Computer Science 3307a/b

Strings and Streams

Jeff Shantz
Department of Computer Science

Some examples adapted from http://cs.stmarys.ca/˜porter/csc/ref/c_cpp_strings.html and from our very own PhD student, Beth Locke

“Strings” in C

- C has no notion of *strings*
- Instead, strings created using character arrays
- `string.h` library provides helper functions to make working with character arrays easier

“Strings” in C

```c
#include <stdio.h>
#include <string.h>

main()
{
  char first_name[80];
  char last_name[80];
  char full_name[159];

  printf("First Name : ");
  fgets(first_name, 80, stdin);

  printf("Last Name : ");
  fgets(last_name, 80, stdin);

  strncpy(full_name, first_name, strlen(first_name) - 1);
  strcat(full_name, " ");
  strncat(full_name, last_name, strlen(last_name) - 1);

  printf("Your full name is %s\n", full_name);
}
```

Strings in C++

- The Standard C++ library introduces the `string` class, defined in the `string` library
- Makes working with string types considerably easier than in C

```
#include <iostream>
#include <string>

main()
{
  std::string s("Hello");

  s += " World!";

  std::cout << s << std::endl;
}
```

Initializing Strings

C

```
char str1[6] = "Hello";
char str2[] = "Hi!";
char str3[] = {’H’, ’i’, ’\0’};
```

Initializing Strings

C

```
char str1[6] = "Hello";
char str2[] = "Hi!";
char str3[] = {’H’, ’i’, ’\0’};
```

C++

```
string str1("Hello World");
string str2 = "Hi there!";
string str3(5, ’*’);
```

Assigning/Copying Strings

C

```c
// not directly possible
char str[6];
str = "Hello";      // won't work

strcpy(str, "Hello");
```

Assigning/Copying Strings

C

```
// not directly possible
char str[6];
str = "Hello";        // won't work

strcpy(str, "Hello");
```

C++

```
string str;
string str2("'Bye");

str.assign("Hi"); // str now stores "Hi"
str.assign(str2); // str now stores "'Bye"
```

Concatenating Strings

C

```
strcat(str1, str2);
```

Concatenating Strings

C

```
strcat(str1, str2);
```

C++

```
str1.append(str2);
str1.append(80, '*');
```

Accessing Individual Characters

C

```
str[index]
```

Accessing Individual Characters

C

```
str[index]
```

C++

```
str.at(index)
```

Computing String Length

C

```
strlen(str)
```

Computing String Length

C

```
strlen(str)
```

C++

```
str.length()
```

Introduction to Operator Overloading

- C++ allows operators to be *overloaded*
- This allows us to define the functionality of the various operators:

Assignment	`=`					
Arithmetic	`+`	`-`	`/`	`*`	`++`	
Comparison	`==`	`!=`	`>`	`<`	`>=`	`<=`
Logical	`&&`	`||`	`!`			

- Potential for great abuse
  - Please don't define a division operator for a Person class
- `string` makes heavy use of operator overloading for ease of use

`string` Operator Overloads

Method	Operator Overload
`oneString.assign(another);`	`oneString = another;`
`oneString.append(another);`	`oneString += another;`
`oneString.at(n);`	`oneString[n];`

Comparing `string`s (using overloads)

C

```c
if (strcmp(str1, str2) < 0)
    printf("str1 comes 1st.");
if (strcmp(str1, str2) == 0)
    printf("Equal strings.");
if (strcmp(str1, str2) > 0)
    printf("str2 comes 1st.");
```

Comparing `string`s (using overloads)

C

```c
if (strcmp(str1, str2) < 0)
    printf("str1 comes 1st.");
if (strcmp(str1, str2) == 0)
    printf("Equal strings.");
if (strcmp(str1, str2) > 0)
    printf("str2 comes 1st.");
```

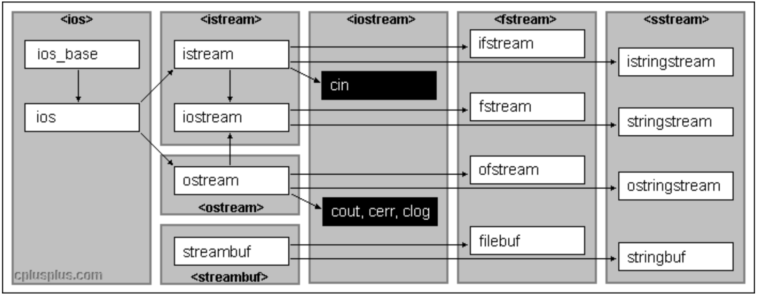
C++

```cpp
if (str1 < str2)
    cout << "str1 comes 1st.";
if (str1 == str2)
    cout << "Equal strings.";
if (str1 > str2)
    cout << "str2 comes 1st.";
```

Streams

- I/O operations in C++ occur on *streams*
- **Stream**: A sequence of bytes of indefinite length
  - Input operations: Bytes flow from a device (e.g. keyboard, disk, network connection) into main memory
  - Output operations: Bytes flow from main memory to a device (e.g. monitor, printer, disk, network connection)
- Allow us to work with various types of I/O devices using a consistent interface
- Streams are serial – we cannot directly read/write a random byte in a stream as we can in an array
  - However, we can *seek* to a specific location just as we might fast-forward or rewind a show on a PVR

Streams


<ios>
ios_base
ios
cplusplus.com
<istream>
istream
iostream
ostream
<ostream>
streambuf
<streambuf>
<iostream>
cin
cout, cerr, clog
<fstream>
ifstream
fstream
ofstream
filebuf
<sstream>
istringstream
stringstream
ostringstream
stringbuf

Stream Operators

<<	Stream insertion operator	Write data to a steam
>>	Stream extraction operator	Read data from a stream

- Both operators defined for all standard C++ data types
- Can *overload* these operators to allow them to work with our own classes

Pre-defined Stream Objects

`cin`	Standard input stream
`cout`	Standard output stream
`cerr`	Standard error stream; characters sent to this stream are flushed on each output operation
`clog`	Buffered error stream; characters sent to this stream will be buffered until the buffer is flushed or becomes full

Standard Input/Output

- Include the `iostream` library
- `std::cout` used to write to standard output

```
std::cout << "Hello world!";
std::cout << std::endl;
```

- `std::cin` used to read from standard input

```
std::string name;

std::cout << "Name: ";
std::cin >> name;
```

- The `>>` operator is defined for all standard C++ types:

```
int i;
float f;
string s;

std::cin >> i;    // Converts input to int
std::cin >> f;    // Converts input to float
std::cin >> s;    // Converts input to string
```

- `<<` and `>>` are left-associative, meaning we can chain them:

```
std::cout << 1 + x;
std::cout << " ";
std::cout << 3 + y;

// equivalent to:
std::cout << 1 + x << " " << 3 + y;
```

- Recall that we can import specific members of a namespace, allowing our code to be more concise:

```cpp
#include <iostream>
#include <string>

using std::cout;
using std::cin;
using std::endl;
using std::string;

main() {
  string name;

  cout << "Name: ";
  cin >> name;
  cout << "Your name is " << name << endl;
}
```

File Streams

- The `fstream` library provides:
  - `ifstream`: provides an interface to read data from files as input streams
  - `ofstream`: provides an interface to write data to files as output streams
  - `fstream`: provides an interface to read and write data from/to files as input/output streams

```cpp
#include <fstream>
#include <iostream>
#include <string>

using std::cout;
using std::endl;
using std::ifstream;
using std::string;

main() {
  ifstream in_file;
  string str;
  in_file.open("input.txt");

  if (in_file.fail())
  {
      cout << "Unable to open file" << endl;
  }
  else
  {
    in_file >> str;
    cout << "First word from file: " << str;
    in_file.close();
  }
}
```

File Streams

- Formatted input works just as with `cin`
- This is the beauty of the C++ I/O system: same interface regardless of the type of the underlying stream

```
int year;
ifstream in_file;

in_file.open("years.txt");
in_file >> year;
in_file.close();
```

File Streams

- File output is similar:

```cpp
#include <fstream>
#include <iostream>
#include <string>

using std::cout;
using std::endl;
using std::ofstream;
using std::string;

main() {
  int year = 2012;
  ofstream out_file;
  out_file.open("output.txt");

  if (out_file.fail())
  {
      cout << "Unable to open file" << endl;
  }
  else
  {
    out_file << "Hello world!  The year is " << year << endl;
    out_file.close();
  }
}
```

String Streams

- The `sstream` library provides:
  - `istringstream` : provides an interface to manipulate strings as input streams
  - `ostringstream` : provides an interface to manipulate strings as if they were output streams
  - `stringstream` : provides an interface to manipulate strings as if they were input/output streams

istringstream

```
#include <iostream>
#include <sstream>
#include <string>

using std::cout;
using std::endl;
using std::istringstream;
using std::string;

main()
{
  string time = "5h22m";
  istringstream in_str(time);

  char ch;
  int hours;
  int minutes;

  in_str >> hours >> ch >> minutes;

  cout << hours << " hours and " << minutes << " minutes" << endl;
}
```

ostringstream

```cpp
#include <iostream>
#include <sstream>
#include <string>

using std::cout;
using std::endl;
using std::ostringstream;
using std::string;

main()
{
    int hours = 5;
    int minutes = 22;

    ostringstream out_str;  // Analogous to Java’s StringBuilder class

    out_str << hours << "h" << minutes << "m";
    string time = out_str.str();

    cout << "The time is " << time << endl;
}
```

I/O Manipulators

- The `iomanip` library provides various I/O *manipulators* that act as filters upon our input/output.
- When "injected" into a stream, these manipulators change the format or characteristics of our I/O.
- Common output manipulators:

`endl`	Write a new line character and flush buffer
`setw(n)`	Set the minimum field width
`left`	Left-justify fields
`right`	Right-justify fields
`setfill(ch)`	Used after `setw`. Sets the fill character

- See `iomanip.cpp`

Operator Overloading

- C++ allows for many operators to be overloaded in a class
- Consider a class `Time` in which we wish to be able to add two times (see `Time.h` and `Time.cpp`)
- In a binary operation (e.g. `+`, `-`, etc.), the overloaded operator method is called on the **left hand side** of the operation:

```
Time t1;
Time t2;

t1 + t2; // equivalent to t1.operator+(t2);

cout << "hi"; // equivalent to cout.operator<<("hi");
```

Overloading Stream Operators

- The stream insertion / extraction operators are defined for all standard C++ types
- We may wish to *overload* these operators in our own classes
- Suppose we have a class Rectangle, and consider what happens when we attempt to output it to a stream:

```
Rectangle r;
cout << r << endl;
```

- Compiler error:

  ```
  error: no match for operator<< in std::cout << r
  ```

Overloading Stream Operators

- We must provide an implementation of operator `<<` for the Rectangle class
- **Problem**:
  - Recall that in a binary operation, the operator method is called on the **left hand side** of an operation
  - The code `cout << r` is equivalent to the function call:

    ```
    cout.operator<<(r);
    ```

  - Thus, the `operator<<` method will be called on the `cout` object (an instance of `ostream`), passing in Rectangle `r` as an argument
  - What is the issue here?

Overloading Stream Operators

- **Solution**:
  - Use a *global* function that is not a member of any class:

```
ostream& operator<<(ostream& out, Rectangle& r)
```

  - Similarly, for input:

```
istream& operator>>(istream& in, Rectangle& r)
```

  - See `Rectangle.h` and `Rectangle.cpp`

Friend Functions

- Observe that both operator overloads in `Rectangle.h` were declared using the keyword `friend`
- When a class declares a *friend function*, it allows access to its private members within that function
- When possible, avoid using friend functions and instead provide a public interface through which a function can access the members it requires
- Exercise: refactor the code in `Rectangle.h` and `Rectangle.cpp` to remove the need for the use of friend functions.

Type-Safe I/O

- One important advantage of C++ over C is that it provides *type-safe I/O*.
- Consider the following C code:

```c
// Example of type-unsafe I/O in C
// Produces a compiler warning, but still compiles

#include <stdio.h>

typedef struct {
  int age;
  char* first_name;
  char* last_name;
} person_t;

main() {
  person_t p;

  p.age = 99;
  p.first_name = "Bob";
  p.last_name = "Caygeon";

  printf("%s\n", p);

  printf("%s\n", p.age); // Segmentation fault (why?)
}
```

Type-Safe I/O

- I/O in C is not type-safe, which can lead to unexpected, subtle, and sometimes catastrophic errors.
- C++ uses type-safe I/O.
  - Each I/O operation is executed in a manner specific to the data type.
  - If an I/O function has been defined to handle a particular data type, then that function is called to handle that data type.
  - If not, we get a compiler error
  - Thus, improper data cannot “sneak” through the system as in C

```
Person p;
cout << p << endl;
```

- Compiler error:

  ```
  error: no match for operator<< in std::cout << p
  ```