Constructor Delegation

In this lesson, we will further build on the idea of an initializer list in a constructor.

WE'LL COVER THE FOLLOWING

So far, we've seen how class members can be initialized in the constructor through an initializer list. However, we are not restricted to just that.

Definition

> **Constructor delegation** occurs when a constructor calls another constructor of the same class in its initializer list.

```
struct Account{
  Account(): Account(0.0){}
  Account (double b): balance(b){} };
```

In the example above, the default constructor calls the parameterized constructor and initializes `balance` to `0.0`.

Rules

- When the constructor call in the initializer list is completed, the object is **created**. This object can then be altered in the calling constructor's body.

- It is important to perform constructor delegation in the initializer list. If it is called in the body, a new class object will be created and we will end up

with **two objects**, which is not the behavior we want.

- Recursively invoking constructors will result in **undefined behavior**.

The aim of delegation is to let one constructor handle initialization. That object can be used or modified by all other constructors. In other words, constructors can delegate the task of object creation to other constructors.

A great advantage of constructor delegation is that the initialization code is localized. Once that code has been tested, the rest of the code can be built upon it robustly. Also, bugs, if there are any, will be localized to the specific constructor(s) instead of being spread around all over the place.

Example

```
#include <cmath>
#include <iostream>

class MyHour{
public:
  MyHour(int hour){
    if (0 <= hour && hour <= 23) myHour = hour;
    else myHour = 0;
    std::cout << "myHour = " << myHour << std::endl;
  }

  MyHour(): MyHour(0){}

  MyHour(double hour):MyHour(static_cast<int>(ceil(hour))) {}

private:
  int myHour;
};

int main(){

  std::cout << std::endl;

  MyHour(10);
  MyHour(100);
  MyHour();
  MyHour(22.45);

  std::cout << std::endl;

}
```

- In this example, the constructor in line 6 is responsible for creating a

`myHour` object, whereas the other constructors simply call it in their initializer lists.

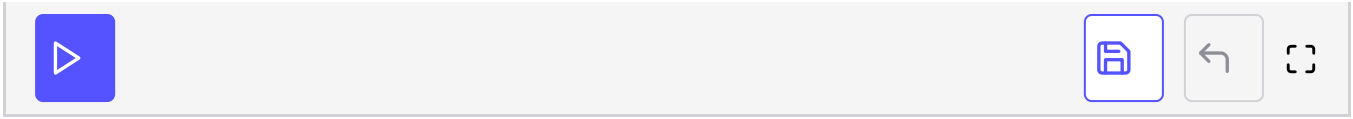
- The constructors in lines 12 and 14 do not require a code in their body since they are simply creating a `myHour` object with the specified integer value. For this, all they need to do is delegate the task to the first constructor.

Recursion

Let's implement a struct in which constructors call themselves recursively. What will happen?

```
struct C {
  C( int ) { }               // 1: non-delegating constructor
  C(): C(42) { }             // 2: delegates to 1
  C( char c ) : C(42.0) { }  // 3: ill-formed due to recursion with 4
  C( double d ) : C('a') { } // 4: ill-formed due to recursion with 3
};

int main(){
  C(1);
  C();
  C('a');
  C(3.5);
}
```



- The problem lies in constructors `3` and `4`. They call each other recursively and end up in a cycle.
- This causes a segmentation fault.
- Take a look at constructor `2`. If it had delegated object creation to constructor `3` or `4` instead of constructor `1`, the recursive loop would have started again.

That ends our discussion on constructors. The next lesson deals with **destructors**.