PIC 10A 1C. Week 9b Notes. TA: Eric Kim. [Solutions]

Functions

A function (or procedure) is a construct that allows you to break up a program into logical parts. A function may have input parameters (or arguments), and can return a value:

```
/**
 * Outputs the sum of the integers inside of a vector.
 * @param v The input vector containing ints.
 * @return Returns the sum of the integers within the vector v.
 **/
int vecsum(const vector<int>& vec) {
    int sum = 0;
    for (size_t i = 0; i < vec.size(); ++i )
        sum += vec[i];
    return sum;
}
```

One reason to use functions is to avoid needlessly repeating code. For instance, suppose we wanted to sum the values of two int vectors:

```
/* Without functions (Bad) */
vector<int> v1 = {1, 2, 3};
vector<int> v2 = {6, 10};
int sum1 = 0;
for (size_t i=0; i<v1.size(); ++i)
    sum1 += v1[i];
int sum2 = 0;
for (size_t i=0; i<v2.size(); ++i)
    sum2 += v2[i];
```

```
/* With functions (Good) */
vector<int> v1 = {1, 2, 3};
vector<int> v2 = {6, 10};
int sum1 = vecsum(v1);
int sum2 = vecsum(v2);
```

By using the vecsum function, we are able to shorten the code and make it more understandable. Breaking up code into logical functions is an effective way to make it more readable and less prone to bugs.

Function Scopes

Consider the above vecsum function. Within the vecsum function body, the only variables visible are:

1. Local variables, ie "sum"
2. Function parameters, ie "vec"
3. Global variables, ie "cout"

Functions are allowed to call other functions, for instance:

```
int square(const int x) {
    return x*x;
}
void vecsquare(vector<int>& vec) {
    for (size_t i = 0; i < vec.size(); ++i) {
        vec[i] = square(vec[i]);
    }
}
```

Recall that the compiler reads code from top-to-bottom (ie beginning to end). Take care that you declare/define a function before you use it though. For instance:

```
// Compile Error: square is used
// before it is defined
void vecsquare(vector<int>& vec) {
  for(size_t i=0; i<vec.size();++i) {
    vec[i] = square(vec[i]); // Error!
  }
}
int square(int x) {
  return x*x;
}
```

```
// This is OK! We declare square
// first, then we use it.
int square(int x); // Declare it first
void vecsquare(vector<int>& vec) {
  for(size_t i=0; i<vec.size(); ++i) {
      vec[i]=square(vec[i]);
  }
}
int square(int x) { // Define square
    return x*x;
}
```

Reference Parameters

In C++, unless otherwise specified, a function will first make a copy of all of its arguments, and then operate only on its copies. For instance, consider the vecsquare1 and vecsquare2 functions:

```
void vecsquare1(vector<int> vec) {
  for(size_t i=0; i<vec.size(); ++i) {
    vec[i] = vec[i]*vec[i];
  }
}
```

```
void vecsquare2(vector<int>& vec) {
  for(size_t i=0; i<vec.size(); ++i) {
    vec[i] = vec[i]*vec[i];
  }
}
```

```
int main() {
    vector<int> v1 = {2, 4, 5};    vector<int> v2 = {6, 7};
    vecsquare1(v1); // v1 is still: [2, 4, 5]
    vecsquare2(v2); // v2 is now: [36, 49]
    return 0;
}
```

vecsquare1 makes a ***copy*** of the input vector - thus, it does not modify the vector v1 that is passed in. Instead, it modifies its own local copy of v1, but this copy is destroyed when the function returns.
However, since vecsquare2 defines its input argument as a reference parameter, this means that "vec" is a reference variable that points to the vector v2 we passed in - thus, any modifications to "vec" also modify v2.

Rule of Thumb: If you pass an object (ie vector, string) to a function, then always pass it by reference. This is to avoid having to make a copy of the object each time you call the function. For instance, suppose you had a vector<int> with a billion elements. This array takes up 4 GB of memory! Copying it will take a noticeable amount of time, as well as take up a ton of memory:

```
/* Pass by value (copy is made) */
void foo(vector<int> v) {
    cout << v[0];
}
```

```
/* Pass by reference (no copy made) */
void bar(vector<int>& v) {
    cout << v[0];
}
```

```
int main() {
    /* Note: Don't try this on your laptop unless you have >8 GB of RAM! */
    vector<int> myvec(1e9); // Creates vector with 1 billion elements
    foo(myvec); // Has to copy myvec, adds several seconds of overhead,
                // and also takes up an additional 4 GB of RAM!
    bar(myvec); // No copy made, no overhead!
    return 0;
}
```

const Parameters

If a function will not modify an input argument, it's considered good practice to declare it as const. This is for several reasons:

1. Documentation: by seeing the "const' keyword, a programmer will immediately know that this function will not modify the variable.
2. Compiler Aid: by telling the compiler that a variable is const, the compiler will help you out by detecting if the function ends up modifying the const variable.

For instance, suppose I wanted to write a function vecprint() that displays all elements of a vector. This function shouldn't modify the input vector at all - thus, I should declare the input vector argument as const:

```
void vecprint(const vector<int>& vec) {
    cout << "[";
    for (int x : vec)
        cout << x;
    cout << "]" << endl;
}
```

If I later use this function in my code, I am **guaranteed** that the function will never modify the input vector because the parameter was declared const:

```
    vector<int> myvec = {3, 1, 4, 1, 5};
    vecprint(myvec);
    // myvec is guaranteed to still be: [3,1,4,1,5]
```

Function Overloading

Sometimes, it makes sense to define the same function but for different input types. For instance, suppose we wanted to define a get_size() function that worked on both strings and vectors. Here are two ways - one that uses function overloading, and one that does not:

```
/* With function overloading */
size_t get_size(string s) {
    return s.size();
}
size_t get_size(vector<int> v)
    return v.size();
}
```

```
/* Without overloading */
size_t get_size_str(string s) {
    return s.size();
}
size_t get_size_vec(vector<int> v) {
    return v.size();
}
```

One benefit of overloading is we don't have to define a separate function name for each type.

Be careful that you don't introduce ambiguous overloading, ie defining two functions for the same input types:

```
int foo(int x, int y) {
    return x + y;
}
int foo(int x, int y) {
    return x*x + x*y + y;
}
```

This is a compiler error, since there are two conflicting function definitions: if you call "`foo(3, 4)`", the compiler doesn't know which function to call!

Note that the compiler only considers function input types for overloading. For instance, the following function declarations are also invalid:

```
    int g(int x);
    string g(int x); // Bad! Compiler can't distinguish the g's apart
    int h(int x);
    string h(char x); // Ok! Different input types.
```

Exercises

1. To overload, or not to overload...

For the following function definitions, determine the output of the program. If an error occurs, explain. Assume all headers are included, and the standard namespace is used. The first is done for you:

`int fizzle(char x) {`
`  return 4;`
`}`
`int fizzle(int x) {`
`  return 8;`
`}`
	`cout << fizzle('c') << endl;`
	`cout << fizzle(2) << endl;`
	Output:
	4
	8

`int foo(int a, string s) {`
`  if (s[0] == 'a')`
`  return a + 1;`
`  return 0;`
`}`
`int foo(int a, int b) {`
`  return a + 2*b;`
`}`
	`cout << foo(1, 2) << endl;`
	`cout << foo(10, string("apple")) << endl;`
	Output:
	[Solution]
	5
	11

`int bar(int a) {`
`  if ((a % 2) == 0)`
`  return a + 1;`
`  return 0;`
`}`
`int bar(int a) {`
`  return a + 2*b;`
`}`
	`cout << bar(2) << endl;`
	`cout << bar(3) << endl;`
	Output:
	[Solution]
	CompileError - bar is defined twice for same input types.

`int g(int x) {`
`  return x + 1;`
`}`
`string g(char x) {`
`  return string("hi") + x;`
`}`
	`cout << g(42) << endl;`
	`cout << g('t') << endl;`
	Output:
	[Solution]
	43
	hit

`int baz(int x, double y) {`
`  return x + 2*y;`
`}`
`int baz(int x, int y) {`
`  return x*x + y;`
`}`
	`cout << baz(3, 4) << endl;`
	`cout << baz(3, 2/4) << endl;`
	`cout << baz(3, 8.0 / 4) << endl;`
	Output:
	[Solution]
	13
	9
	7

2. Fixin' a Hole (in my code)

During a late-night code session -- fueled by Red Bull and pizza -- Louis Reasoner wrote a program that, while correct, is not ideal. How would you rewrite the code so that it uses functions where appropriate? What functions would you define, if any?

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;
int main() {
  /* Given a bunch of names of the form "LASTNAME, FIRSTNAME",
   * output the first names on each line. */
  vector<string> beatles = { "McCartney, Paul", "Harrison, George",
     "Lennon, John", "Starr, Ringo" };
  for (string name : beatles) {
    /* (1) Find the index of the comma */
    size_t comma_ind = 0;
    for (size_t i = 0; i < name.size(); ++i) {
      if (name[i] == ',') {
        comma_ind = i; break;
      }
    }
    /* (2) Output the first name */
    cout << name.substr(comma_ind + 2) << endl;
  }
  vector<string> dudes = { "Chopin, Frederic", "Beethoven, Ludwig",
     "Liszt, Franz", "Barber, Samuel" };
  for (string name : dudes) {
    /* (1) Find the index of the comma */
    size_t comma_ind = 0;
    for (size_t i = 0; i < name.size(); ++i) {
      if (name[i] == ',') {
        comma_ind = i; break;
      }
    }
    /* (2) Output the first name */
    cout << name.substr(comma_ind + 2) << endl;
  }
  return 0;
}
```

[Solution]

There are several ways to restructure the code, but here's one straightforward way. The key idea is to identify that there are several opportunities to logically break down the code into subfunctions.

Note that this is more of an art than a science, but it's a good skill to pick up, as well-structured code tends to be easier to read and maintain.

There is one wart, where find_char() returns 0 if c is not present in s - but for now we'll just assume that there always is a comma in each name.

```
/**
 * Finds the index of character c in the string s. Assumes that c is
 * present within s.
 **/
size_t find_char(const string& s, const char c) {
  for (size_t i = 0; i < s.size(); ++i) {
    if (s[i] == c)
      return i;
  }
  return 0; // Not ideal, but we assume it always is found
}
/**
 * Given a name of the form "LASTNAME, FIRSTNAME", returns the first name.
 **/
string get_firstname(const string& name) {
  size_t comma_ind = find_char(name, ',');
  return name.substr(comma_ind+2);
}
/**
 * Given a vector of names of the form "LASTNAME, FIRSTNAME", displays
 * all of the first names.
 **/
void display_firstnames(const vector<string>& names) {
  for (string name : names)
    cout << get_firstname(name) << endl;
}
int main() {
  /* Given a bunch of names of the form "LASTNAME, FIRSTNAME",
   * output the first names on each line. */
  vector<string> beatles = { "McCartney, Paul", "Harrison, George",
     "Lennon, John", "Starr, Ringo" };
  display_firstnames(beatles);
  vector<string> dudes = { "Chopin, Frederic", "Beethoven, Ludwig",
     "Liszt, Franz", "Barber, Samuel" };
  display_firstnames(dudes);
  return 0;
}
```

3. Just sorting some things out...

Write a function is_sorted that, given a vector of chars, returns true if the vector is in lexicographic order, and false otherwise:

```
    vector<char> v1 = {'a', 'b', 'b', 'c', 'z'};
    cout << is_sorted(v1) << endl; // Outputs: 1
    vector<char> v2 = {'c', 'a', 'z'};
    cout << is_sorted(v2) << endl; // Outputs: 0
```

[Solution]

A natural way to approach this is to compare the current char to the next char, taking care to only go up to (vec.size()-1). Also, note that we declare the input vector to be a const reference.

```
bool is_sorted(const vector<char>& vec) {
    for (size_t i = 0; i < (vec.size()-1); ++i) {
        if (vec[i] > vec[i+1])
            return false;
    }
    return true;
}
```

4. Flip-floppin'

Write a function vec_reverse that, given a vector of ints, reverses the order:

```
    vector<int> myints = {8, 6, 7, 5, 3, 0, 9};
    vec_reverse(myints); // myints is now: [9, 0, 3, 5, 7, 6, 8]
```

[Solution]

Here, we iteratively swap each element. I am very careful to stop at the "halfway" point, otherwise the code would start undo-ing its previously-swapped changes! For instance, for a vector with even-length, ie [1, 3, 2, 0], we should stop at (vec.size()/2) = 2. For a vector of odd-length, ie [4, 2, 1], we should stop at (vec.size()/2) = 1.

```
void vec_reverse(vector<int>& vec) {
    for (size_t i = 0; i < (vec.size()/2); ++i) {
        /* indswap: index to swap with */
        size_t indswap = vec.size() - i - 1;
        int tmp = vec[i];
        vec[i] = vec[indswap];
        vec[indswap] = tmp;
    }
}
```