Object-Oriented Programming

Iuliana Bocicor
iuliana@cs.ubbcluj.ro

Babes-Bolyai University

2016

Overview

Concurrency I

- Several tasks can be executed concurrently.
- Why?
  - To improve throughput (by using several processors for a single computation).
  - To improve responsiveness (by allowing one part of the program to progress while another is waiting for a response).
- The standard library offers concurrency support facilities.

Concurrency II

- *Task* - a computation which could be executed concurrently with other computations.
- *Thread* - a system-level representation of a task in a program.
- A task to be executed concurrently with other tasks is launched by constructing a std::thread (<thread> header) with the task as its argument.

Demo

DEMO

Fibonacci calculator - GUI (*Lecture13_demo_threads*).

Lvaues and rvalues

- An *lvalue* refers to a memory location and allows us to take the address of that memory location via the & operator.
- An *rvalue* is an expression that is not an lvalue.
- Rvalues refer to temporary objects and are destroyed at the end of the expression that contains the value.

```
int a = 2; // a is an lvalue
int b = 3; // b is an lvalue
int c = a *  b; // c is an lvalue, a * b is an rvalue
a * b = 0; // error, a * b is an rvalue, cannot be on the
     left side of an assignment
int d = fct(); // the temporary value returned by fct() (
    a function which returns an int) is an rvalue
```

Remember our dynamic vector? I

- Which object construction/destruction functions are called when the code below is executed?

```
DynamicVector create()
{
    DynamicVector v;
    v.add(1);
    v.add(2);

    return v;
}

int main()
{
    DynamicVector v1;
    v1.add(100);
    v1 = create();
    return 0;
}
```

Remember our dynamic vector? II

1. *v1* is created - memory is allocated.
2. *v* is created - memory is allocated.
3. When the function finishes, *v* is copied in a temporary (memory allocated again) and then *v* is destroyed.
4. The memory from *v* is deallocated. Then new memory is allocated and the resources from the temporary returned by *create()* are copied in *v1*.
5. The temporary returned by *create()* is destroyed (memory is deallocated).

Rvalue references I

- It would be more efficient to destruct *v1*'s initial resources and then make *v1* "steal" the resources held by the temporary.
- *When the righ-hand side is an rvalue, we would like the copy constructor to only "steal" the resources.*
- This is *move semantics*.

Rvalue references II

- Thus, the copy constructor should be overloaded, such that when dealing with an rvalue, this type of behaviour (move instead of copy) should be chosen.
- The parameter of the copy constructor should have a reference type, as it will be modified.
- This parameter will be an **rvalue reference**.

Rvalue references III

- $\Rightarrow$ two overloads of the copy constructor:
  - the conventional copy constructor (which receives a reference to a DynamicVector) - will be called for lvalues;
  - **move constructor**: gets as parameter an rvalue reference - wil be called for rvalues.
- We want the same behaviour for the assignment operator $\Rightarrow$ **move assignment operator**.
- Rvalue references allow a function to branch at compile time (via overload resolution) on the condition "Am I being called on an lvalue or an rvalue?"

Rvalue references IV


```
// copy constructor
DynamicVector(const DynamicVector& v);

// move constructor
DynamicVector(DynamicVector&& v);

// copy assignment operator
DynamicVector& operator=(const DynamicVector& v);

// move assignment operator
DynamicVector& operator=(DynamicVector&& v);
```

- Rule of three $\rightarrow$ Rule of five.

Demo

DEMO

Move semantics - Dynamic vector (*Lecture13_demo_move_semantics*).