
CS11 *Advanced* C++

Spring 2007-2008

Lecture 6

Today's Topics

- C++ strings
- C++ streams



C++ Strings

- C++ inherits C notion of `char*` as a “string”
 - Zero-terminated array of `char` values
 - Useful C functions for string manipulation in `<cstring>` header (C++ name for `string.h`)
 - C++ introduces the `string` class
 - Dynamically allocated, resizable string
 - Provides many features and benefits over `char*` strings
 - Generally painless to use in very complex ways
 - Prefer `string` to `char*`, wherever possible!
 - `#include <string>`
-

What Is A String?

- `string` is an instantiation of `basic_string` template for `char` sequences

```
typedef basic_string<char> string;
```

- Can support other kinds of strings!

- `wchar_t` is a Unicode character type

```
typedef basic_string<wchar_t> wstring;
```

- Highly customizable data type

- Different in-memory representations of characters
 - Locale-specific comparisons
 - etc.
-

C++ String Initialization

- C++ `string` objects can be initialized from other `strings`, or from `char*` values

```
string s1 = "green";           // Same as s1("green");  
string s2 = s1;               // Same as s2(s1);
```

- `s2` is an independent copy of `s1`

- Can also initialize to be a repeated character

```
string reps(5, 'a');          // reps == "aaaaa"
```

- Can also initialize to a substring

```
string s3(s1, 2, 2);          // s3 == "ee"
```

- First number is the position (zero-based)
- Second number is count

- Other constructor options too...

C++ String Assignment

- **string** supports assignment operator

```
string s1 = "orange";  
string s2 = "yellow";  
s2 = s1;  
s1 = "gray";
```

- Can also use **assign()** member-function

```
s2.assign(s1);  
s1.assign("gray");
```

- Strings do not share underlying storage
 - Assignment makes a *copy* of what is assigned.
 - Technically, implementations might optimize this using a “copy on write” technique

Comparison, Concatenation

- **string** supports comparison operators
 - == != < > etc.
 - Case sensitive by default
 - Depends on locale!
 - Use + or += for concatenation

```
string title = "purple";  
title = title + " people";  
title += " eater";
```

 - Can also append individual characters

```
title += 's';
```
 - Can also use `append()` member-function
-

String Lengths and Indexes

- `length()` member-function reports number of characters in string

```
string color = "chartreuse";  
cout << color << " has " << color.length()  
    << " characters." << endl;
```

- Characters have indexes 0 to `length() - 1`
 - `string::npos` indicates “invalid index”
 - All strings have `length() < string::npos`
-

Individual Characters

- Individual character access with []

```
string word = "far";
```

```
word[1] = 'o';    // now word == "for"
```

- Index values are *unchecked*. Fast, but risky.

- Can also use `word.at(1) = 'o';`

- Index values are checked; out of range values cause `out_of_range` exception to be thrown

- Both of these can be used on LHS or RHS of assignment
-

Classifying Characters

- Useful helper functions in `<cctype>` header

- (from the C standard header `cctype.h`)

<code>int isalpha(int)</code>	Any letter: a..z or A..Z in C locale
<code>int isupper(int)</code>	Uppercase letter: A..Z in C locale
<code>int islower(int)</code>	Lowercase letter: a..z in C locale
<code>int isdigit(int)</code>	Decimal digit: 0..9
<code>int isxdigit(int)</code>	Hexadecimal digit: 0..9, a..f or A..F
<code>int isspace(int)</code>	Any whitespace character
<i>etc.</i>	
<code>int toupper(int)</code>	Convert letter to uppercase
<code>int tolower(int)</code>	Convert letter to lowercase

- Equivalent functions in `<cwctype>` header

String Traversal

- `string` objects are *collections* of characters
- They also provide iterators over their characters
 - `begin()` is iterator at start of `string` value
 - `end()` is iterator at end of `string` value

```
string col = "purple";
string::iterator si;
// Send the contents of col to cout
for (si = col.begin(); si != col.end(); si++)
    cout << *si;
```

- You *can* use `strings` with STL algorithms
 - ...but it's not very efficient!
 - Best to use the provided `string` member-functions

Finding Substrings

- Four versions of `find()` member function

```
size_type find(const string &, size_type start = 0)
```

```
size_type find(const char *, size_type start,  
              size_type length)
```

```
size_type find(const char *, size_type start = 0)
```

```
size_type find(char, size_type start = 0)
```

- Returns index of match, or `string::npos` for no match

- The `rfind()` member function searches backwards through a `string`

- Provides same four versions, with appropriate default values for arguments

Find Variants

- **find_first_of()**, **find_last_of()**
 - Unlike **find()**, matches if *any* character in argument appears in string
 - **find_first_of()** starts at beginning and goes forward
 - **find_last_of()** starts at end and goes backwards
 - **find_first_not_of()**, **find_last_not_of()**
 - Finds first character not in argument
 - Again, can search from beginning, or from end
-

String Manipulation

- **substr()** extracts a substring

```
substr(size_type start = 0,  
       size_type length = npos)
```

 - Returns a new **string** containing the substring
 - Note that default arguments just copy entire string
- **replace()** modifies a substring
 - Again, many versions of **replace()**
 - Some take iterators; some take start, length arguments
- **erase()** removes a substring
 - “Replace with nothing”
 - Can call with start, length; or with iterators

More String Manipulation!

- **append()** member functions allow for appending characters or strings
 - Can append a C++ **string**, a C **char*** string, or an individual **char** value
 - **insert()** member functions for inserting characters into a string
 - Can insert at a specific index
 - Can use an iterator to indicate location
-

Converting Strings to `char*`

- Can convert `strings` to `char*` values
 - `c_str()` returns a zero-terminated `char*`
 - `data()` returns a *non-terminated* `char*`
 - `copy(...)` member function copies a string into a `char*` buffer
- Examples:

```
string value = "orange";  
printf("%s\n", value.data());    // WRONG!  
printf("%s\n", value.c_str());  // Correct
```

 - Don't use `data()` when you need the trailing 0!

c_str() Gotchas!

- Don't cache pointers returned by `c_str()` or `data()` member functions
 - May not have valid data after a non-const call to `string` member function
- Don't return `c_str()` or `data()` value from a `string` local variable!
 - Memory is managed by `string` instance
 - It goes away when the string variable goes out of scope!

```
char * getUsername() {  
    string name;  
    cout << "Enter username:  ";  
    cin >> name;  
    return name.c_str(); // BAD!  
}
```

String IO

- **string** provides >> and << implementation

```
string name;  
cout << "Enter your name: ";  
cin >> name;  
cout << "Hello, " << name << "!" << endl;
```

- Can also read a whole line of input from a stream

```
string inputLine;  
getline(cin, inputLine); // Read a whole line
```

- Default end-of-line (eol) character is “\n”
- Can specify different eol character if needed
- eol character is consumed from stream, but does not appear in **string** contents

C++ Streams

- C++ provides general-purpose, stream-based facility for program input and output
 - Output: converting variables/objects into char sequences
 - Input: converting char sequences into variables/objects
 - Locale affects formatting in the conversion process
- C++ Stream IO is extensible to user-types
 - Primitive types and standard classes are supported
 - Easy to incorporate user-defined types into C++ Stream IO
- Supports console IO, file IO, etc.
 - Can also treat strings as streams
 - Can write stream wrappers for networking sockets, etc.

Console-IO Streams

- Standard program input/output uses:
 - `cin` Standard input stream
 - `cout` Standard output stream
 - `cerr` Unbuffered output for error messages
 - `clog` Buffered output for error messages
 - Also `wchar_t` versions!
 - `wcin, wcout, wcerr, wclog`
 - Defined in `<iostream>` header
-

Stream State

- All streams have state associated with them
- Streams provide flags indicating “what happened” or “what might happen”

```
bool good()      // Next operation might succeed
bool eof()      // End of input seen
bool fail()     // Next operation will fail
                // (i.e. a previous operation failed)
bool bad()      // Stream is corrupted

iostate rdstate()           // Get IO-state flags
void clear(iostate f = goodbit) // Set IO-state flags
void setstate(iostate f)    // Add f to flags
```

Looping on Streams

- Streams also provide test operations

```
operator void*();    // Nonzero return value if !fail()
bool operator!() const { return fail(); }
```

- Can use streams in loop-conditions

```
string word;
while (cin >> word) {
    ... // Do stuff with each word.
}
```

- Remember, >> returns `istream&`
- Then `istream&` is cast to `void*` (standard C++ behavior)
- Return-value of cast depends on stream's status
- Loop terminates when there are no more words to read

Stream-State Flags

- State flags are defined in `ios_base` class

```
ios_base::badbit  
ios_base::eofbit  
ios_base::failbit  
ios_base::goodbit
```

- Can use `rdstate()` and flags to do stuff:

```
ios_base::iostate s = cin.rdstate();  
if (s & ios_base::badbit) {  
    ... // Handle input errors.  
}
```

- Or just use `fail()`, `bad()`, etc.

- Setting flags is a little simpler:

```
cin.setstate(ios_base::failbit); // state += failbit  
cin.clear(ios_base::goodbit);   // state = goodbit
```

Stream-State and Exceptions

- Testing stream-state can be annoying
 - Can configure streams to throw exceptions when state changes
 - `void exceptions(iostate except);`
 - Specify the states that should cause exceptions
 - Example: `ios_base::badbit | ios_base::failbit`
 - When stream goes into those states, `ios_base::failure` exception is thrown
 - To find out what states will throw an exception:
`iostate exceptions();`
 - Throwing exceptions on IO errors is *off* by default
-

Reading Unformatted Characters

- The `>>` operator is for formatted input
 - Whitespace is automatically skipped
- `istream`s provide several `get()` functions for reading unformatted input

```
int get(); // Reads one character
istream & get(char &ch); // Reads a character into ch
istream & get(char *p, int max)
istream & get(char *p, int max, char term)
istream & getline(char *p, int max)
istream & getline(char *p, int max, char term)
```

- `get()`, `getline()` read characters, up to a terminator
 - Default terminator is newline
 - `get()` does not remove terminator from stream!
 - `getline()` is preferred to `get()` because of this
-

Stream IO for User-Defined Types

- Implement << operator for user-type output
 - Signature:
`ostream & operator<<(ostream &os, const UserType &u);`
 - Should not be a member function of anything
 - If anything, it should be an `ostream` member, but we can't change `ostream`!
 - Implement >> operator for user-type input
 - Signature:
`istream & operator>>(istream &is, UserType &u);`
 - Remember the *non-const* user-type reference!
-

Stream-Input Example: Complex

- Stream-input operator for reading **complex** values
- Read formats: f or (f) or (f , f)
 - f is a decimal number
 - Whitespace padding can be included or excluded
- Implementation:

```
// Handles f or (f) or (f,f) formats
istream & operator>>(istream &s, complex &a) {
    double re = 0, im = 0; // Components of complex number
    char ch = 0;           // Chars read from stream

    if (!s)
        return s; // Stream already in fail state!
    ...
}
```

Stream-Input Example: Complex (2)

```
...
s >> ch;           // Get first non-whitespace char
if (ch == '(') {   // Value(s) surrounded in parens
    s >> re >> ch;

    if (ch == ',') s >> im >> ch; // Found comma

    if (ch != ')') s.clear(ios_base::failbit);
}
else {             // Value not surrounded in parens
    s.putback(ch); // "Unread" the char we just read
    s >> re;       // Try reading a number instead
}

if (s) a = complex(re, im); // Stream state still good

return s;
}
```

C++ File IO

- File IO is almost as easy as console IO
 - `#include <fstream>`
 - `fstream` for reading and writing to a file
 - `ifstream` for reading from a file
 - `ofstream` for writing to a file
 - Filename, mode can be passed to constructor
 - Example 1: open a word list for reading

```
ifstream wordList("words.txt");
```
 - Example 2: open a result-file for appending

```
ofstream resultData("result.dat", ios_base::append);
```
 - Also has member-functions for opening/closing files

```
void open(const char *p, openmode m = out);  
void close();  
bool is_open();
```
-

File IO Modes

- **ios_base** class defines file IO modes
 - **app** Open for appending
 - **ate** Open, seek to end of file (“at end”)
 - **binary** Binary-mode IO (instead of text-mode)
 - **in** Open for reading
 - **out** Open for writing
 - **trunc** Truncate file to zero length
- Can bitwise-OR these values together
- Example:

```
fstream dictionary("dict.txt",  
                  ios_base::in | ios_base::out);
```

Strings as Streams

- `<sstream>` header declares string-streams
 - `string` objects are read from or written to, like a stream
 - Three types (like file IO)
 - `stringstream` for read/write
 - `istringstream` for read-only
 - Useful for easily parsing data from a string
 - `ostringstream` for write-only
 - Useful for formatting output messages
 - Won't overflow; grows as needed
 - Can access/modify string-stream's underlying data
 - Pass `string` value to `stringstream` constructor
 - `string str()` returns a copy of stream's internal data
 - `void str(const string &)` sets stream's contents
-

This Week's Assignment

- Extend your ray tracer to use stream IO for scene descriptions
 - Implement stream-input operator for vectors and colors
 - Make sure to flag stream input errors properly!
 - Then, build functions to read scene objects
 - Probably best to not use stream-input operator for this task
 - Write functions that use stream-input for vectors, colors, etc. to construct scene objects
-

Reading Scene Objects

- Simple scene description format:
 - A red sphere at location (0, 0.5, 0) with radius 0.5:
`sphere (0, 0.5, 0) 0.5 [1, 0, 0]`
 - A purple plane through the origin:
`plane (0, 0, 0) 0 [1, 0, 1]`
- Can't really implement this with a stream-input operator for **SceneObject**
`istream & operator>>(istream &is, SceneObject &so)`
 - We would need to know what kind of scene-object we're reading, before we actually read it!

Reading Scene Objects (2)

- A simple approach:
 - First value in the scene format is an “object type”
`sphere (0, 0.5, 0) 0.5 [1, 0, 0]`
`plane (0, 0, 0) 0 [1, 0, 1]`
- Two steps:
 - Read the “object type” from the stream
 - e.g. read in “sphere”
 - Dispatch to a function for reading in that kind of object
 - e.g. `SPSceneObject ReadSphere(istream &is)`
 - Function can use `Vector3f/Color` stream-input impl.

Extensible Scene Description Language

- Can make this process data-driven:

```
// Define a type for scene-object reader functions.
```

```
typedef SPSceneObject (*SceneObjectReader)(istream &is);
```

```
// Create specific reader functions:
```

```
SPSceneObject ReadSphere(istream &is);
```

```
SPSceneObject ReadPlane(istream &is);
```

```
...
```

- Finally, create a map to drive the input:

```
map<string, SceneObjectReader> readFuncs;
```

```
readFuncs["sphere"] = ReadSphere;
```

```
readFuncs["plane"] = ReadPlane;
```
