Initializers for Instances

Let's understand the purpose of initializers in constructors.

Member initializer lists

An initializer list is used to initialize the members in the constructor of a class. The list is added **before** the body of the constructor.

Rather than creating the variables and then assigning values to them within the constructor's body, an initializer list initializes the variables with their particular values. The list also makes the constructor's implementation simpler and more readable.

One may wonder why this is useful. Well, consider that our class has a `const` attribute. Naturally, it would have to be initialized. Assigning it values in the constructor's body will cause an error.

This is where an initializer would solve the problem.

The list is appended after the parameters of the constructor and a `:`.

```
class Account{
public:
  Account (double b): balance(b), minBalance(25.0){}
private:
  double balance;
  const double minBalance; ....
};
```

Rules

1. As discussed, non-static attributes that are declared `const` or as a reference must be initialized in the initializer list.

2. The sequence of initialization corresponds to the sequence in which the attributes were declared.

3. `static` attributes cannot be initialized in the initializer of the constructor.

Example

```
#include <iostream>
#include <string>

class Account{
public:
  Account(double b, std::string& rev):balance(b), reviser(rev), minBalance(25.0){}
  /*
  Account(double b, std::string& rev){
    balance = b;
    reviser = rev;
    minBalance = 25.0;
  }*/

private:
    double balance;
    const double minBalance;
    std::string& reviser;
};

int main(){

  std::string reviser{"grimm"};
  Account account(100.0, reviser);

}
```

- The initializer list in line 6 allows us to omit the longer constructor from lines 8 to 12.

- The commented constructor won't work since `minBalance` is a `const` variable.

Initializing class attributes

Class members can be directly initialized outside the constructor.

```
class MyClass{
    const static int oldX = 5; // C++98
    int newX = 5; // C++11
    vector<int> myVec{1, 2, 3, 4, 5}; // C++11
};
```

If class attributes are initialized in both the class body and the initializer list of the constructor, the latter has a higher priority.

```
struct MyClass{
  MyClass(int n): newX(n){} int newX = 5;
};
```

Example

```
#include <iostream>
#include <string>
#include <vector>

struct ClassMemberInitializer{

    ClassMemberInitializer() = default;
    ClassMemberInitializer(int num):x(num){};

    //valid with C++98
    const static int oldX=5;

    // valid with C++11
    int x = 5; //class member initializer

    // valid with C++11
    std::string s = "Hello C++11";

    // valid with C++11
    std::vector<int> myVec{1, 2, 3, 4, 5};

};

int main(){

    std::cout << "\n";

    // class member initialization
    ClassMemberInitializer cMI;
    std::cout << "cMI.oldX " << cMI.oldX << "\n";
    std::cout << "cMI.x " << cMI.x << "\n";
    std::cout << "cMI.s " << cMI.s << "\n";
    for (auto vec: cMI.myVec) std::cout << vec << " ";
    std::cout << "\n\n";


    // class member initialization
    // x will be overriden by the constructor value
    ClassMemberInitializer cMI2(10);
    std::cout << "cMI2.oldX " << cMI2.oldX << "\n";
    std::cout << "cMI2.x " << cMI2.x << "\n";
    std::cout << "cMI2.s " << cMI2.s << "\n";
    for (auto vec: cMI2.myVec) std::cout << vec << " ";

    std::cout << "\n\n";

}
```

- This example simply shows how class members can be initialized outside the constructor.
- Initializing `const static` attributes, as in line 11, has been a feature since C++98.
- C++11 adds support for several other types such as `int`, `string`, and `vector`.

Let’s test what we’ve learned about constructors.