Practice test for midterm 2

March 25, 2019

1 Functions

▶ Write a function which takes in two `int` parameters and returns their *average*. (Remember that if a function takes in parameters, it does not need to use `cin`, and if it returns a value it does not need `cout`.) Write the implementation (definition) of this function, write its declaration, and write an example of a function call using this function.

```
// Declaration
int avg(int x, int y);

// Definition
int avg(int x, int y) {
    return (x + y) / 2;
}

// Function call
cout << avg(5,7) << endl;
```

▶ Write a function which prompts the user to enter a *positive* (> 0) integer and which returns the value the user entered. If the user does not enter a positive integer, the function should return `0`.

```
int read_pos() {
    int x;
    cout << "Enter a positive integer: ";
    cin << x;
    if(x > 0)
        return x;
    else
```

```
        return 0;

    // Could also be written shorter as
    // return x > 0 ? x : 0;
}
```

► Write a function which prompts the user to enter a *positive* (> 0) integer and which returns the value the user entered. If the user does not enter a positive integer, the function should use a loop to repeatedly prompt the user until they do.

```
int read_pos() {
    int x = -1;
    while(x <= 0) {
        cout << "Enter a positive integer: ";
        cin >> x;
    }
    return x;
}
```

You could also use a do-while loop for this.

► Write a function which takes a `char` parameter and returns `true` if it is a numeric character (`0` through `9`) and false otherwise.

```
bool is_numeric(char c) {
    return c >= '0' && c <= '9';
}
```

(This function is available built-in with `#include <cctype>` as `isdigit()`.)

► Using the function from the previous, write a function which takes a `string` parameter and returns true if every character in it is numeric (use a loop).

```
// Note: this is an *overload* of the previous function
bool is_numeric(string s) {
    for(int i = 0; i < s.length(); ++i)
        if(!is_numeric(s.at(i)))
            return false;

    return true;
}

// Using a ranged-for loop:
bool is_numeric(string s) {
```

```
    for(char c : s)
        if(!is_numeric(c))
            return false;

    return true;
}
```

► What will the following program print as output?

```
int f(int x) {
    x *= 3;
    cout << x << endl;
    return x;
}

int g(int y) {
    cout << y - 1 << endl;
    return f(y) + 1;
}

int h(int x, int y) {
    cout << x + y << endl;
    x = 1;
    return x * y;
}

int main() {
    int x = 3, y = 5;
    cout << f(h(g(x), f(y))) << endl;
    return 0;
}
```

Output:

```
15
2
9
25
45
45
```

2 Vectors and Arrays

► Translate the vector variable declaration

```
vector<string> colors = {"red", "orange", "yellow", "green",
                         "blue", "indigo", "violet" };
```

into an array variable declaration.

```
string colors[] = {"red", "orange", "yellow", "green",
                         "blue", "indigo", "violet" };
```

► Given a vector `v`:

```
vector<int> v;
```

Draw the contents of the vector that will result after the following code is executed:

```
v.resize(5,10);
v.pop_back();
v.insert(v.begin() + 2, 13);
v.push_back(-1);
v.erase(v.begin() + 0);
v.push_back(-4);
```

Index	Tabular
0	10
1	13
2	10
3	10
4	-1
5	-4

► Write a function that will read in `float`s from the user until they press Ctrl-D and then return a vector containing every value entered.

```
vector<float> read_floats() {
    vector<float> vf;
    float x;
    while(cin >> x)
        vf.push_back(x);

    return vf;
}
```

► Write a function which takes a `vector<int>` parameter and which returns `true` if the vector contains any odd numbers, and false otherwise.

```
bool contains_odd(vector<int> v) {
    for(int x : v)
        if(x % 2 == 1)
            return true; // Found an odd number

    return false; // No odds found
}
```

► Write a function which takes a `int n` parameter and which returns a vector containing the integers from 1 to n. E.g., if $n = 4$ then the vector returned should contain {1,2,3,4}. If the parameter is 0 or negative the returned vector should be empty.

```
vector<int> count_up(int n) {
    vector<int> out;
    for(int i = 1; i < n; ++i)
        out.push_back(i);

    return out;
}
```

► Suppose vectors did not have a `.size()` operation, only a `.empty()` operation (returns true if the vector is empty). Could you still write a function which determined the size (number of elements) in the vector? Write a substitute for the `.size()` operation:

```
int size(vector<int> v) {
    int s = 0;
    while(!v.empty()) {
        ++s;
        v.pop_back();
    }
    return s;
}
```

► What are the restrictions that arrays have, compared to vectors?

Arrays cannot change size, cannot be empty, and the size must be a constant (not a variable or an expression).

3 References, Pointers, and Dynamic Memory

► What will the values of the variables a, b, c be after the following code executes?

```
int a = 5, b = 6, c = 7;
int& d = b;
int& e = a;
a += b + d;
b *= c - e;
c -= a + b - d - e;
d *= 2;
e = a * b + c * d - e;
```

$a = -2897, b = -120, c = 7$

($d = -120$ as it's a reference to b, and $e = -2897$ as its a reference to a.)

- What are the differences between references and pointers?

References cannot change during their scope, and must refer to something. Pointers can change what they point to, and can be set to point to nothing at all (`nullptr`).

- What is wrong with the following code fragment:

```
int* p = nullptr;
{
    int x = 12;
    p = &x;
}
*p = 13;
```

p is set to point to x, but then x goes out of scope. So in the last line, `*p` points to something which no longer exists.

- Use reference parameters to write a function `clamp`:

```
void clamp(int& x, int low, int high);
```

The effect should be to constrain the value of x to be in the range $[\texttt{low}, \texttt{high}]$. If $x <$ `low` then set x to `low`; if $x >$ `high` then set x to `high`, otherwise leave x unchanged.

```
void clamp(int& x, int low, int high)
    if(x > high)
        x = high;
    else if(x < low)
        x = low;

    // Or in one line:
```

```
    x = x < low  ? low :
        x > high ? high :
        x ;
```

► What is the *type* of the following variables?

```
int             x = 1;     // int
int&            y = x;     // ref. to int
int*            z = &y;    // pointer to int
int*&           a = z;     // ref. to pointer to int
int**           b = &a;    // pointer to pointer to int
vector<int*>    vp;        // vector of pointers to ints
vector<int*>&   vr = vp;   // ref. to vector of pointers to ints
vector<int*>*   vpp = &vp; // pointer to vector of poitners to ints

// Pointer to pointer to vector of pointers to ints
vector<int*>** vppp = &vpp;
```

► What will be the final values of the variables a, b, c after the following code fragment is executed:

```
int a = 1, b = 2, c = 4;
int* p = &c;
int* q = &b;
int* r = &a;
*p = b;
*r = a;
p = r;
r = q;
q = &a;
*p *= *q;
*q += *r + a;
*r -= *p;
```

$a = 4, b = -2, c = 2$

► What is the difference between `delete` and `delete[]`, and when is each used?

`delete[]` is used to delete dynamically allocated arrays, created via `new T[]` (where T is a type). `delete` is used for deleting single values, dynamically allocated.

► Write a function

```
int* read_ints(int n);
```

which reads in exactly n integers from the user and then returns a pointer to a *dynamically allocated* array containing the values entered.

```
int* read_ints(int n) {
    int* data = new int[n];
    for(int i = 0; i < n; ++i) {
        int x;
        cin >> x;
        data[i] = x;
    }

    return data;
}
```

- Explain the difference between the *scope* of a variable and the *lifetime* of a value.

Scope is what part of a program a *name* can be used; *lifetime* is how long, when the program is running, a value is alive. Some lifetimes are linked to scopes (statically-allocated values), but dynamically allocated values are not linked to any scope.