Characters and Strings

Characters

Ex 6.1
In the old days, there was a very simple character set, ASCII, which represented the basic English language characters, and that is essentially what the standard char type represents.

- indicate with single quotes

```cpp
char my_char = 'a';
```
The world is not just English

A char is only 8 bits (1 byte) so it can only represent 256 characters. Not enough to deal with the world's character sets.

Unicode is a way to represent these character sets, but it is complicated.

UTF8

After a long and sorted kind of story, a committee created a Unicode standard called UTF8

- ASCII stuff unchanged
- Variable size byte values to store an essentially infinite number of characters.
new char types

c++ allows for new char types:
• wchar_t: older, implementation dependent
• char16_t and char32_t: c++11 for unicode

We'll worry about this later

This is just a complicated topic and we'll worry about it later
• plenty of other problems in C++
Character operations

Ex 6.2

these are all tests of various kinds you can place on a character. Most are booleans.
#include<cctype>

## Table 3.3: cctype Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isalnum(c)</td>
<td>true if c is a letter or a digit.</td>
</tr>
<tr>
<td>isalpha(c)</td>
<td>true if c is a letter.</td>
</tr>
<tr>
<td>iscntrl(c)</td>
<td>true if c is a control character.</td>
</tr>
<tr>
<td>isdigit(c)</td>
<td>true if c is a digit.</td>
</tr>
<tr>
<td>isgraph(c)</td>
<td>true if c is not a space but is printable.</td>
</tr>
<tr>
<td>islower(c)</td>
<td>true if c is a lowercase letter.</td>
</tr>
<tr>
<td>isprint(c)</td>
<td>true if c is a printable character (i.e., a space or a character that has a visible representation).</td>
</tr>
<tr>
<td>ispunct(c)</td>
<td>true if c is a punctuation character (i.e., a character that is not a control character, a digit, a letter, or a printable whitespace).</td>
</tr>
<tr>
<td>isspace(c)</td>
<td>true if c is whitespace (i.e., a space, tab, vertical tab, return, newline, or formfeed).</td>
</tr>
<tr>
<td>isupper(c)</td>
<td>true if c is an uppercase letter.</td>
</tr>
<tr>
<td>isxdigit(c)</td>
<td>true if c is a hexadecimal digit.</td>
</tr>
<tr>
<td>tolower(c)</td>
<td>If c is an uppercase letter, returns its lowercase equivalent; otherwise returns c unchanged.</td>
</tr>
<tr>
<td>toupper(c)</td>
<td>If c is a lowercase letter, returns its uppercase equivalent; otherwise returns c unchanged.</td>
</tr>
</tbody>
</table>

---

**strings**

our first STL container
Standard Template Library (STL)
3 parts

- containers
  - string
  - vector
  - deque
  - map

- generic algorithms
  - find
  - sort
  - count
  - remove

- iterators

Chars and strings

More STL

Containers:
- data structures to hold other data, with various capabilities/efficiencies
  - most are templated

Generic Algorithms
- algorithms for common tasks that work with container contents (mostly)

Iterators
- a kind of pointer, allowing access to containers independent of type
String Class Library

• A string is an STL class used to represent a sequence of characters.
  • an STL sequence, but not templated as it can only hold characters
  • templated containers can hold any type.
• As with other classes we have seen, there is a representation for the string objects and a set of operations.
• Use #include <string>

objects and methods

A string is a C++ object. The word object has special meaning in programming but there are two we care about for the moment:
• what data it stores
• what methods we can call
First Strings

Example 6.3

Declaring Strings

```c
string my_str;
Creates a string object and initializes it to the empty string "".

const string my_str = "tiger";
Creates a string object with 5 characters "tiger"
```
Internal structure

- Each element in a string is a single character
  - char my_char = 'a';
- In this case, a string is a sequence of char type elements.
- Thus a variable of type string can hold a large number of individual characters

Copy assignment

Declaration
string str1, str2 = "tiger";

Assignment
str1 = str2;

makes a copy of str2 so

str1 t i g e r  str2 t i g e r
Other ways to initialize a string

{} contains universal initializer, a list of elements to go in the string

Since strings hold characters, we list individual characters

```cpp
string first{'H', 'o', 'm', 'e', 'r'};
cout << first << endl;
// prints Homer
```

more initializers

Can create copies of an individual character in a string
- first arg is the count
- second arg is the character

```cpp
string a_5(5, 'a');
cout << a_5 << endl;
```

prints aaaaa
more initializers

copy construction is technically different from assignment, but it does the same kind of thing

```cpp
string first = "Homer";
string new_first = first;
cout << new_first << endl;
```

prints Homer

It's a copy of the original.

we worry about copying

If we copy a long string (say, a copy of Shakespeare as a string) we do a lot of work:

- we have to make memory (which the string class does) to hold it
- we have to use the CPU to move all that data around

We will discuss this more.
methods like functions

A method is a function that is:
• called in the context of a particular instance of an object
• uses the dot notation for the call

example methods size() and length()

```cpp
string my_str = "tiger";
size() method returns the number of characters in the string.
cout << my_str.size();
Will output the integer 5

.length() is the same as .size()
```
Data member: Subscript

• To access individual characters in a string, use the `.at` member function. Index starts at 0.
• `string my_str = "tiger";`
  `string my_str = "tiger";`
  `my_str [0 1 2 3 4]`
  `my_str [t i g e r]`
• `cout << my_str.at(2);`
• outputs 'g' (the character g)

[ ] instead of .at

You can also use the subscript operator [ ].
`string my_string;`
`my_string="hello";`
`cout << my_string[4]`
  `// output is 'o'`
There is one important difference:

If you access an non-existant index, `.at` will throw an error, `[ ]` will not (it will do something weird, but not throw an error)

One of the most important things to remember about strings (any sequence of things in C++) is that they start at 0.

You will save yourself grievous headaches if you remember this!!!
can assign values

You can assign using the .at or [ ] operator

```cpp
string my_str;
my_str = "hello";
my_str[0] = 'j'
// string is now jello
my_str.at(0) = 'h';
// back to hello
```

---

Subscript Assignment

```cpp
string my_str = "tiger";
my_str.at(2)= 'm';
cout << my_str;
• Outputs "timer"
```
assign method

You can also use the `assign` operator and get `substring` assignment

```cpp
string a_str;
a_str = "myTry";
string next_str;
next_str.assign(a_str, 2,
               string::npos);
// next_str becomes "Try"
```

string::npos

The `::` is the scope resolution operator. It gives you access to functions and variables that are defined as part of a class. So `string::npos` is the name of a variable within the string class.

It stands for "no position", a position not found in the string.
**Character Processing**

```cpp
string my_str = "tiger";
for(int i=0; i<my_str.size(); i++){
    cout << i << " : " << my_str[i] << endl;
}
```

**Output:**

```
0: t
1: i
2: g
3: e
4: r
```

---

**not int, string::size_type**

```cpp
string my_str = "tiger";
for(int i=0; i<my_str.size(); i++){
    cout << i << " : " << my_str[i] << endl;
}
```

Every STL container has a `size_type`. For strings it is `string::size_type`. Though you can get away with `int`, you should not. Instead:

```cpp
string my_str = "tiger";
for(decltype(my_str.size()) i=0; i<my_str.size(); i++){
    cout << i << " : " << my_str[i] << endl;
}
```

whatever size returns a `size_type`
size_types are unsigned

As with all unsigned types, you can get some strange behavior if you go below 0.

Watch for that (try it, see what it prints).

---

string input

Example 6.4
Some regular functions: I/O

- Input operator >> is overloaded:
  ```cpp
  string my_str;
  cin >> my_str;
  ```
- Reads first word in `istream` up to whitespace.
- If input is "fred", `my_str` is "fred".
- If input is "mary jones", `my_str` is only "mary"

More I/O, full line input

- To read a whole line of text (up to a newline character, '\n') use
  ```cpp
  getline( cin, my_str );
  ```
- If input is "Mary Jones likes cats", then `my_str` is "Mary Jones likes cats"
  ```cpp
  my_str = "Mary Jones likes cats"
  ```
- '\n' not included, is discarded
  ```cpp
  Mary Jones like ...
  ```
range based for loop

Example 6.5

Much better, range based for loop
- this is the for loop in Python!
- it's a C++11 thing

```cpp
string my_str = "tiger"
for (auto chr : my_str)
    cout << chr <<", ";
```

C++ can determine the type of each element, so we just auto the type
String Comparison

- Beginning at character 0 (leftmost) compare each character until a difference is found. The ASCII values of those different characters determines the comparison value.
- E.g. "aardvark" < "ant" since the second characters 'a'<'n' because 97<110

String ops

Ex 6.6
Concatenation

Concatenation appends one string to another.

```cpp
string result;
string tig = "tiger"
string ant = "ant";
result = tig + ant;
cout << result;

• Output is "tigerant"
```

Ex 6.6, substrings

The method is `substr`

```cpp
string my_str = "abc123";
my_str.substr(0, 4) // start at 0, len 4
    → "abcl"

if length is past end or no length argument, assume to the end
my_str.substr(1, 100)
my_str.substr(1);
my_str.substr(1, string::npos)
    → "bc123"
```
another initializer

You can do this at the initializer stage

```c
string last = "Simpson";
string sub_last(last, 3, 2);
copy from last, start at index 3, length of 2. prints "ps"
```

Constructors

Methods/functions called in the context of initializing a newly declared variable are called constructors.

Can have multiple based on arguments

All the initializers we've seen are constructors. We will write our own for our new classes later.
some general seq ops

```cpp
string my_str = "abc";
// push_back, append 1 element to end
my_str.push_back('d');  //"abcd"
// append string at end
my_str.insert(my_str.size(), "efgh");
```

### Section 9.5 Additional string Operations

<table>
<thead>
<tr>
<th>Table 9.13: Operations to Modify Strings</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>arg</strong></td>
</tr>
<tr>
<td><strong>arg</strong></td>
</tr>
<tr>
<td><strong>arg</strong></td>
</tr>
<tr>
<td><strong>arg</strong></td>
</tr>
<tr>
<td><strong>arg</strong></td>
</tr>
</tbody>
</table>

- args must be of the following: `append` and `assign` can use all forms
- `erase` can use one of the following: `append` and `assign` can use all forms

### Args for replace and insert depend on how range or pos is specified.

<table>
<thead>
<tr>
<th><strong>arg</strong></th>
<th><strong>replace</strong></th>
<th><strong>insert</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>replace</code></td>
<td><code>replace</code></td>
<td><code>insert</code></td>
</tr>
<tr>
<td><code>pos, len</code></td>
<td><code>pos, len</code></td>
<td><code>iter</code></td>
</tr>
<tr>
<td><code>args</code></td>
<td><code>pos, args</code></td>
<td><code>iter, args</code></td>
</tr>
</tbody>
</table>

- `replace` and `insert` can be used with `pos, len` or `iter`.
**String find function**

Example 6.7

String find function finds the first occurrence of char in a string, starting at the start position.

```cpp
string my_str = "hello world"
string::size_type pos = 0;
pos = my_str.find('e', pos); // pos gets set to 1
// doesn't exist? return string::npos
```
lots of find functions

Look at table 9.14 (pg 365). Works for characters and strings

- `s.rfind(arg)`: find last of arg in s
- `s.find_first_of(arg)`: first of any of the args in s
- `s.find_last_of(arg)`: find last of any of the args in s
- `s.find_first_not_of(args)`: find first of any char in s that is not in arg
- `s.find_last_not_of(args)`: find last of any char in s that is not in arg

lychrel number

Example 6.8