```
Name: Muhammad Asim Ali
Student ID: V00854120
Email: maali@uvic.ca
Course: SENG475
Section: T01

Assignment ID: cpp_compile_time
Assignment Title: Compile-Time Computation

Submission Source: cpp_compile_time-MAsimSENG/
Commit ID: ?

Submitted Files
===============

drwxrwxr-x    4096 2020-06-05 13:53 ./app
-rw-rw-r--    1916 2020-06-05 13:53 ./app/test_cexpr_basic_string.cpp
-rw-rw-r--     682 2020-06-05 13:53 ./CMakeLists.txt
-rw-rw-r--     145 2020-06-05 13:53 ./IDENTIFICATION.txt
drwxrwxr-x    4096 2020-06-05 13:53 ./include
drwxrwxr-x    4096 2020-06-05 13:53 ./include/ra
-rw-rw-r--    7528 2020-06-05 13:53 ./include/ra/cexpr_basic_string.hpp

Results
=======

Package          Operation Target           Status
nonprog          generate  ---              FAIL (1 0.0s 1L)
string_orig      generate  ---              FAIL (1 0.1s 1L)
string_sane      generate  ---              FAIL (1 0.1s 1L)
math_orig        generate  ---              FAIL (1 0.1s 1L)
math_sane        generate  ---              FAIL (1 0.1s 1L)

Normally, an operation is indicated as having a status of either "OK" or
"FAIL".  A status of "?" indicates that the operation could not be performed
for some reason (e.g., due to an earlier error or being a manual step).
The time (in seconds) required for an operation is denoted by an expression
consisting of a number followed by the letter "s" (e.g., "5.0s").
In the case of a test that consists of multiple test cases, the number of
failed test cases and total number of test cases is expressed as a fraction
(e.g., "10/50" means 10 test cases failed out of 50 test cases in total).
The length (in lines) of the log file generated by an operation is denoted by
an expression consisting of a number followed by the letter "L" (e.g., "10L").
To ascertain the reason for the failure of an operation, check the contents
of the log file provided.

Legend
======

Package: nonprog
    Nonprogramming exercises

Package: string_orig
    The code as originally submitted by the student.

Package: string_sane
    Code with modifications to perform API sanity checking.

Package: math_orig
```

```
    The code as originally submitted by the student.

Package: math_sane
    Code with modifications to perform API sanity checking.
```

```
ERROR: missing file/directory README.pdf
```

```
ERROR: missing file/directory include/ra/mandelbrot.hpp
```

```
ERROR: missing file/directory include/ra/mandelbrot.hpp
```

Log: math_orig generate

```
ERROR: missing file/directory include/ra/cexpr_math.hpp
```

Log: math_sane generate

```
ERROR: missing file/directory include/ra/cexpr_math.hpp
```

```
cmake_minimum_required(VERSION 3.1 FATAL_ERROR )

project(string_array LANGUAGES CXX)


set(CMAKE_CXX_STANDARD 17)

set(CMAKE_CXX_STANDARD_REQUIRED ON)
#set(sources include/ra/cexpr_basic_string.hpp)

add_library(lib INTERFACE)

target_sources(lib INTERFACE cexpr_basic_string.hpp)

target_include_directories(lib INTERFACE
    "${PROJECT_SOURCE_DIR}/include/ra/"
)

add_library(Catch INTERFACE)

set(CATCH_INCLUDE_DIR ${CMAKE_CURRENT_SOURCE_DIR}/catch2/)

target_include_directories(Catch INTERFACE ${CATCH_INCLUDE_DIR})


set(test_sources app/test_cexpr_basic_string.cpp)

# Make test executable
add_executable(test ${test_sources})




target_link_libraries(test Catch lib)





```

```
#include <stdio.h>
#include <cstddef>
#include <assert.h>
#include<stdexcept>
#include<iostream>

    /* assert */

using namespace std;
namespace ra::cexpr {

    // A basic string class template for use in constexpr contexts.
    template <class T, std::size_t M>
    class cexpr_basic_string
    {
        // An unsigned integral type used to represent sizes.
         using size_type = std::size_t;

         // The type of each character in the string (i.e., an alias for
         // the template parameter T).
         using value_type = T;


         // The type of a mutating (ie a non const pointer) pointer to each character in the string.
        using pointer = T*;

     // The type of a non-mutating pointer to each character in the
     // string.
     using const_pointer = const T*;

     // The type of a mutating reference to a character in the string.
     using reference = T&;

     // The type of a non-mutating reference to a character in the
     // string.
     using const_reference = const T&;

     // A mutating iterator type for the elements in the string.
     using iterator = pointer;

     // A non-mutating iterator type for the elements in the string.
     using const_iterator = const_pointer;


    private:
        value_type _string [M];

    public:



 // Creates an empty string (i.e., a string containing no
 // characters).

 constexpr cexpr_basic_string(): _string{'\0'}{

 };

 // Explicitly default some special members.
 constexpr cexpr_basic_string(const cexpr_basic_string&) =default;
 constexpr cexpr_basic_string& operator=(const cexpr_basic_string&) = default;
 ~cexpr_basic_string() = default;

 // Creates a string with the contents given by the
 // null-terminated character array pointed to by s.
 // If the string does not have sufficient capacity to hold
 // the character data provided, an exception of type
 // std::runtime_error is thrown.
/*
Checklist for constexpr constructors
1. each of parameters is literal type : (Void, pointer, int,float, reference ...
)
    or class with a trivial destructor and atleast one socntructor which is cons
texpr

*/

 constexpr cexpr_basic_string(const value_type* s) :_string()
 {
     // if M=5 then max value N can have is 6

      int N=0;
     for(int i=0; s[i]!='\0'; i++){
         if(N>M){ throw std::runtime_error("can't fit this into our array");}
         _string[i] = s[i];
         N+=1;
     }
     _string[N] = '\0';
 }

 const value_type* get_string()const {
     return _string;
 }

 // Creates a string with the contents specified by the characters
 // in the iterator range [first, last).
 // If the string does not have sufficient capacity to hold
 // the character data provided, an exception of type
 // std::runtime_error is thrown.
 constexpr cexpr_basic_string(const_iterator first,const_iterator last)
 : _string()
  {
        int N=0;
         // const_iterator = const char *
         for(const T* i=first; i!=last; ++i ){
             if(N>M){ throw std::runtime_error("can't fit this into our array");}

             _string[N]= *i;
             N+=1;
         }
         _string[N]='\0';

     }

 // Returns the maximum number of characters that can be held by a
 // string of this type.
 // The value returned is the template parameter M.
 static constexpr size_type max_size() {
     const int len=sizeof(_string)/sizeof(T);
     return len;
  }

 // Returns the maximum number of characters that the string can
 // hold. The value returned is always the template parameter M.
 constexpr size_type capacity() const {
     return M;
 }

 // Returns the number of characters in the string (excluding the
 // dummy null character).
 constexpr size_type size() const {
     int num=0;
     for(const T * it=&(_string[0]); *it!='\0'; ++it  ) {
         num+=1;
     }
     const int len =num;
     return len;
 };

 // Returns a pointer to the first character in the string.
 // The pointer that is returned is guaranteed to point to a
 // null-terminated character array.
 // The user of this class shall not alter the dummy null
 // character stored at data() + size().
 value_type* data()
{
     int len = this->size();
     int num =0;
    for(int i=0; i<len; ++i) {
            num+=1;
    }
    assert(_string[num]=='\0');

    return &(_string[0]);
}
 const value_type* data() const{
     return _string;
 };

 // Returns an iterator referring to the first character in the
 // string.
 constexpr iterator begin(){
     constexpr iterator it = &(_string[0]);
     return it;
 }
 constexpr const_iterator begin() const{
      const const_iterator it = &(_string[0]);
    return it;
}

 // Returns an iterator referring to the fictitious
 // one-past-the-end character in the string.
 constexpr iterator end(){
     const int len = this->size();
     iterator it = &(_string[len]);
     return it;
 }
 constexpr const_iterator end() const{
     const int len = this->size();
      const_iterator it = &(_string[len]);
    return it;


 }

 // Returns a reference to the i-th character in the string if i
 // is less than the string size; and returns a reference to the
 // dummy null character if i equals the string size.
 // Precondition: The index i is such that i >= 0 and i <= size().
 constexpr reference operator[](size_type i) {
     // size returns full size of array ie arr[size] retunrs dummy char
     if(!(i>=0 && i<=this->size())){ throw std::runtime_error("can't fit this into our array");}
     int string_size = this->size();
     if( i <string_size){
         T& ref = _string[i];
         return ref;
     }
     else {
         T& ref= _string[string_size];
         return ref;
     }


 }
 constexpr const_reference operator[](size_type i) const {

     if(!(i>=0 && i<=this->size())){ throw std::runtime_error("can't fit this into our array");}
     int string_size = this->size();
     if(i <string_size){
         const T& ref = _string[i];
         return ref;
     }
     else {
         const T& ref= _string[string_size];
         return ref;
     }


 }

// Appends (i.e., adds to the end) a single character to the
// string. If the size of the string is equal to the capacity,
// the string is not modified and an exception of type
// std::runtime_error is thrown.
constexpr void push_back(const T& x){
if(this->size()==this->capacity()){ throw std::runtime_error("can't fit this into our array");}
    _string[this->size()] = x;
    _string[this->size() +1 ]='\0';
}

// Erases the last character in the string.
 // If the string is empty, an exception of type std::runtime_error
 // is thrown.
 constexpr void pop_back(){
     if(this->size()==0){throw std::runtime_error("empty string");}
        const int prev_size = this->size();
     _string[prev_size -1]='\0';
     const int now_size = this->size();
     assert(prev_size-1 == now_size);
 }

 // Appends (i.e., adds to the end) to the string the
 // null-terminated string pointed to by s.
 // Precondition: The pointer s must be non-null.
 // If the string has insufficient capacity to hold the new value
 // resulting from the append operation, the string is not modified
 // and an exception of type std::runtime_error is thrown.
 constexpr cexpr_basic_string& append(const value_type* s);

 // Appends (i.e., adds to the end) to the string another
 // cexpr_basic_string with the same character type (but
 // possibly a different maximum size).
 // If the string has insufficient capacity to hold the new value
 // resulting from the append operation, the string is not modified
 // and an exception of type std::runtime_error is thrown.
 template <size_type OtherM>
 constexpr cexpr_basic_string& append(
 const cexpr_basic_string<value_type, OtherM>& other);


 // Erases all of the characters in the string, yielding an empty
 // string.
 constexpr void clear(){
     for( T * it=&(_string[0]); *it!='\0'; ++it  ) {
             *it =0;
     }
     _string[0]='\0';
     static_assert(this->size() == 0);
 }

 };
}
```

```cpp
#define CATCH_CONFIG_MAIN  // This tells Catch to provide a main() - only do this in one cpp file
#include "../catch2/catch.hpp"
#include "../include/ra/cexpr_basic_string.hpp"
#include <stdio.h>
typedef ra::cexpr::cexpr_basic_string<char,5> charC;
using namespace ra::cexpr;



TEST_CASE( "testing copy string", "[cexpr_basic_string]" ) {

  charC C;
  // get string is a custom function delete later if necessary
  const char * c = C.get_string();

  SECTION("test no parameter constructor"){
    REQUIRE(*(c)=='\0');

  }

   char arr[6]={1,2,3,4,5,6};
  charC D(arr);
  // get string is a custom function delete later if necessary
  const char * myArr = D.get_string();

SECTION("test if one parameter array copied properly"){

  REQUIRE(*(myArr)==1);
  REQUIRE(*(myArr+1)==2);
  REQUIRE(*(myArr+3)==4);
  REQUIRE(*(myArr+3)!=5);

}
  constexpr const char carr[3] = {12,2,3};

  constexpr charC E(&carr[0],&carr[3]);
  const char * yArr = E.get_string();

  // get string is a custom function delete later if necessary
SECTION("test if array was copied properly from iterator"){
  REQUIRE(*(yArr)==12);
  REQUIRE(*(yArr+1)==2);
  REQUIRE(*(yArr+3)==0);


}
  static constexpr size_t siz = E.max_size();
   constexpr int len = E.size();
SECTION("test sizetype"){
  REQUIRE(siz==5);
  REQUIRE(siz!=4);
  REQUIRE(len==3);

}
const char * begin = E.data();
SECTION("test data func"){
  REQUIRE(*begin==12);


}


const char *first = E.begin();
 const char * const end = E.end();

constexpr std::size_t size = E.size();
SECTION("test iterators "){
  REQUIRE(size ==3);
  REQUIRE(*first==12);
  REQUIRE(*end ==’\0’);
}
const char ref = E[2];
const char ref1 = E[3];

SECTION("operators"){
  REQUIRE(ref ==3);
  REQUIRE(ref1 ==0);
}
charC G(&carr[0],&carr[3]);
G.pop_back();
G.push_back(110);
const char * g_arr = G.get_string();
const int g_size = G.size();
SECTION("pushpop"){
  REQUIRE(g_arr[g_size-1]==110);

}


}
```