Recursion (strings + arrays)

Oct. 27, Ch 7.4-8.2 + 14
Announcements

- midterm graded?

- HW 4 due tomorrow

- for each loops (new in C++ 11)

```cpp
int x[] = {1, 4, 5, 2};

for(int a : x)
{
  // lab computers don't seem to like them
  (g++ -std=c++11 fileName.cpp)
```
Highlights

- multi-dimensional arrays

```c
int x[10][20]; // 10 rows, 20 columns of fun!
```

- recursion

```c
int main()
{
    cout << "HI\n!";
    main();
}
```
Multidimensional Arrays

So far we have dealt with simple (one dimensional) arrays

We have represented this as all the data being stored in a line

(See: lineWorld.cpp)
Multidimensional Arrays

```java
int foo[][] = new int[3][5];
```

- `foo`'s length = 3 (number of rows)
- `foo[0]`'s length = 5 (number of columns in row 0)
Multidimensional Arrays

If we think of a couple simple (one dimensional) arrays on top of each other...

(See: gridWorld.cpp)
Multidimensional Arrays

Recreate:

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
<td>88</td>
<td>89</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>91</td>
<td>92</td>
<td>93</td>
<td>94</td>
<td>95</td>
<td>96</td>
<td>97</td>
<td>98</td>
<td>99</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

(See: oneToAHundred.cpp)
C-Strings and strings

There are actually two types of “strings” (multiple characters) in C++

A **C-String** is a char array, and this is what you get when you put quotes around words

```cpp
cout << "HI!\n";  // C-String
```

A **string** (the thing you #include) is a more complicated type called a **class** (next week)
Last lab, you basically worked with C-Strings (char arrays)

```c
char word[] = {'o', 'm', 'g', '\0'};
```

You should end C-Strings with null character, as this tells cout when to stop displaying.

This means you can initialize char arrays with quotes **(BUT NOT OTHER TYPES)** (see: cstring.cpp)
C-Strings and strings

The string class is a char array packaged up with some more functionality.

```cpp
string favFood = "anything free";
```

Strings are nice as they are more flexible:

- `+`: concatenates (mashes together) strings
- `[]`: can be used to index into them (as arrays)
- `==`: can be used to compare equality

(see: string.cpp)
C-Strings and strings

It is fairly easy to convert between C-Strings and strings:

```cpp
char cString[] = "move zig";
string IMAString = cString;
cout << IMAString.c_str() << endl;
// above converts it back to C-String
```

You can also convert between numbers and strings:

```cpp
char number1[20];
string number2;
cin >> number1 >> number2;
cout << "sum is: " << (atof(number1) + stod(number2)) << endl;
```
(see: stringConversion.cpp)
Recursion

Recursion is simply when a function calls itself (we did this for the maze in week 5)

This is quite powerful, but also confusing

(see: towerHanoi.cpp)
Recursion

There are two important parts of recursion:
- A stopping case that ends the recursion
- A base case that reduces the problem

What are the base and stopping cases for the Fibonacci numbers?

\[ F_n = F_{n-1} + F_{n-2}, \]
\[ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots \]

(sum of the previous two numbers)
(see: fibonacciRecursion.cpp)
Recursion

If you forget your stopping case, you will not get an infinite loop but crash the program. This is because every function call takes up more memory, so you constantly ask for more memory.

Eventually the memory (stack) cannot store anymore (see: stackOverflow.cpp)
How would you sum the numbers 1 to n using recursion (not a loop)?

For example sumToN(5) = 15, as 1+2+3+4+5 = 15

What is the stopping case?
How do you reduce the problem?

(see: sumToN.cpp)
Recursion

What is the difference between:

```cpp
void tail(int x) {
    if (x == 0) {
        return;
    } else {
        tail(x - 1);
        cout << x << " ";
    }
}
```

```cpp
void head(int x) {
    if (x == 0) {
        return;
    } else {
        cout << x << " ";
        head(x - 1);
    }
}
```

(see: headTailRecursion.cpp)
Recursion

How would you solve a sudoku problem?

Rules:
1. Every row has numbers 1-9
2. Every column has numbers 1-9
3. The nine 3x3 boxes have numbers 1-9

Reduce problem?
Stopping case?

(see: sudokuSolver.cpp)
Recursion

Do not try to solve chess in this manner!

You will segfault
(you will also not finish computing before the sun burns the earth to a crisp)