```
class stack {
    int     arr[2], top ;
public : stack( ){top = -1 ;}
    class full{ } ; class empty{ } ;
    void push(int i) {
        if (top == 1) throw full() ;
        top++ ;arr[top] = i; }
    int pop( ) {
        if(top == -1) throw empty() ;
        else {top-- ; return 1 ;} }
} ;
int main( ) {
    stack   s ;
    try{ s.push(1) ;cout<<"1\n" ;
    s.push(2) ;cout<<"2\n" ;
    s.push(3) ;cout<<"3\n" ;
    } catch(stack::full f){cout<<"full\n" ;} }
```

OUTPUT :

```
1
2
full
```

ade

- **If an exception is thrown before the constructor completes execution then the associated destructor will not be called for that object.**

- **When an exception is thrown, a destructor is automatically called for any object that was created by the code up to that point in the try block.**

- Exceptions impose an overhead in terms of program size and (when an exception occurs) in time. So we should not try to overuse it.

- An exception is caught by specifying its type.
- What is thrown is not a type but an object.
- If we need to transmit extra information from the *throw* point to the handler, we can do so putting data into that object.

Example :

```
class Vector {
      int sz ; int *p ;
      public :  class Range {
                        public  : int index ;
                        Range(int i) : index(i) { } } ;
                  Vector (int s) : sz(s) { /* initialize p */}
                  int & operator[ ] (int i) {
                        if (0 <= i && i < sz) return p[i] ;
                        throw Range(i) ; }} ;
void do_something(Vector& v) {}
void f(Vector &v) {
// ...
try { do_something(v) ;}
catch (Vector :: Range r) {cerr<<"bad index ::"<<r.index<<endl ;}
} ;
```

The following example shows that if an exception is thrown before the constructor completes execution then the associated destructor will not be called for that object.

```
class Cl {
public :  class exc{} ;
     Cl( ) {    cout<<"in constr\n" ;
          throw exc() ;
          cout<<"exception thrown\n" ; }

     ~Cl( ) {  cout<<"in destr\n" ; }
} ;

main( )
{    try
     { Cl obj ; }
     catch(Cl::exc  E) {cout<<"exception caught" ;}
}
```

OUTPUT

```
in constr
exception caught
```

The following example shows that when an exception is thrown, a destructor is automatically called for any object that was created by the code up to that point in the try block.

```
class Cl {
public :  class exc{} ;
     Cl() {    cout<<"in constr\n" ;
          throw exc() ;
          cout<<"exception thrown\n" ; }
     ~Cl() {  cout<<"in destr\n" ; } } ;

class CC {
public :  CC() {cout<<"constr of CC\n" ;}
     ~CC() {cout<<"destr of CC\n" ;} } ;

main() {  try
     {    CC  o1 ;
          Cl obj ; }
     catch(Cl::exc  E) {cout<<"exception caught" ;}
}
```

OUTPUT

```
constr of CC
in constr
destr of CC
exception caught
```

ade

A destructor is called during stack unwinding resulting from an exception being thrown. If the destructor itself throws an exception, having been called as the result of an exception being thrown, then the function std::terminate() is called with the default effect of calling std::abort(). Hence, destructors must satisfy the no-throw guarantee, that is, they must not throw an exception if they themselves have been called as the result of an exception being thrown.

```
class A {
  public :
  A() {cout<<"constr\n" ;}
  ~A() {
     cout<<"destr #1\n" ;
     if(/*error cond*/)    throw int(99) ;
     cout<<"destr #2\n" ;
  }
} ;
main() {
  try {
     A aobj ;
     if(/*error cond*/)    throw char('s') ;
  }
  catch(int ch) {cout<<"exc::"<<ch<<"\n" ;}
  catch(char ch) {cout<<"exc::"<<ch<<"\n" ;} //the order of the these catches would not matter
}
```

A destructor is called during stack unwinding resulting from an exception being thrown. If the destructor itself throws an exception, having been called as the result of an exception being thrown, then the function std::terminate() is called with the default effect of calling std::abort(). Hence, destructors must satisfy the no-throw guarantee, that is, they must not throw an exception if they themselves have been called as the result of an exception being thrown.

```
class A {
  public :
  A() {cout<<"constr\n" ;}
  ~A() {
     cout<<"destr #1\n" ;
     if(/*error cond*/)   throw int(99) ;
     cout<<"destr #2\n" ;
  }
} ;
main() {
  try {
     A aobj ;
     if(/*error cond*/)   throw char('s') ;
  }
  catch(int ch) {cout<<"exc::"<<ch<<"\n" ;}
  catch(char ch) {cout<<"exc::"<<ch<<"\n" ;} //the order of the these catches would not matter
}
```

```
class full { public: void disp() {cout<<"full\n" ;}} ;
class BigNum { public: void disp() {cout<<"tooooooooo big num\n" ;}} ;
class stack {  int          arr[2], top ;
         public :  stack( ){top = -1 ;}
         void push(int i) {
              if (i > 100) throw  BigNum() ;
              if (top == 1) throw full() ;
              else  {top++ ;arr[top] = i;} } } ;

void f1(stack s) {
     try{ s.push(101) ;cout<<"1" ;
          s.push(2) ;cout<<"2" ;
          s.push(3) ;cout<<"3" ;
          }
     catch(BigNum  big){big.disp();}}

main( ) {       stack     s ;
          try { f1(s) ;}
          catch(full f){ f.disp();}
}
```

push throws two exceptions one of them is handled in main and the other is handled in f1()

ade

USAGE of *bad_exception*

```
#include <iostream>
#include <fstream>
#include <exception>
using namespace std ;
class X{} ;     class Y{} ;
void unexpectedExcHandler() {throw ;}
void f(int x) throw (int, X, bad_exception) {
   if(x==2) throw int(8) ;
   if(x==3) throw X();
   if(x==4) throw Y() ;//double(8) ;
}
main() {
   set_unexpected(unexpectedExcHandler) ;
   try{
      cout<<"1---\n" ;f(1) ;
      cout<<"2---\n" ;//f(2) ;
      cout<<"3---\n" ;//f(3) ;
      cout<<"4---\n" ;f(4) ;
      cout<<"5---\n" ;
   }
   catch(int i) {cout<<"int exc\n"  ;}
   catch(bad_exception e) {cout<<"bad exc\n" ;}
   catch(Y e) {cout<<"Y\n" ;}
   catch(...) {cout<<"def\n" ;} }
```

```
class stack {
    int     arr[2], top ;
public : stack( ){top = -1 ;}
    class full{ } ; class empty{ } ;
    void push(int i) {
        if (top == 1) throw full() ;
        top++ ;arr[top] = i; }
    int pop( ) {
        if(top == -1) throw empty() ;
        else {top-- ; return 1 ;} }
} ;
int main( ) {
    stack   s ;
    try{ s.push(1) ;cout<<"1\n" ;
    s.push(2) ;cout<<"2\n" ;
    s.push(3) ;cout<<"3\n" ;
    } catch(stack::full f){cout<<"full\n" ;} }
```

OUTPUT :

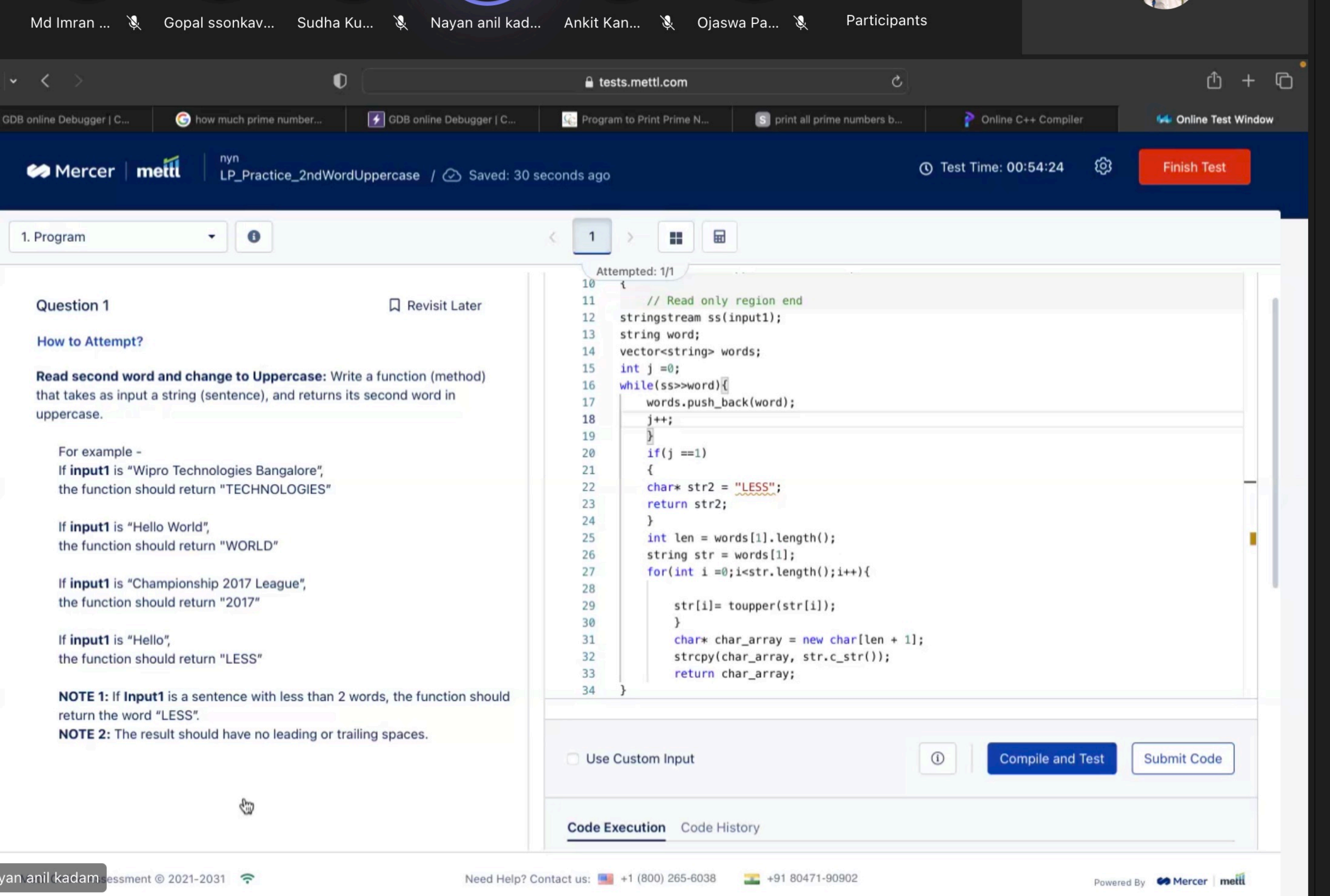
```
1
2
full
```

- Exceptions refer to unusual conditions in the program.

- The purpose of exception handling is to provide means to detect and report errors such that necessary action can be taken.

- This mechanism involves the following tasks :
  - Find the problem – *hit* the exception
  - Inform that an exception has occurred – *throw* the exception
  - Receive the error information – *catch* the exception
  - Take corrective actions - *handle* the exception

- Keywords used :
  - **throw**
  - **try**
  - **catch**



Question 1

Revisit Later

How to Attempt?

Read second word and change to Uppercase: Write a function (method) that takes as input a string (sentence), and returns its second word in uppercase.

For example -
If **input1** is "Wipro Technologies Bangalore",
the function should return "TECHNOLOGIES"

If **input1** is "Hello World",
the function should return "WORLD"

If **input1** is "Championship 2017 League",
the function should return "2017"

If **input1** is "Hello",
the function should return "LESS"

NOTE 1: If **Input1** is a sentence with less than 2 words, the function should return the word "LESS".
NOTE 2: The result should have no leading or trailing spaces.

```
{
    // Read only region end
stringstream ss(input1);
string word;
vector<string> words;
int j =0;
while(ss>>word){
    words.push_back(word);
    j++;
    }
    if(j ==1)
    {
    char* str2 = "LESS";
    return str2;
    }
    int len = words[1].length();
    string str = words[1];
    for(int i =0;i<str.length();i++){

        str[i]= toupper(str[i]);
        }
        char* char_array = new char[len + 1];
        strcpy(char_array, str.c_str());
        return char_array;
}
```