Data Structures in C++

Chapter 7

Tim Budd

Oregon State University Corvallis, Oregon USA

Outline – Chapter 7

The `string` data type

- Primitive C++ strings
- Problems solved using Strings
  - Palendrome testing
  - Splitting line into words
- Operations of the standard data type
- Implementation of the string data type

Characters in C++

In C++ a character is simply a form of integer. Arithmetic and relational operations can be performed on characters, just as on integers.

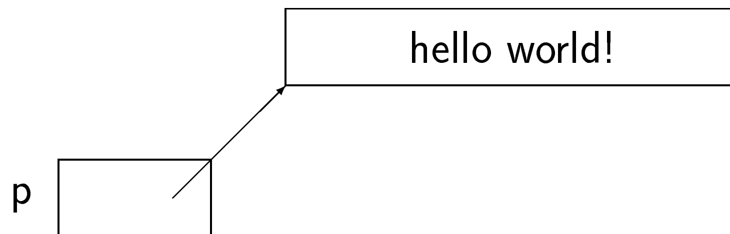
```
int isupper(char c)
{
    // return true if c is an upper case letter
    return (c >= 'A') && (c <= 'Z');
}
```

```
char tolower(char c)
{
    if (isupper(c))
        return (c - 'A') + 'a';
    return c;
}
```

Primitive C++ strings

In C++ a literal string is simply a pointer to a sequence of characters. No movement of characters is performed in the following, only an assignment of pointers.

```
char * p;
p = "hello world!";
```



Pointers can be subscripted

Like all pointers, pointers to characters can be subscripted, and thus literals can be modified.

```
p[0] = 'y';
p[5] = p[6];
p[6] = ' ';
p[8] = 'i';
p[10] = '\?';
```

Changes the text array into `yellow oil?!`.

Not checks are made on the validity of pointer subscripts. Can even be negative!

Null Character Termination

Strings are terminated with a null character, a character with value zero.

```
char * p;
p = "hello world!";
```

The literal contains 13 characters. 12 printing characters and one null character. The programmer must remember the null character, and make sure space is allocated to hold it.

h	e	l	l	o		w	o	r	l	d	!	*null*

Low level string routines

The standard C++ library `string.h` defines a number of simple routines to manipulate null-terminated character strings.

- `strlen(str)` – length (number of characters in string)
- `strcpy(to, from)` – duplicate string value
- `strcmp(str1, str2)` – compare strings, returning negative, 0 or positive

The `string` data abstraction

Here are some ways in which the `string` data type is an improvement over viewing strings just as an array of characters:

- Bounds checking on subscription, copys, and catenation.
- Assignments which result in copies.
- Comparisons performed using relational operators.
- High level operations such as substrings, pattern matching.

Example Problem – Palendrome testing

The following functions are intended to illustrate

- Use of the string data type
- String member functions and operations
- Generic algorithms that are useful with strings

Problem is to tell if a string represents a palendrome – reading the same forward and backward.

First type of palendrome is a simple word, such as “madam”.

Palindrome Type 1

```
bool palindrome_type1 (string & aString)
    // test aString is a type 1 palindrome
{
    string temp; // declare temporary
    temp = aString; // duplicate argument
    reverse (temp.begin(), temp.end()); // reverse
    return temp == aString; // test for equality
}
```

Three step process:

- Duplicate string (uses assignment)
- Reverse duplicate (uses generic algorithm)
- Tests for equality (uses relational operator)

A Palendrome that is not type 1

Won't work for

Rats Live on No Evil Star

ratS livE oN no eviL staR

Solution, first translate all characters to lower case, then test.

Palendrome Type 2

```
bool palindrome_type2 (string & aString)
   // test if aString is a type 2 palindrome
{
   string allLow (aString);
   transform (aString.begin(), aString.end(),
      allLow.begin(), tolower);
   return palindrome_type1 (allLow);
}
```

Again, a three step process:

1. Duplicate string (this time performed using a copy constructor)
2. Transform every character, using a generic algorithm that applies to each character copied the function `tolower`
3. Test the resulting value to see if it is a type 1 palendrome

The Transform Generic Algorithm

The transform generic algorithm looks something like this:

```
void transform
    (iterator start, iterator stop,
        iterator to, char fun(char))
{
    while (start != stop)
        *to++ = fun(*start++);
}
```

(This isn't exactly right, but we won't introduce the `template` mechanism, which is essential to the real form, until the next chapter.)

A Palendrome that is not type 2

Won't work on

A man, a Plan, a Canal, Panama!

Problem, need to remove spaces and punctuation.

Palendrome Type 3

```
bool palindrome_type3 (string & aString)
    // see if text is a type 3 palindrome
{
        // remove all punctuation and space
    string temp = remove_all (aString, " ,.!?");

        // then test resulting string
    return palindrome_type2 (temp);
}
```

- remove all punctuation and space characters
- then test the result to see if it is a palindrome of type 2

Removal Routine

```
string remove_all (string & text, string & spaces)
    // remove all instances of spaces
{
    string result;
    int textLen = text.length();
    int spacesLen = spaces.length();

    for (int i = 0; i < textLen; i++) {
        string aChar = text.substr(i, 1);
        if (spaces.find(aChar, 0) >= spacesLen)
            result += aChar;
            }
    return result;
}
```

Uses:

- substring – return a portion of a string
- find – see if one string is contained in another
- **+=** – append one string to another

- Returning a string as a result

Example 2 – split a line into words

```
void split (string & text,
   string & separators, list<string> & words)
{
   int n = text.length();

      // find first non-separator character
   unsigned int start =
      text.find_first_not_of(separators, 0);
      // loop as long as we have
      // a non-separator character
   while (start < n) {
         // find end of current word
      unsigned int stop =
         text.find_first_of(separators, start);
      if (stop > n) stop = n;
         // add word to list of words
      words.push_back
         (text.substr(start, stop - start));
         // find start of next word
      start =
         text.find_first_not_of (separators, stop+1);
      }
}
```

Using this function

```
void main() {
   string text =
"it was the best of times, it was the
worst of times.";
   string smallest = "middle";
   string largest = "middle";

   list<string> words;
   split(text, " .,!?:", words);

   list<string>::iterator start;
   list<string>::iterator stop = words.end();
   start = words.begin();
   for (; start != stop; start++) {
      if (*word < smallest)
         smallest = *word;
      if (*word > largest)
         largest = *word;
      }
   cout<< "small: "<< smallest<< "\n";
   cout<< "large: "<< largest<< "\n";
}
```

String Operations

Before you can use the string data type, you must include the header file:

```
# include <string>
```

String Operations

Constructors	
string s;	default constructor
string s ("text");	initialized with literal string
string s (aString);	copy constructor
Character Access	
s[i]	subscript access
s.substr(pos, len)	return substring starting at position of given length
Length	
s.length()	number of characters in string
s.resize(int, char)	change size of string, padding with char
s.empty()	true if string has no characters
Assignment	
s = s2;	assignment of string
s += s2;	append second string to end of first
s + s2	new string containing s followed by s2
Iterators	
string::iterator t	declaration of new iterator
s.begin()	starting iterator
s.end()	starting iterator
Insertion, Removal, Replacement	
s.insert(pos, str)	insert string after given position
s.remove(start, length)	remove length characters after start
s.replace(start, length, str)	insert string, replacing indicated characters
Comparisons	
s = s2 s != s2	comparisons for equality/inequality
s < s2 s <= s2	comparisons for relation
Searching Operations	
s.find(str)	find start of argument string in receiver string
s.find(str, pos)	find with explicit starting position
s.find_first_of(str, pos)	first position of first character from argument
s.find_first_not_of(str, pos)	first character not from argument
Input / Output Operations	
stream << str	output string on stream
string >> str	read word from stream
getline(stream, str, char)	read line of input from stream

Declaration

Declaration can either provide no value, or an initial value.

```
string s1;
string s2 ("a string");
string s3 = "initial value";
```

A *copy constructor* initializes a string as a copy of another string.

```
    // initialize s4 with value of s3
string s4 (s3);
```

Character Access

The subscript operator provides access to individual characters, can also be assigned to.

```
cout << s4[2] << endl;
s4[2] = 'x';
```

The substr operator provides access to portions of a string. Arguments are starting location and length.

```
cout << s4.substr(3, 2) << endl;
```

Extent of string

The `length` function tells how long a string is.

The `resize` operation makes an existing string longer or shorter, padding with characters if necessary.

```
    // add tab characters at end
s7.resize(15, '\t');

    // write new length
cout << s7.length() << endl;
```

The `empty` function tests if string is empty, and is more efficient than testing length against zero.

```
if (s7.empty())
    cout << "string is empty" << endl;
```

Assignment and Append

Strings can be assigned another string, a literal, or a character value:

```
s1 = s2;
s2 = "a new value";
s3 = 'x';
```

The += operator appends any of these forms to the end of a string.

```
s3 += "yz";    // s3 is now xyz
```

The + operator forms a new value, the catenation of the arguments

```
cout << s2 + s3 << endl;
```

Iterators

The member functions `begin()` and `end()` return beginning and past-the-end random access iterators. The type `string::iterator` can be used to declare an iterator value.

```
string::iterator itr = aString.begin();
for ( ; itr != aString.end() ; itr++)
    ...
```

Insertion, Removal and Replacement

```
    // insert after position 3
s3.insert (3, "abc");

    // remove positions 4 and 5
s3.remove (4, 2);

    // replace position 4 and 5 with "pqr"
s3.replace (4, 2, "pqr");
```

Searching Operations

The member function `find` searches for the argument in the receiver string.

```
s1 = "It was the best of times, it was the
worst of times.";

    // the following returns 19
cout << s1.find("times") << endl;

    // the following returns 46
cout << s1.find("times", 20) << endl;
```

The functions `find_first_of` and `find_first_not_of` treat argument as a set of characters.

```
        // find first vowel
    i = s2.find_first_of ("aeiou");
        // next non-vowel
    j = s2.find_first_not_of ("aeiou", i);
```

Useful Generic Algorithms

`reverse (iterator start, iterator stop)` reverse text in the given portion of string
`count (iterator start, iterator stop, target value, int & counter)` count elements that match target value, incrementing counter
`count_if (iterator start, iterator stop, unary fun, int & counter)` count elements that satisfy function, incrementing counter
`transform (iterator start, iterator top, iterator destination, unary)` transform text using unary function from source, placing into destination
`find (iterator start, iterator stop, value)` find value in string, returning iterator for location
`find_if (iterator start, iterator stop, unary function)` find value for which function is true, returning iterator for location
`replace (iterator start, iterator stop, target value, replacement value)` replace target character with replacement character
`replace_if (iterator start, iterator stop, unary fun, replacement value)` replace characters for which fun is true with replacement character
`sort(iterator start, iterator stop)` places characters into ascending order

Input / Output routines

```
string aString =
    "Find Average Word Length\n";
cout << aString;
string aWord;
int count = 0;
int size = 0;
while (cin >> aWord) {
    size += aWord.length();
    count++;
    }
cout << "Average word length:"
    << (size / count) << "\n";
```

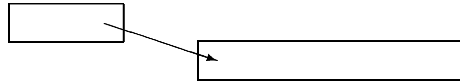
The `string` class description

```
class string {
public:
   typedef char * iterator; // define iterator type

   string    ();        // constructors
   string    (char *);
   string    (string &);
   ~string   ();    // destructor

      // member functions
   iterator  begin      ();
   bool      empty      ();
   iterator  end        ();
   int       find       (string &, int);
   int       find_first_of      (string &, unsigned int);
   int       find_first_not_of  (string &, unsigned int);
   void      insert     (unsigned int, string &);
   int       length     ();
   string    substr     (unsigned int, unsigned int);
   void      remove     (unsigned int, unsigned int);
   void      replace    (unsigned int, unsigned int, string &);
   void      resize     (unsigned int, char)

      // operators
   char &    operator [ ]    (unsigned int);
   void      operator =      (string &);
   void      operator +=     (string &);

      // friends
   friend bool operator == (string &, string &);
   friend bool operator != (string &, string &);
   friend bool operator <  (string &, string &);
   friend bool operator <= (string &, string &);
```

```
    friend bool operator >  (string &, string &);
    friend bool operator >= (string &, string &);

private:    // data areas
    char *              buffer;     // pointer to dynamic buffer
    unsigned short int  bufferLength; // length of dynamic buffer
};
```

Internal Buffer

The string data structure uses an internal buffer that grows and shrinks as the operations are performed.



Besause the size of the buffer cannot be predicted when the string is created, it must use dynamic memory allocation.

One of THE most important rules in developing software components

The following rule should become second nature:

Wherever possible, seek out repeated or common operations, and factor the code performing these operations into their own routines.

In the case of the string abstraction, the common operation will be allocating a buffer of a given size, as performed by the `resize()` member function.

Constructors

```
string::string ()
    // default constructor, length zero
{
    buffer = 0;
        // allocate buffer of length zero
    resize (0, ' ');
}

string::string (char * cp)
    // initialize string from literal string
{
    buffer = 0;
        // allocate buffer of correct size
    resize (strlen(cp), ' ');
        // then fill with values
    strcpy (buffer, cp);
}
```

Copy Constructor

A *copy constructor* is simply a constructor that duplicates a value of the same type, taking the original as argument.

```
string::string (string & str)
    // initialize string from argument string
{
    buffer = 0;

        // allocate buffer of correct size
    resize (str.length());

        // then fill with values
    strcpy (buffer, str.buffer);
}
```

It is good practice to always create copy constructors.

Assignment

Assignment is, not surprizingly, very similar to initialization.

```
void string::operator = (string & str)
    // reassign string to the argument value
{
    resize (str.length());
    strcpy (buffer, str.buffer);
}
```

Functions or Methods

When do you want to make a binary into a member function (such as assignment) and when do you want to make it into an ordinary function (such as <<) ?

- An ordinary function is normally not permitted access to the private portions of the class, whereas a member function is allowed such access. (The phrase "normally" is used, since we will later describe a mechanism to override this restriction).
- Implicit conversions, say from integer to float or integer to rational, will be performed for both right and left argument if the operator is defined in functional form, but only for the right argument if the operator is defined as a member function.

Destructor

A destructor is called implicitly when a value is about to be deleted. Needs to do whatever "housecleaning" is necessary before termination. For strings, it must simply return the memory associated with the buffer.

```
string::~string()
    // called implicitly when a string
    // is about to be deleted
    // free the memory associated
    // with the buffer
{
    delete [ ] buffer;
}
```

Resize the buffer

```
void string::resize (unsigned int newLength, char pd)
{   int i;

        // if no current buffer, length is zero
    if (buffer == 0)
        bufferLength = 0;
        // case 1, getting smaller
    if (newLength < bufferLength) {
            // just add new null character
        newbuffer[newLength] = '\0';
        }
    else {    // case 2, getting larger
            // allocate new buffer,
            // allow space for null character
        char * newbuffer = new char[newLength + 1];
        assert (newbuffer != 0);
            // first copy existing characters
        for (i = 0; i < bufferLength && buffer[i] != '\0';
i++)
            newbuffer[i] = buffer[i];
                // then add pad characters
        for ( ; i < newLength; i++)
            newbuffer[i] = pad;
                // add terminating null character
        newbuffer[i] = '\0';
            // free up old area, assign new
```

```
        if (buffer != 0)
            delete [ ] buffer;
        buffer = newbuffer;
        bufferLength = newLength;
    }
}
```

Computing Length

```
int string::length ()
    // return number of characters in string
{
    for (int i = 0; i < bufferLength; i++)
        if (buffer[i] == '\0')
            return i;
    return bufferLength;
}

bool string::empty ()
    // see if string is empty
{
    return buffer[0] == '\0';
}
```

Character Access

```
char & string::operator [ ] (unsigned int index)
    // return reference to character at location
{
    assert (index <= bufferLength); // not req by standard
    return buffer[index];
}
```

Note that this returns a *reference* to an existing character, can therefore be used as the target of an assignment.

Can only return references when the object being referenced will continue to exist even after the function returns.

Creating a substring

```
string string::substr
    (unsigned int start, unsigned int len)
{
    assert (start + len <= length());
    string sub;    // create new value
        // resize appropriately
    sub.resize (len, ' ');
    for (int i = 0; i < len; i++)
            // copy characters
        sub[i] = buffer[start + i];
    return sub;
}
```

Iterators

Can use use pointers for iterators (later we will see some iterators that are not simple pointers). A `typedef` in the class allows us to hide this fact from casual users.

```
class string {
public:
        // define iterator type
    typedef char * iterator;
```

Allows users to declare iterators without knowing their representation:

```
    string::iterator start = aString.begin();
    string::iterator stop = aString.end();

    for ( ; start != stop; start++)
        ...
```

Begin and End

Begin and end simply return pointers to the start and end of the internal buffer.

```
string::iterator string::begin ()
    // return starting iterator
    // just use pointer to buffer
{
    return buffer;
}

string::iterator string::end ()
    // return ending iterator
{
    return buffer + length();
}
```

Removal

Simply slide characters over.

```
void string::remove
    (unsigned int start, unsigned int len)
    // remove characters from given location
{
        // compute end of deleted run,
        // make sure it is in range
    int stop = start + len;
    assert (stop <= length());

        // move characters into place
    while (buffer[stop] != '\0')
        buffer[start++] = buffer[stop++];

        // make sure string is null terminated
    buffer[start] = '\0';
}
```

Insert

Insert is more complex, as we have to open up space before we copy values into position.

```
void string::insert
    (unsigned int position, string & newText)
    // insert text, starting at position
{
    int len = length();    // current length
    int ntLen = newText.length(); // additional
    int newLen = len + ntLen; // new length

        // if necessary, resize buffer
    resize(newLen, '\0');

        // move existing characters over
    for (int i = len; i > position; i--)
        buffer[i + ntLen] = buffer[i];

        // insert new characters
    for (int i = 0; i < ntLen; i++)
        buffer[position + i] = newText[i];
}
```

Replace and Append

Illustrate idea of reusing previously defined operations:

```
void string::replace
    (unsigned start, unsigned len,
        string & newText)
    // replace start to start + len
    // with new text
{
    remove (start, len);
    insert (start, newText);
}

void string::operator += (string & right)
    // append argument string to end
    // of current string
{
    insert (length(), right);
}
```

Catenation

Catenation is combination of duplication and append.

```
string operator + (string & left, string & right)
{
    string clone(left);    // copy left argument

        // append right argument
    clone += right;

    return clone;          // return result
}
```

Comparisons

Comparisons can all be related to a common routine, which is defined as follows:

```
int strcmp (char * p, char * q)
{
    while ((*p != '\0') && (*p == *q))
        { p++; q++; }
    return *p - *q;
}
```

Returns negative when first is less than second, 0 if equal, and positive when first is larger than second.

Defining the Relational Operators

All six relationals are easily defined using `strcmp`.

```
int operator < (string & left, string & right)
    // test if left string is lexicographically
    // less than right string
{
    return strcmp(left.buffer, right.buffer) < 0;
}
```

Problem

The problem is, the buffers are private. Solution, declare that these six functions are somehow special, namely “friends”.

Friends are allowed to look at the private parts of a class.

Notice that friendship is something that the class gives away, not something that can be taken.

```
class string {
public:
    ...
        // friends
    friend bool operator == (string &, string &);
    friend bool operator != (string &, string &);
    friend bool operator <  (string &, string &);
    friend bool operator <= (string &, string &);
    friend bool operator >  (string &, string &);
    friend bool operator >= (string &, string &);
```

Substring matching

```
int string::find (string & target, unsigned int start)
    // search for target string as a substring
{
    int targetLength = target.length();
        // stop is last possible starting position
    int stop = length() - targetLength;

    for (int i = start; i <= stop; i++)
        if (substr(i, targetLength) == target)
            return i;

        // no match found
        // return out of bound index
    return bufferLength;
}
```

Uses the equality testing operator, as well as substr operator.