CME 211: Lecture 16

Topics

- conditionals
- basic file operations in C++

Conditional statements in C++

C++ has three conditional statements:

- `if`
- `switch`
- C++ ternary operator: `(x == y) ? a : b`

C++ `if`

```cpp
#include <iostream>

int main() {
  int n = 2;

  std::cout << "n = " << n << std::endl;
  if (n > 0) {
    std::cout << "n is positive" << std::endl;
  }

  return 0;
}
```

Output:

```
$ ./if1
n = 2
n is positive
```

Note: brackets `{...}` are not needed for a single line `if` block. However, I recommend always putting them in.

`else if`

```cpp
#include <iostream>

int main() {
  int n = -3;

  std::cout << "n = " << n << std::endl;

  if (n > 0) {
    std::cout << "n is positive" << std::endl;
  }
  else if (n < 0) {
    std::cout << "n is negative" << std::endl;
  }

  return 0;
}
```

Output:

```
$ ./if2
n = -3
n is negative
```

else

```
#include <iostream>

int main() {
  int n = 0;

  std::cout << "n = " << n << std::endl;

  if (n > 0) {
    std::cout << "n is positive" << std::endl;
  }
  else if (n < 0) {
    std::cout << "n is negative" << std::endl;
  }
  else {
    std::cout << "n is zero" << std::endl;
  }

  return 0;
}
```

Output:

```
$ ./if3
n = 0
n is zero
```

Common mistakes

Empty `if` due to extraneous semi-colon:

```
if (n < 0);
  std::cout << "n is negative" << std::endl;
```

Assignment in the conditional expression:

```
if (n = 0)
  std::cout << "n is zero" << std::endl;
```

Note: some people recommend always putting the 'literal' before the variable. This is known as a Yoda Condition.

`break`

The `break` keyword breaks out of the current loop.

```cpp
#include <iostream>

int main() {

  for (unsigned int n = 0; n < 10; n++) {
    std::cout << n << std::endl;
    if (n > 3) break;
  }

  return 0;
}
```

Output:

```
$ ./break
0
1
2
3
4
```

`continue`

The `continue` keyword moves to the next loop iteration.

```cpp
#include <iostream>

int main() {
  for (unsigned int n = 0; n < 10; n++) {
    if (n < 7) continue;
    std::cout << n << std::endl;
  }

  return 0;
}
```

Output:

```
$ ./continue
7
8
9
```

Logical operators

- C++ has two choices for logical operators
- Newer style `and`, `or`, `not`
- Older style `&&`, `||`,
- Latter are backwards compatible with C

Logical AND

```
#include <iostream>

int main() {
  int a = 7;
  int b = 42;

  // the following are equivalent

  if (a == 7 and b == 42)
    std::cout << "a == 7 and b == 42 is true" << std::endl;

  if (a == 7 && b == 42)
    std::cout << "a == 7 && b == 42 is true" << std::endl;

  return 0;
}
```

Output:

```
$ ./logical1
a == 7 and b == 42 is true
a == 7 && b == 42 is true
```

`0` is false, everything else is true

```
#include <iostream>

int main() {
  int a[] = {-1, 0, 1, 2};

  for (int n = 0; n < 4; n++) {
    if (a[n])
      std::cout << a[n] << " is true" << std::endl;
    else
      std::cout << a[n] << " is false" << std::endl;
  }

  return 0;
}
```

Output:

```
$ ./logical2
-1 is true
0 is false
1 is true
2 is true
```

Bitwise results

```
#include <iostream>
```

```
int main() {
  int a = 1;
  int b = 2;

  if (a)
    std::cout << "a is true" << std::endl;
  else
    std::cout << "a is false" << std::endl;

  if (b)
    std::cout << "b is true" << std::endl;
  else
    std::cout << "b is false" << std::endl;

  if (a & b)
    std::cout << "a & b is true" << std::endl;
  else
    std::cout << "a & b is false" << std::endl;

  return 0;
}
```

Output:

```
$ g++ -Wall -Wconversion -Wextra logical3.cpp -o logical3
$ ./logical3
a is true
b is true
a & b is false
```

switch

- `if`, `else if`, `else`, etc. gets verbose if you have many paths of execution
- Can use a switch statement instead:

```
if (choice == `C')
  clearRecord();
else if (choice == `D')
  deleteRecord();
else if (choice == `A')
  addRecord();
else if (choice == `P')
  printRecord();
else
  std::cout << "Bad choice\n";
```

Becomes:

```
switch (choice) {
  case `C': clearRecord(); break;
  case `D': deleteRecord(); break;
  case `A': addRecord(); break;
  case `P': printRecord(); break;
  default: std::cout << "Bad choice\n";
}
```

`switch` and `enum` example

```
enum direction {
  left,
  right,
  up,
  down
};

int main() {
  direction d = right;

  std::string txt = "you are going ";
  switch (d) {
    case left:
      txt += "left"; break;
    case right:
      txt += "right"; break;
    case up:
      txt += "up"; break;
    case down:
      txt += "down"; break;
  }
  std::cout << txt << std::endl;
  return 0;
}
```

Output:

```
$ ./switch1
you are going right
```

Advantage

Compiler warnings will tell you if you are missing some cases.

```
switch (d)
{
  case left:
    txt += "left"; break;
  case right:
    txt += "right"; break;
  case down:
    txt += "down"; break;
}
```

Output:

```
$ g++ -Wall -Wconversion -Wextra switch2.cpp -o switch2
switch2.cpp: In function 'int main()':
switch2.cpp:16:10: warning: enumeration value 'up' not handled in switch [-Wswitch]
switch (d)
^
```

Common mistake

Neglecting to add `break` in each case.

```
std::string txt = "you are going ";
switch (d) {
  case left:
    txt += "left";
  case right:
    txt += "right";
  case up:
    txt += "up";
  case down:
    txt += "down";
}
std::cout << txt << std::endl;
```

Output:

```
$ g++ -Wall -Wconversion -Wextra switch3.cpp -o switch3
$ ./switch3
you are going rightupdown
```

Ternary operator

This is called the “ternary” operator:

```
a = b < 0 ? -b : b;
```

Equivalent code:

```
if (b < 0)
  a = -b;
else
  a = b;
```

Anatomy:

```
[conditional] ? [return expression if true] : [return expression if false];
```

`goto`

“If you find yourself using a `goto` statement within a program, then you have not thought about the problem and its implementation for long enough”

See: http://xkcd.com/292/

Figure 1: fig

File I/O

File I/O in Python:

File I/O in C++:

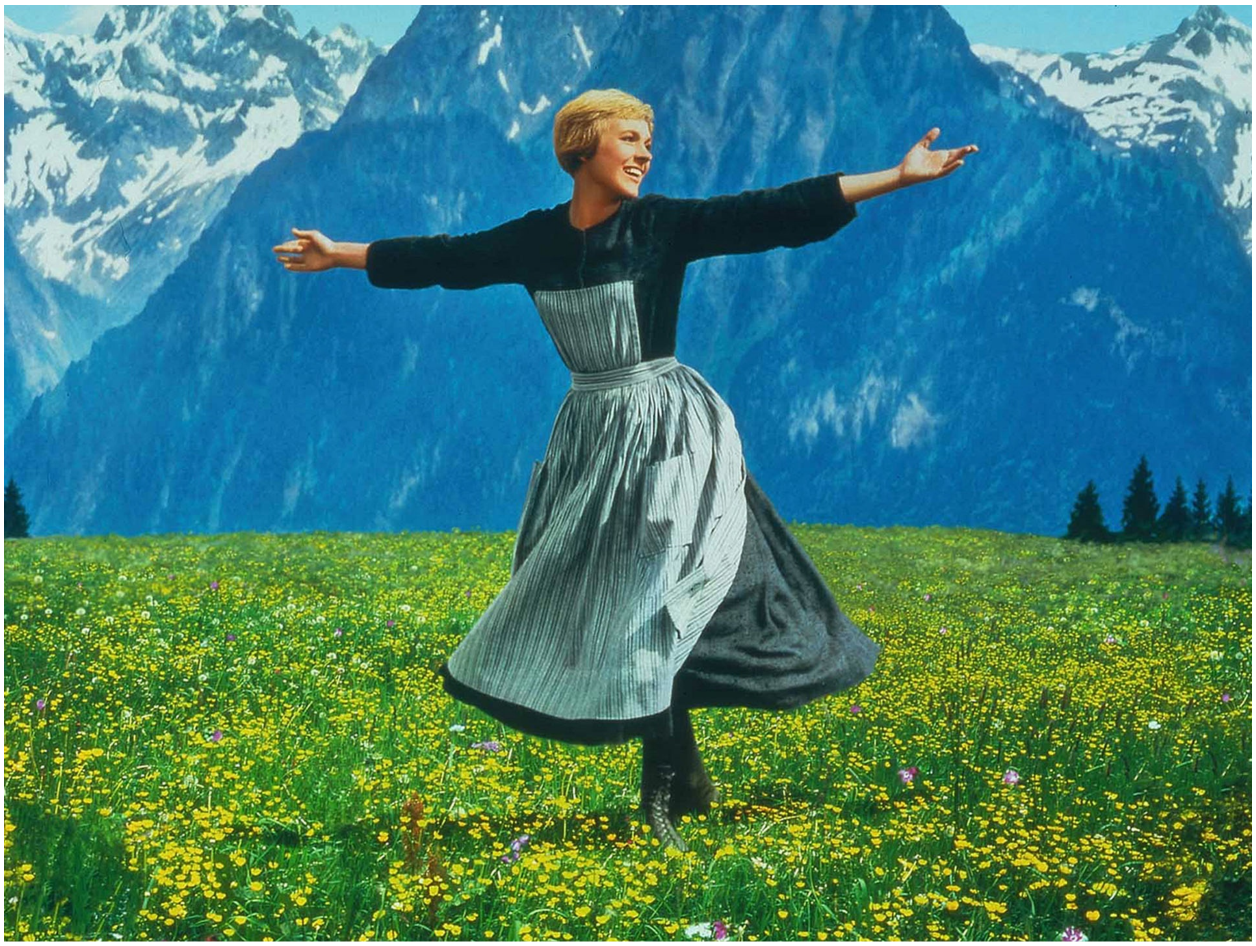

Figure 2: File I/O in Python

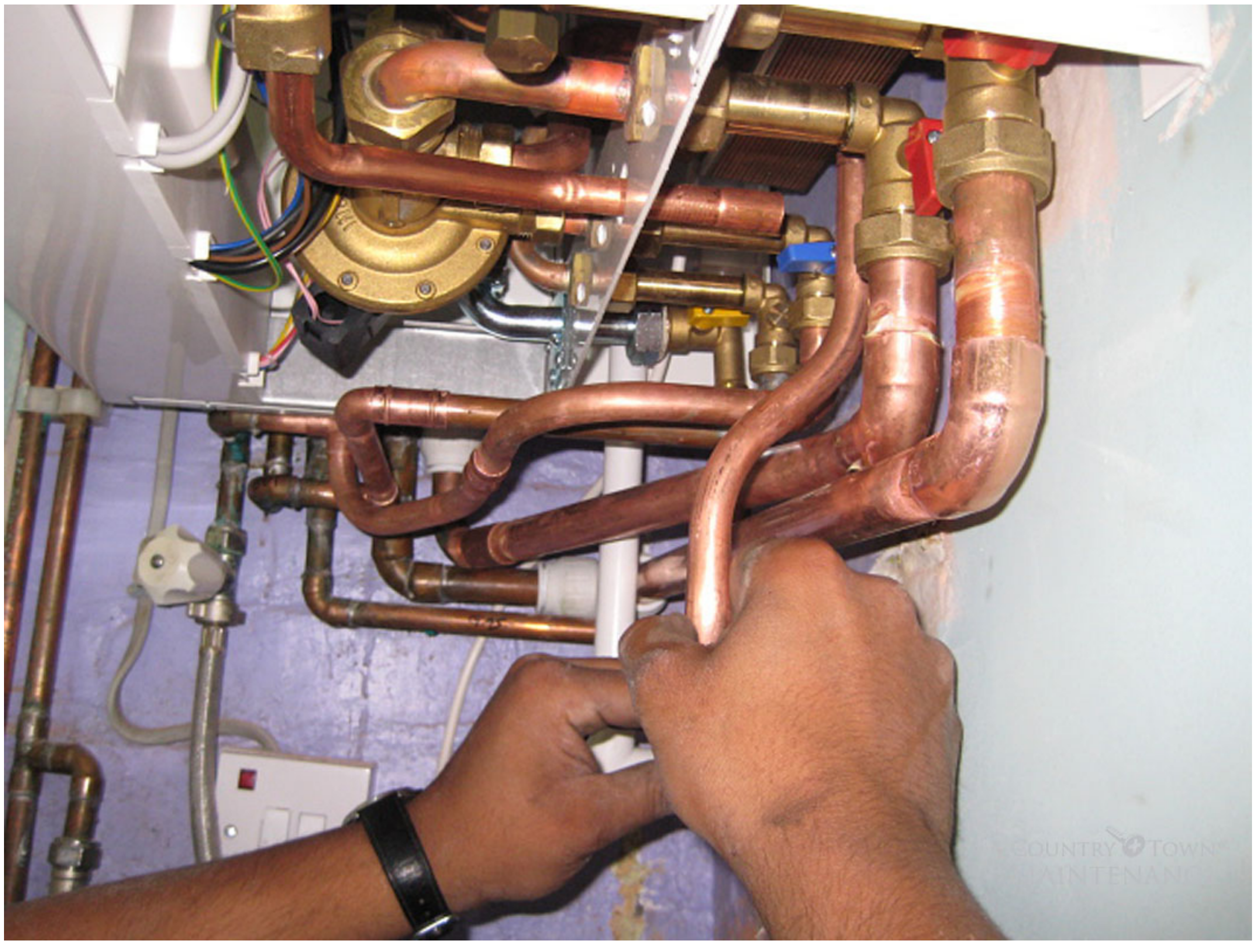

Figure 3: File I/O in C++

C++ file I/O

- Like outputting to the screen, file I/O is also handled via streams
- Three stream options:
- `ofstream`: output file stream (i.e. write)
- `ifstream`: input file stream (i.e. read)
- `fstream`: file stream (i.e. read or write)

`ofstream`

```
#include <iostream>
#include <fstream>

int main() {
  std::ofstream f;

  f.open("hello.txt");
  if (f.is_open()) {
    f << "Hello" << std::endl;
    f.close();
  }
  else {
    std::cout << "Failed to open file" << std::endl;
  }

  return 0;
}
```

Output:

```
$ g++ -Wall -Wconversion -Wextra ofstream1.cpp -o ofstream1
$ rm -f hello.txt
$ ./ofstream1
$ cat hello.txt
```

Using a variable for the filename

Code:

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

int main() {
  std::string filename = "file.txt";

  std::ofstream f;
  f.open(filename);
  if (f.is_open()) {
    f << "Hello" << std::endl;
    f.close();
  }
  else {
    std::cout << "Failed to open file" << std::endl;
  }

  return 0;
}
```

Output:

```
$ g++ -Wall -Wconversion -Wextra ofstream2.cpp -o ofstream2
ofstream2.cpp: In function 'int main()':
ofstream2.cpp:10:18: error: no matching function for call to
'std::basic_ofstream<char>::open(std::string&)'
f.open(filename);
^
ofstream2.cpp:10:18: note: candidate is:
In file included from ofstream2.cpp:2:0:
/usr/include/c++/4.8/fstream:713:7: note: void std::basic_ofstream<_CharT,
_Traits>::open(const char*, std::ios_base::openmode) [with _CharT = char; _Traits =
std::char_traits<char>; std::ios_base::openmode = std::_Ios_Openmode]
open(const char* __s,
^
/usr/include/c++/4.8/fstream:713:7: note:
no known conversion for argument 1 from
'std::string {aka std::basic_string<char>}' to 'const char*'
```

Change to:

```
  f.open(filename.c_str());
```

Output:

```
$ g++ -Wall -Wconversion -Wextra ofstream3.cpp -o ofstream3
$ rm -f file.txt
$ ./ofstream3
$ cat file.txt
```

C++ 2011 standard

Specify usage of the C++ 2011 standard. Passing an `std::string` to `f.open` is supported:

```
g++ -std=c++11 -Wall -Wconversion -Wextra ofstream2.cpp -o ofstream2
rm -f file.txt
./ofstream2
cat file.txt
```

Writing an array of values

```
#include <iostream>

//  Define constants to size the static array
#define ni 2
#define nj 3

int main() {
  int a[ni][nj];
  // Initialize the array values
  int n = 0;
  for (int i = 0; i < ni; i++) {
    for (int j = 0; j < nj; j++) {
      a[i][j] = n;
      n++;
    }
  }

  // Store the array values in a file
  std::ofstream f("array.txt");
  if (f.is_open()) {
    f << ni << " " << nj << std::endl;
      for (int i = 0; i < ni; i++) {
        f << a[i][0];
        for (int j = 1; j < nj; j++) {
          f << " " << a[i][j];
        }
        f << std::endl;
      }
    f.close();
  }
  return 0;
}
```

fstream

```
#include <iostream>
#include <fstream>

int main() {
  std::fstream f;

  // specify output mode with second argument
  f.open("hello.txt", std::ios::out);
  if (f.is_open()) {
    f << "Hello" << std::endl;
    f.close();
  }
  else {
    std::cout << "Failed to open file" << std::endl;
  }

  return 0;
}
```

Reading from a file

- Not as easy or convenient as in Python
- We will start by looking at how to read the simple array file we previously wrote

ifstream

```
#include <iostream>
#include <fstream>

int main() {
  // Read the array values from the file
  std::ifstream f("array.txt");
  if (f.is_open()) {
    int i;
    while (f >> i) { // Stream extraction operator
      std::cout << i << std::endl;
    }
    f.close();
  }
  return 0;
}
```

Output:

```
$ g++ -std=c++11 -Wall -Wconversion -Wextra ifstream1.cpp -o ifstream1
$ ./ifstream1
2
3
0
1
2
3
4
5
```

Reading the array

```
// Read the array values from the file
std::ifstream f("array.txt");

if (f.is_open()) {
  // Read the size of the data and make sure storage is sufficient
  int nif, njf; // Values of ni and nj read to be read from file
  f >> nif >> njf;
  if (nif > ni or njf > nj) {
    std::cout << "Not enough storage available" << std::endl;
    return 0; // quit the program
  }

  // Read the data and populate the array
  for (int i = 0; i < nif; i++) {
    for (int j = 0; j < njf; j++) {
      f >> a[i][j];
    }
  }
  f.close();
}
```

Reading

- **C++ Primer, Fifth Edition** by Lippman et al.
- Chapter 1: Statements: Sections 5.3 - 5.5
- Chapter 8: The IO Library: Section 8.2