21210 (SCIENTIFIC COMPUTING LAB) NOTES

OYEKOLA OYEKOLE

Abstract. Notes for the Scientific Computing lab scheduled for Feb 27, 2017

1. More about strings, with examples

- **Converting a number to a string.**

```
int number = 42; // the answer to the universe!
string foobar = to_string(number); //foobar will be a string "42"
```

- **Length of a string.**

```
string name = "Einstein";
cout << name.length(); //outputs 8
```

- **Copying a string.**

```
string me = "Agent Smith";
string me_too(me) ; // me_too is now also "Agent Smith"
```

- **Comparing two strings.** Given two strings (name and lastname), then

```
int equality = name.compare(lastname); //this will return 0 if equal
bool is_equal = (name==lastname); // another way to compare them
```

- **String concatenation (joining).**

```
string first = "Out of many,";
string second = " one.";
cout << first+second << endl; // outputs "Out of many, one."
```

- **Sub-strings.** Get a portion of an existing string. The string is indexed from 0. The first argument will specify where to begin the sub-string. The second (optional) argument will specify how many characters to include in the substring.

```
string the_word = "Unimaginatively";
cout << the_word.substr(2) << endl; // outputs "imaginatively"
cout << the_word.substr(7,6) << endl; // outputs "native"
```

- **Iterating on a string.** `string.begin()` and `string.end()` refer to the iterators for the first character, and for the last character of the string respectively. We will see an example shortly.

2. Recursion: self-calling functions

"To iterate is human, to recurse divine".- L. Peter Deutsch.

- **Know when to iterate.** In practice, you should hesitate to use recursion because it is only better than iteration - in maybe less than 1% of cases. Iteration is always better whenever a recursion will involve a wastage of computing resources. e.g. to compute e^x using the first $n+1$ terms of its Taylor series expansion, we have

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}.$$

Date: February 26, 2018.

If we already have a function `factorial`, and we have included the `cmath` header library, then we may choose to use recursion here.

```cpp
double e_to_x(double x, int n){
  if (n==1){
     return 1.0;
  }
  return pow(x,n*1.0)/factorial(n) + e_to_x(x, n); //A recursion
}
```

The code works, but it's not at all good code! This is because we have multiple repeated calculations of intermediate factorials and powers all the way down i.e.

$$
\begin{aligned}
& e^x[\text{using n terms}] = x^n/n! + e^x[\text{using n-1 terms}] \\
\textit{where} \quad & e^x[\text{using n-1 terms}] = x^{n-1}/(n-1)! + e^x[\text{using n-2 terms}] \\
\textit{where} \quad & e^x[\text{using n-2 terms}] = x^{n-2}/(n-2)! + e^x[\text{using n-3 terms}], \cdots \\
& \cdots\cdots\cdots\cdots\cdots\cdots\cdots \quad \text{where} \quad e^x[\text{using 1 term}] = 1
\end{aligned}
$$

However if we use iteration, we can re-use the values that we already calculated.

```cpp
double good_e_to_x(double x, int n){
  double result=0.0, term=1.0;
  for (int i=1; i<= n; i++){
    term *= x/i; // Each term. This is a double divided by an int.
    result += term; // Add the term to the series
  }
  return result;
}
```

This is a lot faster, saves memory and also we didn't need a factorial function.

- **If you must use recursion, beware of infinite recursion.** Always ensure you have a well-defined stopping condition, and/or a well-defined base case. e.g. in the recursion for e^x above, the base case / stopping condition is when $n = 1$. Such a base case must not also depend on the recursion function or else it will run forever.

3. Class Exercise

What is the output of the program below?

```cpp
#include <iostream>
using namespace std;

void cheers(int n);

int main(){
  cheers(3);
  return 0;
}

void cheers(int n){
  if (n == 1){
     cout << "Hurray!\n" << endl;
  }else{
     cout << "Hip ";
     cheers(n-1);
  }
  return;
}
```

4. Class Example 1

Write a C++ program that requests a non-negative whole number and counts its digits. Ensure your code works for numbers like 0 and 12345 as well as padded numbers like 007 (in the last case, the number of digits is simply 1).

We will consider three ways to solve the problem. Below are some ideas.

- The number of digits in a non-negative integer is the number of times we can have an integer division of 10 into the number.
- The number of digits in a non-negative integer is the base 10 logarithm of the number, plus 1.
- If we convert a non-negative integer to a string, the number of digits will be the length of that string.

```cpp
#include <iostream>
#include <cmath>
using namespace std;

int method_one(int number);
int method_two(int number);
int method_three(int number);

int main(){
  int number=-1; // Forcing a request for the integer for the first time
  while (number < 0){
     cout << "Enter a non-negative integer to count its digits." << endl;
     cin >> number;
  }
  cout << endl;
  cout << "Number of digits" << endl;
  cout << "\t Method one: " << method_one(number) << endl;
  cout << "\t Method two: " << method_two(number) << endl;
  cout << "\t Method three: " << method_three(number) << endl;
  return 0;
}

int method_one(int number){
  int count=0;
  if (number == 0) return 1;
  while (number){ // This while loop continues until number is 0
    number /= 10;
    count++;
  }
  return count;
}

int method_two(int number){
  if (number == 0) return 1;
  return (int) log10((double) number) + 1;
}

int method_three(int number){
  return to_string(number).length();
}
```

5. Class Example 2

Below is a C++ program that accepts two words one after the other, and reports if they are correct string reversals. e.g. "code" and "edoc" are string reversals. If they are not correct string reversals, the program will display the correction. We will also see the use of an in-built reversal function (needs to include algorithm header file) for the same purpose.

```cpp
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

string reverse_me(string word);
bool is_good_reversal(string word1, string word2);
void reverse_internal(string word1, string word2);

int main()
{
  string word1, word2;
  cout << "Enter the first word:" << endl;
  cin >> word1;
  cout << "Enter the second word:" << endl;
  cin >> word2;
  cout << endl;

  if (is_good_reversal(word1, word2)){
    cout << word1 << " and " << word2 << " are string-reverses." << endl;
  }else{
    cout << "The reverse of " << word1 << " is " << reverse_me(word1);
    cout << ", and reverse of " << word2 << " is " << reverse_me(word2) <<
    endl;
  }
  cout << endl;
  cout << "Using an in-built function, we have: " << endl;
  reverse_internal(word1, word2);
  return 0;
}

string reverse_me(string word){
  string reverse_word="";
  int len = word.length();
  for (int i=len; i>=0; i--){ // start from last character, reverse till 0
      reverse_word += word.substr(i,1); // single character substrings
  }
  return reverse_word;
}

bool is_good_reversal(string word1, string word2){
    return (reverse_me(word1)==word2);
    //return !word1.compare(word2); // NOT compare = nonzero i.e. equality
    // compare is more effective here as there's no need to call reverse_me
}

void reverse_internal(string word1, string word2){
  string my_word1(word1); // copying the word so as not to really modify it
  reverse(my_word1.begin(),my_word1.end()); // syntax of reverse function

  string my_word2(word2);
  reverse(my_word2.begin(),my_word2.end());

  cout << "The reverse of " << word1 << " is " << my_word1;
  cout << " and the reverse of " << word2 << " is " << my_word2 << endl;
  cout << endl;
  return; // This is how to use a return for a void function
}
```