Practice test for midterm 4 solutions

May 15, 2019

1 Recursion

- Here is a recursive function:

```
void f(string s) {
    if(s.empty())
        return;
    else {
        cout << s.back();
        s.pop_back();
        f(s);
    }
}
```

  - Label the parameter of recursion, the base case, and the recursive case.

    s is the parameter of recursion, `if(s.empty()) return;` is the base case, while `else...` is the recursive case.

  - Draw the recursion tree for the function call `f("Hello")`.

```
f("Hello")      prints 'o'
   |
f("Hell")       prints 'l'
   |
f("Hel")        prints 'l'
   |
f("He")         prints 'e'
   |
f("H")          prints 'H'
   |
f("")           returns
```

- What does this function do?

  Prints the string in reverse.

► Here is a recursive function that is intended to count the number of `0`'s in a vector, but some parts are missing:

```
int count0s(vector<int> vs) {
    if(vs.empty())
        return 0;
    else if(vs.back() == 0) {
        vs.pop_back();
        return 1 + count0s(vs);
    }
    else {
        vs.pop_back();
        return count0s(vs);
    }
}
```

Fill in the missing portions to make the function work correctly.

► In class we looked at a pair of *mutually recursive functions* `is_even` and `is_odd`:

```
bool is_even(int x) {
    if(x == 0)
        return true;
    else if(x == 1)
        return false;
    else
        return is_odd(x-1);
}
```

```
bool is_odd(int x) {
    if(x == 0)
        return false;
    else if(x == 1)
        return true;
    else
        return is_even(x-1);
}
```

Using a similar technique, write *three* mutually recursive functions which should test whether a number is divisible by 3:

```
// Divides with remainder 0
bool div3_0(int x) {
    if(x == 0)
        return true;
    else if(x == 1 || x == 2)
        return false;
    else
        return div3_2(x-1);
}
```

```
// Divides with remainder 1
bool div3_1(int x) {
    if(x == 1)
        return true;
    else if(x == 0 || x == 2)
        return false;
    else
        return div3_0(x-1);
}
```

```
// Divides with remainder 2
bool div3_2(int x) {
    if(x == 2)
        return true;
    else if(x == 0 || x == 1)
        return false;
    else
        return div3_1(x-1);
}
```

► Write a recursive function `length` which determines the length of a string, *without* using `.length()`:

```
int length(string s) {
    if(s.empty())
        return 0;
    else
        return 1 + length(s.substr(1));
```

You could also use `s.pop_back()`.

► Write a recursive function `is_palindrome` which takes a string and returns true if it is a palindrome (the same backwards and forwards):

```
bool is_palindrome(string s) {
    if(s.length() <= 1)
        return true;
    else
        return s.front() == s.back() &&
               is_palindrome(s.substr(1, s.length() - 2));
}
```

2 Inheritance and polymorphism

Several of the problems in this section will refer to the following collection of classes:

```
class item {
  public:
    float weight;
    string name;
};

class weapon : public item {
  public:
    float damage;
    int skill;
};

class sword : public weapon {
  public:
    float length;
};
```

```
class armor : public item {
  public:
    float av;
    int skill;
};

class potion : public item {
  public:
    float amount;
    int attr; // 0 = health, etc
};
```

- Assuming we have the following variables:

```
weapon cudgel;
sword  scimitar;
armor  shield;
potion poison;
```

which of the following data members are valid, and which will cause an error?

- `cudgel.name` OK
- `scimitar.skill` OK
- `poison.weight` OK
- `poison.skill` Error
- `shield.skill` OK
- `cudgel.av` Error
- `shield.name` OK
- `scimitar.weight` OK

- Suppose we add the variables

```
weapon  w1 = scimitar;
weapon& w2 = scimitar;
```

What is the difference between these two? What will change if we execute the assignments:

```
w1.weight = 10;
w2.weight = 20;
```

The first one is *a* slice; *a* copy of the scimitar but with all the `sword` specific details removed. The second one is *a* reference to the scimitar, viewed through the lens of `weapon`. So doing `w2.weight = 20` will actually change the scimitar's weight, while doing so on `w1` will have no effect on the original (because its *a* copy).

- Suppose we add *a virtual method* `.use()` to these classes:

```
class item {
  public:
    ...
    virtual bool use();
};

bool item::use() {
    return false;
}

bool weapon::use() {
    cout << "Attack for "
         << damage << endl;
    return false;
}

bool potion::use() {
    cout << "You drink the potion"
         << endl;
    return true;
}
```

(The idea is that `use` returns true if the item is "used up" by being used.)

What will be printed by each of the following calls to `use`:

```
cudgel.use(); // Attack for ... damage


scimitar.use(); // Attack for ... damage


shield.use(); // prints nothing


w1.use();  // Attack for ... damage
```

```
w2.use();  // Attack for ... damage
```

► Write a set of classes with inheritance intended to model things you might put in a salad. A base class `salad_ingredient` is provided for you.

```
class salad_ingredient { };

class lettuce : public salad_ingredient { };

class tomato : public salad_ingredient { };

class cheese : public salad_ingredient { };
```

► Suppose we want to now create a class `salad` that can contain any number of different ingredients. Will the following class definition work? If not, why not?

```
class salad {
  public:
    vector<salad_ingredient> ingredients;
};
```

Because the vector is not storing pointers or references, it will slice the ingredients. (And vectors can't store references, so it has to be a vector of pointers.)

► Explain the difference between IS-A and HAS-A relationships, and give examples of classes with each kind of relationship.

Lettuce is a salad ingredient, but a salad has salad ingredients. IS-A is for inheritance (subtyping), while HAS-A is for containment (data members).

3 Advanced topics

Note that this section won't be on the test. I left it here in case you want to test your knowledge.

► Here is a class for colors:

```
class color {
  public:
    string name;
    float r,g,b;
};
```

Overload the << insertion operator so that we can print colors to `cout` naturally.

```
ostream& operator<< (ostream& out, color c) {
    out << c.name; // Print r,g,b, if you want

    return out;
}
```

► Overload the equality == and inequality != operators on `color` (either as normal functions or member functions inside the class) so that we can compare colors.

```
bool operator== (color a, color b) {
    // Check name equality too, if you want.
    return a.r == b.r && a.g == b.g && a.b == b.b;
}

bool operator!= (color a, color b) {
    return !(a == b);
}
```

► Write a template function `is_sorted` which takes a vector of *any* type of elements and returns true if they are sorted (if each element is $\leq$ the following one.)

```
template<typename T>
bool is_sorted(vector<T> v) {
    for(int i = 0; i < v.size() - 1; ++i)
        if(v.at(i) > v.at(i+1))
            return false;

    return true;
}
```

► What modification(s) would you need to make to the function from the previous function to allow it to work on vectors or strings? Write the modified function.

```
// This will work on any container that supports .size() and .at(),
// and whose elements support >
template<typename V>
bool is_sorted(V v) {
    for(int i = 0; i < v.size() - 1; ++i)
        if(v.at(i) > v.at(i+1))
            return false;

    return true;
}
```

- Using the higher-order functions we looked at in class:

```
template<typename T, typename R>
R reduce(vector<T> vs, R start, function<R(R,T)> fn)

template<typename T, typename R>
vector<R> map(vector<T> vs, function<R(T)> fn)

template<typename T>
vector<T> filter(vector<T> vs, function<bool(T)> pred)
```

and a vector v:

```
vector<float> v = ...;
```

write loop-free code which will perform the following operations on v:

1) Square every element
2) Remove any elements that are > 100
3) Sum the remaining elements

(You can write named functions for the function parameters, or you can use the anonymous function syntax we showed in class.)

```
// Using free functions:
float square(float x) { return x*x; }
bool le_100(float x) { return x <= 100; }
float add(float a, float b) { return a + b; }

reduce(filter(map(v, square), le_100), 0, add)

// Using anonymous functions
reduce(filter(map(v, [](float x) { return x*x; })
              [](float x) { return x <= 100;})
       0,
       [](float a, float b) { return a + b; })
```