ChunkyString-2

HW 7

Generated by Doxygen 1.8.7

Fri Nov 4 2016 14:31:38

Contents

Chapter 1

HW7: Implementing (most of) ChunkyString

1.1 Introduction

The C++ standard `string` is convenient for most applications, but vague about asymptotic complexity for its operations. While its often a waste of time to reimplement STL structures, in this case we want a string that can ensure efficiency in the following areas:

- Memory Usage
- Insertion of Characters
- Deletion of Characters

To satisfy this, we write a compromise between a linked-list of characters and an array of characters called ChunkyString.

1.2 Usage

The ChunkyString class can be used by writing `#include "chunkystring.hpp"` in any file. A test suite to assert its correctness is provided in stringtest.cpp.

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Class Documentation

4.1 TestingLogger::AssertInfo Struct Reference

Public Attributes

- int **asserts_** = 0
- int **failures_** = 0

The documentation for this struct was generated from the following file:

- testing-logger.hpp

4.2 ChunkyString::Chunk Struct Reference

Public Member Functions

- Chunk (size_t length)

  Convenience constructor of Chunk.
- **Chunk** (const Chunk &rhs)=default
- Chunk & **operator=** (const Chunk &rhs)=default

Public Attributes

- size_t **length_**
- char **chars_** [CHUNKSIZE]

Static Public Attributes

- static const size_t **CHUNKSIZE** = 12

4.2.1 Constructor & Destructor Documentation

4.2.1.1 ChunkyString::Chunk::Chunk (size_t *length*)

Convenience constructor of Chunk.

Parameters

length	Value we set length_ data member to.

The documentation for this struct was generated from the following files:

- chunkystring.hpp
- chunkystring.cpp

4.3 ChunkyString Class Reference

Efficiently represents strings where insert and erase are constant-time operations.

`#include <chunkystring.hpp>`

Classes

- struct Chunk
- class Iterator

  STL-style iterator for ChunkyString.

Public Types

- using **value_type** = char
- using **size_type** = size_t
- using **difference_type** = ptrdiff_t
- using **reference** = value_type &
- using **const_reference** = const value_type &
- using **iterator** = Iterator< value_type, chunklist_iter_t >
- using **const_iterator** = Iterator< const value_type, chunklist_const_iter_t >

Public Member Functions

- ChunkyString ()

  Default constructor.
- **ChunkyString** (const ChunkyString &)=default
- ChunkyString & **operator=** (const ChunkyString &)=default
- iterator begin ()

  Return an iterator to the first character in the ChunkyString.
- iterator end ()

  Return an iterator to "one past the end".
- const_iterator begin () const

  Return a const iterator to the first character in the ChunkyString.
- const_iterator end () const

  Return a const iterator to "one past the end".
- void push_back (char c)

  Inserts a character at the end of the ChunkyString.
- size_t size () const

  String size.
- ChunkyString & operator+= (const ChunkyString &rhs)

  String concatenation.
- bool operator== (const ChunkyString &rhs) const

Equality.

- bool operator!= (const ChunkyString &rhs) const

  Inequality.

- bool operator< (const ChunkyString &rhs) const

  Comparison.

- std::ostream & print (std::ostream &out) const

  Printing.

- iterator insert (iterator i, char c)

  Insert a character before the character at i.

- iterator erase (iterator i)

  Erase a character at i.

- double utilization () const

  Average capacity of each chunk, as a percentage.

Private Types

- using **chunklist_t** = std::list< Chunk >
- using **chunklist_iter_t** = chunklist_t::iterator
- using **chunklist_const_iter_t** = chunklist_t::const_iterator

Private Attributes

- size_t **size_**
- chunklist_t **chunks_**

Friends

- template<typename Element , typename ChunklistIterator >
  bool **operator==** (const ChunkyString::Iterator< Element, ChunklistIterator > &lhs, const ChunkyString::↩ Iterator< Element, ChunklistIterator > &rhs)
- template<typename Element , typename ChunklistIterator >
  bool **operator!=** (const ChunkyString::Iterator< Element, ChunklistIterator > &lhs, const ChunkyString::↩ Iterator< Element, ChunklistIterator > &rhs)

4.3.1 Detailed Description

Efficiently represents strings where insert and erase are constant-time operations.

This class is comparable to a linked-list of characters, but more space efficient.

Remarks

reverse_iterator and const_reverse_iterator aren't supported. Other than that, we use the STL container typedefs such that STL functions are compatible with ChunkyString.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 ChunkyString::ChunkyString ( )

Default constructor.

Note

constant time

The default constructor builds a ChunkyString object with a size_ of 0, and leaves the chars_ array null.

Note

constant time

4.3.3 Member Function Documentation

4.3.3.1 ChunkyString::iterator ChunkyString::begin ( )

Return an iterator to the first character in the ChunkyString.

Iterator begin() method.

Returns

an iterator that points to the first Chunk of the ChunkyString

4.3.3.2 ChunkyString::const_iterator ChunkyString::begin ( ) const

Return a const iterator to the first character in the ChunkyString.

Iterator constant begin() method.

Returns

a constant iterator that points to the first Chunk of the ChunkyString

4.3.3.3 ChunkyString::iterator ChunkyString::end ( )

Return an iterator to "one past the end".

Iterator end() method.

Returns

an iterator that points past the last Chunk of the ChunkyString

4.3.3.4 ChunkyString::const_iterator ChunkyString::end ( ) const

Return a const iterator to "one past the end".

Iterator constant end() method.

Returns

a constant iterator that points past the last Chunk of the ChunkyString

4.3.3.5 iterator ChunkyString::erase (iterator *i*)

Erase a character at i.

What makes ChunkyString special is its ability to insert and erase characters quickly while remaining space efficient.

Parameters

i	iterator pointing to the character to erase

Returns

an iterator pointing to the character after the one that was deleted.

Note

constant time

Warning

invalidates all iterators except the returned iterator

4.3.3.6 iterator ChunkyString::insert (iterator *i*, char *c*)

Insert a character before the character at i.

What makes ChunkyString special is its ability to insert and erase characters quickly while remaining space efficient.

Parameters

i	iterator to specify insertion point
c	character to insert

Returns

an iterator pointing to the newly inserted character.

Note

constant time

Warning

invalidates all iterators except the returned iterator

4.3.3.7 bool ChunkyString::operator!= (const ChunkyString & *rhs*) const

Inequality.

operator!=

leverages operator==

4.3.3.8 ChunkyString & ChunkyString::operator+= (const ChunkyString & *rhs*)

String concatenation.

operator+= method

concatenates two ChunkyStrings together

4.3.3.9 bool ChunkyString::operator< (const ChunkyString & *rhs*) const

Comparison.

operator<

lexicographical_compare method, leverage built-in library method

4.3.3.10 bool ChunkyString::operator== (const ChunkyString & *rhs*) const

Equality.

operator==

compares two ChunkyStrings for equality

Returns

true if the two ChunkyStrings are the same size and have the same contents, false otherwise

4.3.3.11 std::ostream & ChunkyString::print (std::ostream & *out*) const

Printing.

print method

prints contents of ChunkyString to standard out

4.3.3.12 void ChunkyString::push_back (char *c*)

Inserts a character at the end of the ChunkyString.

Parameters

c	Character to insert

Note

constant time, will invalidate the end() iterator

Parameters

c	Character to insert

Note

constant time

4.3.3.13 size_t ChunkyString::size ( ) const

String size.

size() method to return size_ of ChunkyString

Note

constant time

4.3.3.14 double ChunkyString::utilization ( ) const

Average capacity of each chunk, as a percentage.

This function computes the fraction of the ChunkyString's character cells that are in use. It is defined as

$$\frac{\text{number of characters in the string}}{\text{number of chunks} \times \text{CHUNKSIZE}}$$

For reasonably sized strings (i.e., those with more than one or two characters), utilization should never fall to near one character per chunk; otherwise the data structure would be wasting too much space.

The utilization for an empty string is undefined (i.e., any value is acceptable).

The documentation for this class was generated from the following files:

- chunkystring.hpp
- chunkystring.cpp

4.4 ChunkyString::Iterator< Element, ChunklistIterator > Class Template Reference

STL-style iterator for ChunkyString.

Public Types

- using **value_type** = char
- using **reference** = Element &
- using **pointer** = Element *
- using **difference_type** = ptrdiff_t
- using **iterator_category** = std::bidirectional_iterator_tag
- using **const_reference** = const value_type &

Public Member Functions

- Iterator ()=default

  < Default constructor
- Iterator (const Iterator< value_type, chunklist_iter_t > &i)

  Synthesized destructor.
- Iterator & operator++ ()

  Incrementer.
- Iterator & operator-- ()

  Decrementer.
- reference operator* () const

  Dereference.
- bool operator== (const Iterator &rhs) const

  Equality comparison.
- bool operator!= (const Iterator &rhs) const

  Equality comparison.

Private Member Functions

- **Iterator** (size_t index, ChunklistIterator chunksIter)

Private Attributes

- size_t **charsIndex_**
- ChunklistIterator chunksIterator_

  Templated Iterator.

Friends

- class ChunkyString

  Parameterized constructor.

4.4.1 Detailed Description

template<typename Element, typename ChunklistIterator>class ChunkyString::Iterator< Element, ChunklistIterator >

STL-style iterator for ChunkyString.

Synthesized copy constructor, destructor, and assignment operator are okay.

The five typedefs and the member functions are such that the iterator works properly with STL functions (e.g., copy).

Since this is a bidirectional_iterator, `operator--` is provided and meaningful for all iterators except ChunkyString::begin.

Remarks

The design of the templated iterator was inspired by these two sources: www.drdobbs.com/the-standard-librarian-defining-iterato/184401331 www.sj-vs.net/c-implementing-const_iterator-and-non-const-iterator-without-code-duplication

4.4.2 Constructor & Destructor Documentation

4.4.2.1 template<typename Element, typename ChunklistIterator> ChunkyString::Iterator< Element, ChunklistIterator >::Iterator ( ) [default]

< Default constructor

Convert a non-const iterator to a const-iterator, if necessary

4.4.3 Member Function Documentation

4.4.3.1 template<typename Element , typename ChunklistIterator > ChunkyString::Iterator< Element, ChunklistIterator >::reference ChunkyString::Iterator< Element, ChunklistIterator >::operator* ( ) const

Dereference.

Note

returns character

The documentation for this class was generated from the following files:

- chunkystring.hpp
- iterator-private.hpp

4.5 TestingLogger Class Reference

Classes

- struct AssertInfo

Public Member Functions

- **TestingLogger** (std::string name)
- bool **summarize** (bool verbose=false)
- void **clear** ()
- void **abortOnFail** ()

Static Public Member Functions

- static void **check** (bool assertion, std::string description)
- template<typename Function >
  static void **checkSafely** (Function assertionFn, std::string description)

Static Public Attributes

- static TestingLogger * **currentLogger** = nullptr

Private Types

- using **const_iter** = std::map< std::string, AssertInfo >::const_iterator
- using **iter** = std::map< std::string, AssertInfo >::iterator

Static Private Member Functions

- static void **exn_fail** (std::exception *exn, std::string what, std::string description)

Private Attributes

- std::map< std::string, AssertInfo > **assertions_**
- std::string **testName_**
- bool **failedSome_**
- bool **abortOnFail_**
- TestingLogger * **previousLogger_**

The documentation for this class was generated from the following files:

- testing-logger.hpp
- testing-logger.cpp

Chapter 5

File Documentation

5.1 chunkystring.cpp File Reference

Implements the ChunkyString class.

```
#include "chunkystring.hpp"
#include "testing-logger.hpp"
#include <cstddef>
#include <string>
#include <list>
#include <iterator>
#include <iostream>
#include <type_traits>
#include <cassert>
```

5.1.1 Detailed Description

Implements the ChunkyString class.

Authors

bobcat and heron

5.2 chunkystring.hpp File Reference

Declares the ChunkyString class.

```
#include <cstddef>
#include <string>
#include <list>
#include <iterator>
#include <iostream>
#include <type_traits>
#include "iterator-private.hpp"
```

Classes

- class ChunkyString

Efficiently represents strings where insert and erase are constant-time operations.

- class ChunkyString::Iterator< Element, ChunklistIterator >

  STL-style iterator for ChunkyString.
- struct ChunkyString::Chunk
- class ChunkyString::Iterator< Element, ChunklistIterator >

  STL-style iterator for ChunkyString.

Functions

- std::ostream & operator<< (std::ostream &out, const ChunkyString &text)

  Print operator.

5.2.1 Detailed Description

Declares the ChunkyString class.

Authors

CS 70 given code, with additions by bobcat and heron

5.2.2 Function Documentation

5.2.2.1 std::ostream& operator<< (std::ostream & *out,* const ChunkyString & *text*) [inline]

Print operator.

Remarks

Like the ones above, it's just a wrapper around a member function that does the actual work, and we don't mind if people know that.

5.3 stringtest.cpp File Reference

Tests a ChunkyString for correctness.

```
#include "testing-logger.hpp"
#include "chunkystring.hpp"
#include <string>
#include <sstream>
#include <stdexcept>
#include <cstddef>
#include <cstdlib>
#include <cassert>
#include "signal.h"
#include "unistd.h"
```

Macros

- #define **LOAD_GENERIC_STRING** 0

Typedefs

- using **TestingString** = ChunkyString

Enumerations

- enum **randomness_t** {
  MIN_VALUE, **MAX_VALUE**, **MID_VALUE**, **RANDOM_VALUE**,
  MIN_VALUE, **MAX_VALUE**, **MID_VALUE**, **RANDOM_VALUE** }
- enum **insertpoint_t** { **FRONT**, **BACK**, **FRONT**, **BACK** }
- enum **usepush_t** { **INSERT_ONLY**, **PUSH_AND_INSERT**, **INSERT_ONLY**, **PUSH_AND_INSERT** }

Functions

- void checkUtilization (const TestingString &test, size_t divisor)

  Assuming chunks are supposed to be at least 1/divisor full, checks for the lowest allowable utilization for the input string.
- template<typename T >
  std::string stringFrom (const T &thing)

  Converts any type that operator << can write out into a string.
- void checkDeepCopy (TestingString &test, const TestingString &copy)

  Make sure that equivalent TestingStrings are not actually using the same data (i.e. not a shallow copy)
- void checkWithControl (const TestingString &test, const string &control)

  Compare the TestingString to an expected value.
- void checkTwoWithControl (const TestingString &first, const TestingString &second, const string &fControl, const string &sControl)

  Compare two TestingStrings to an expected values and each other.
- void checkBothIdentical (const TestingString &first, const TestingString &second)
- void checkIterWithControl (const TestingString &test, const string &control, const TestingString::const_iterator &tIter, const string::iterator &cIter)

  Compare a TestingString and a TestingString iterator with expected values.
- bool **exampleTest** ()
- bool defaultConstructorTest ()

  Check all known conditions of a single default constructed object.
- bool copyConstructorTest ()

  Test the copy constructor.
- void assignTest (const TestingString &lhs, const TestingString &rhs)

  Assign one TestingString to another, then verify the assignment.
- void setUpTwoArguments (TestingString *helpers)

  Setup for functions requiring two arguments.
- bool assignmentTest ()

  Test Assignment in as many combinations as possible.
- bool appendTest ()

  Test += in as many combinations as possible.
- bool equalityTest ()

  Test equality and inequality.
- bool pushBackTest ()

  Test push_back on a variety of characters.
- bool iterateTest ()

  Basic iteration tests.
- void basicInsertTest (randomness_t action)

  Create a long string using insert.
- bool insertTest ()

  Basic insert tests (with iterator checks)
- void **setUpLongString** (TestingString &testString, string &controlString)
- void eraseAllTest (TestingString &test, string &control, randomness_t action)

Erase a long string using erase.

- bool eraseLongStringTest ()

  Basic erase tests (with iterator checks)
- bool iterateLongStringTest ()

  Basic iteration tests.
- bool appendUtilizationTest ()

  Create a low-utilization string by repeated appending.
- void myAdvance (TestingString::iterator &iter, const TestingString::iterator &end, size_t dist)

  Advances iter by dist or until it hits end, whichever comes first.
- bool utilizationOverflowLongStringTest ()
- void randomInsertTest (size_t n, insertpoint_t where, randomness_t randomize, usepush_t method)
- void randomEraseTest (size_t n, insertpoint_t where, randomness_t randomize)
- bool insertionUtilizationTest ()

  Try to break utilization by pure insertion.
- bool pushAndInsertUtilizationTest ()

  Try to break utilization by a combo of insertion and push_back.
- bool eraseUtilizationTest ()

  Try to break utilization by pure erasing.
- int main (int argc, char **argv)

  Run tests.

Variables

- const size_t NUM_HELPERS = 8

  Size of helpers array.
- const size_t **LONG_STRING_SIZE** = 500

5.3.1 Detailed Description

Tests a ChunkyString for correctness.

author: CS70 Sample Solution

5.3.2 Function Documentation

5.3.2.1 bool appendTest ( )

Test += in as many combinations as possible.

< Array of useful TestingStrings

5.3.2.2 bool assignmentTest ( )

Test Assignment in as many combinations as possible.

< Array of useful TestingStrings

5.3.2.3 void assignTest (const TestingString & *lhs*, const TestingString & *rhs*)

Assign one TestingString to another, then verify the assignment.

Remarks

This relies on a correct copy constructor, since the inputs are copied for repeated testing.

Parameters

lhs	The left side of assignment
rhs	The right side of assignment

5.3.2.4 void basicInsertTest (randomness_t *action*)

Create a long string using insert.

Parameters

action	Method for inserting. MIN_VALUE = begin(), MAX_VALUE = end(), RANDOM_VALUE = MID_VALUE = random spot in the string.

5.3.2.5 void checkBothIdentical (const TestingString & *first,* const TestingString & *second*)

Compares two TestingStrings by redirecting to checkWithControl(const TestingString&, const TestingString&, const string&, const string&, string).

5.3.2.6 void checkDeepCopy (TestingString & *test,* const TestingString & *copy*)

Make sure that equivalent TestingStrings are not actually using the same data (i.e. not a shallow copy)

Parameters

test	TestingString that will be modified in this test
copy	TestingString that will remain constant in this test

Remarks

The first input will be modified during the test, but is returned to its original value before the test returns.

5.3.2.7 void checkIterWithControl (const TestingString & *test,* const string & *control,* const TestingString::const_iterator & *tIter,* const string::iterator & *cIter*)

Compare a TestingString and a TestingString iterator with expected values.

Tests iterator correctness and all functions tested by checkWithControl(test, control).

Parameters

test	TestingString to check
control	Expected value of test
tIter	TestingString iterator to check
cIter	Expected value of tIter

5.3.2.8 void checkTwoWithControl (const TestingString & *first,* const TestingString & *second,* const string & *fControl,* const string & *sControl*)

Compare two TestingStrings to an expected values and each other.

Tests operator==, operator!=, operator$<$. and all functions tested by checkWithControl(test, control).

Parameters

first	TestingString to check
second	TestingString to check
fControl	Expected value of 'first'
sControl	Expected value of 'second'

5.3.2.9 void checkUtilization (const TestingString & *test*, size_t *divisor*)

Assuming chunks are supposed to be at least 1/divisor full, checks for the lowest allowable utilization for the input string.

Remarks

For insertion only, we assume chunks are at least 1/2 full. For erase, chunks can go down to 1/4 full. Since checkUtilization is not a test on its own, but rather a helper function to be used in other tests, it doesn't create its own TestingLogger object. Instead, its affirms will be associated with the TestingLogger of the calling function.

Parameters

test	TestingString to check
divisor	Fullness of chunk = 1/divisor

5.3.2.10 void checkWithControl (const TestingString & *test*, const string & *control*)

Compare the TestingString to an expected value.

Tests size, operator<<, begin, end, forward iteration, and edge case comparisons.

Parameters

test	TestingString to check
control	Expected value of the TestingString

5.3.2.11 bool equalityTest ( )

Test equality and inequality.

< Array of useful TestingStrings

5.3.2.12 void eraseAllTest (TestingString & *test*, string & *control*, randomness_t *action*)

Erase a long string using erase.

Parameters

test	String to erase
control	Expected value
action	Method for erasing. MIN_VALUE = begin(), MAX_VALUE = end(), RANDOM_VALUE = MI↩ D_VALUE = random spot in the string.

5.3.2.13 bool eraseLongStringTest ( )

Basic erase tests (with iterator checks)

< TestingString created

< Expected value of testString

5.3.2.14 bool iterateLongStringTest ( )

Basic iteration tests.

< TestingString created

< Expected value of testString

5.3.2.15 void randomEraseTest (size_t *n*, insertpoint_t *where*, randomness_t *randomize*)

Test erase. Erases characters into a string within a locus of 'n' from the front or back (depending on whether where is FRONT or BACK), exactly how the locus is defined depends on randomize and the number of characters currently in the string.

5.3.2.16 void randomInsertTest (size_t *n*, insertpoint_t *where*, randomness_t *randomize*, usepush_t *method*)

Test insert. Inserts characters into a string within a locus of 'n' from the front or back (depending on whether where is FRONT or BACK), exactly how the locus is defined depends on randomize and the number of characters currently in the string.

5.3.2.17 template<typename T > std::string stringFrom (const T & *thing*)

Converts any type that operator << can write out into a string.

Parameters

thing	thing to convert

Returns

string representation of thing

Exceptions

std::invalid_argument	if conversion fails

5.3.2.18 bool utilizationOverflowLongStringTest ( )

Try to break the utilization of the string using insert and erase. Strategy: force overflow of each chunk, then erase full chunks. < TestingString created

< Expected value of testString