Practice test for midterm 4

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.
- Draw the recursion tree for the function call `f("Hello")`.
- What does this function do?

▶ 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
    else if(vs.back() == 0)
        return
    else
        return
}
```

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) {
```

```
// Divides with remainder 1
bool div3_1(int x) {
```

```
// Divides with remainder 2
bool div3_2(int x) {
```

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

```
int length(string s) {
```

► 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) {
```

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

► 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;
```

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

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

```
scimitar.use();
```

```
shield.use();
```

```
w1.use();
```

```
w2.use();
```

▶ 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 { };
```

▶ 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;
};
```

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

3 Advanced topics

▶ 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.

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

▶ 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.)

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

▶ Using the functional programming building-blocks 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.)