUS ON `transform`, BECAUSE IT DOES SOME INTERESTING STUFF

find count sort

search merge for_each

transform

ALGORITHMS

"FUNCTION OBJECTS", CLASSES THAT ENCAPSULATE ALGORITHMS

`find` `count` `sort`

`search` `merge` `for_each`

`transform`

transform

```
string someString = "Vitthal";
cout << "Initial value: " << someString << endl;
transform(someString.begin(), someString.end(), someString.begin(), ::toupper);
cout << "Final value: " << someString << endl;
```

HERE WE USE THE TRANSFORM ALGORITHM OBJECT TO CONVERT A STRING TO UPPERCASE

```
Initial value: Vitthal
Final value: VITTHAL
```

HERE WE USE THE TRANSFORM ALGORITHM OBJECT TO CONVERT A STRING TO UPPERCASE

```
string someString = "Vitthal";
cout << "Initial value: " << someString << endl;
transform(someString.begin(), someString.end(), someString.begin(), ::toupper);
cout << "Final value: " << someString << endl;
```

FIRST - REMEMBER THAT WE CAN APPLY ALGORITHM OBJECTS ON NON-STL TYPES TOO (STRINGS, POINTERS, ANYTHING)

```
Initial value: Vitthal
Final value: VITTHAL
```

HERE WE USE THE TRANSFORM ALGORITHM OBJECT TO CONVERT A STRING TO UPPERCASE

```
string someString = "Vitthal";
cout << "Initial value: " << someString << endl;
transform(someString.begin(), someString.end(), someString.begin(), ::toupper);
cout << "Final value: " << someString << endl;
```

THE TRANSFORM ALGORITHM TAKES IN A FUNCTION POINTER

```
Initial value: Vitthal
Final value: VITTHAL
```

HERE WE USE THE TRANSFORM ALGORITHM OBJECT TO CONVERT A STRING TO UPPERCASE

THE TRANSFORM ALGORITHM TAKES IN A FUNCTION POINTER

```
string someString = "Vitthal";
cout << "Initial value: " << someString << endl;
transform(someString.begin(), someString.end(), someString.begin(), ::toupper);
cout << "Final value: " << someString << endl;
```

HERE, IT IS TO THE C++ STANDARD FUNCTION CALLED **::toupper**

```
Initial value: Vitthal
Final value: VITTHAL
```

HERE WE USE THE TRANSFORM ALGORITHM OBJECT TO CONVERT A STRING TO UPPERCASE

THE TRANSFORM ALGORITHM TAKES IN A FUNCTION POINTER

```
string someString = "Vitthal";
cout << "Initial value: " << someString << endl;
transform(someString.begin(), someString.end(), someString.begin(), ::toupper);
cout << "Final value: " << someString << endl;
```

HERE, IT IS TO THE C++ STANDARD FUNCTION CALLED `::toupper`

`::toupper` TAKES IN 1 CHARACTER AND CONVERTS IT TO UPPER CASE

```
Initial value: Vitthal
Final value: VITTHAL
```

HERE WE USE THE TRANSFORM ALGORITHM OBJECT TO CONVERT A STRING TO UPPERCASE

```
string someString = "Vitthal";
cout << "Initial value: " << someString << endl;
transform(someString.begin(), someString.end(), someString.begin(), ::toupper);
cout << "Final value: " << someString << endl;
```

THE TRANSFORM ALGORITHM TAKES IN A FUNCTION POINTER

```
Initial value: Vitthal
Final value: VITTHAL
```

HERE WE USE THE TRANSFORM ALGORITHM OBJECT TO CONVERT A STRING TO UPPERCASE

```
string someString = "Vitthal";
cout << "Initial value: " << someString << endl;
transform(someString.begin(), someString.end(), someString.begin(), ::toupper);
cout << "Final value: " << someString << endl;
```

AND APPLIES THAT FUNCTION..

```
Initial value: Vitthal
Final value: VITTHAL
```

HERE WE USE THE TRANSFORM ALGORITHM OBJECT TO CONVERT A STRING TO UPPERCASE

```
string someString = "Vitthal";
cout << "Initial value: " << someString << endl;
transform(someString.begin(), someString.end(), someString.begin(), ::toupper);
cout << "Final value: " << someString << endl;
```

TO A RANGE BETWEEN A STARTING AND ENDING ITERATOR

```
Initial value: Vitthal
Final value: VITTHAL
```

HERE WE USE THE TRANSFORM ALGORITHM OBJECT TO CONVERT A STRING TO UPPERCASE

```
string someString = "Vitthal";
cout << "Initial value: " << someString << endl;
transform(someString.begin(), someString.end(), someString.begin(), ::toupper);
cout << "Final value: " << someString << endl;
```

TO A RANGE BETWEEN A STARTING AND ENDING ITERATOR

```
Initial value: Vitthal
Final value: VITTHAL
```

HERE WE USE THE TRANSFORM ALGORITHM OBJECT TO CONVERT A STRING TO UPPERCASE

```
string someString = "Vitthal";
cout << "Initial value: " << someString << endl;
transform(someString.begin(), someString.end(), someString.begin(), ::toupper);
cout << "Final value: " << someString << endl;
```

TO A RANGE BETWEEN A STARTING AND ENDING ITERATOR

```
Initial value: Vitthal
Final value: VITTHAL
```

HERE WE USE THE TRANSFORM ALGORITHM OBJECT TO CONVERT A STRING TO UPPERCASE

```
string someString = "Vitthal";
cout << "Initial value: " << someString << endl;
transform(someString.begin(), someString.end(), someString.begin(), ::toupper);
cout << "Final value: " << someString << endl;
```

TO A RANGE BETWEEN A STARTING AND ENDING ITERATOR

```
Initial value: Vitthal
Final value: VITTHAL
```

HERE WE USE THE TRANSFORM ALGORITHM OBJECT TO CONVERT A STRING TO UPPERCASE

```
string someString = "Vitthal";
cout << "Initial value: " << someString << endl;
transform(someString.begin(), someString.end(), someString.begin(), ::toupper);
cout << "Final value: " << someString << endl;
```

AND COPIES THE OUTPUT OF THESE FUNCTION CALLS..

```
Initial value: Vitthal
Final value: VITTHAL
```

HERE WE USE THE TRANSFORM ALGORITHM OBJECT TO CONVERT A STRING TO UPPERCASE

```
string someString = "Vitthal";
cout << "Initial value: " << someString << endl;
transform(someString.begin(), someString.end(), someString.begin(), ::toupper);
cout << "Final value: " << someString << endl;
```

..TO SOME AREA SPECIFIED (WHICH ALSO IS USUALLY AN ITERATOR)

```
Initial value: Vitthal
Final value: VITTHAL
```

transform

```
string someString = "Vitthal";
cout << "Initial value: " << someString << endl;
transform(someString.begin(), someString.end(), someString.begin(), ::toupper);
cout << "Final value: " << someString << endl;
```

HERE WE USE THE TRANSFORM ALGORITHM OBJECT TO CONVERT A STRING TO UPPERCASE

```
Initial value: Vitthal
Final value: VITTHAL
```

ALGORITHMS

"FUNCTION OBJECTS", CLASSES THAT ENCAPSULATE ALGORITHMS

```
find    count    sort

search    merge    for_each

transform
```

ALGORITHMS

"FUNCTION OBJECTS", CLASSES THAT ENCAPSULATE ALGORITHMS

WE'VE ONLY SCRATCHED THE SURFACE OF ALGORITHMS, AND REALLY OF THE STL.

find count sort

search merge for_each

transform

WE'VE ONLY SCRATCHED THE SURFACE OF ALGORITHMS, AND REALLY OF THE STL.

BUT WE'VE GOT A GOOD SENSE OF WHAT ITS ALL ABOUT.