Object-Oriented Programming
Singleton Pattern

CSIE Department, NTUT
Woei-Kae Chen
Singleton pattern: Intent

- Ensure a class only has one instance, and provide a global point of access to it.
  - Example:
    - clock,
    - printer spooler
    - file system
    - window manager
    - A/D converter
    - I/O port
Singleton pattern: Motivation

- How do we ensure that a class has only one instance and the instance is easily accessible?
  - global variable
    - makes an object accessible
    - but cannot ensure from multiple objects
  - make the class itself responsible
    - ensure no other instance can be created
    - provide a way to access the instance
Singleton pattern: Structure

**Singleton**
- static Instance()
- SingletonOperation()
- GetSingletonData()
- static uniqueInstance
- singletonData

*Static member function*
- return uniqueInstance

*Static member variable*
Singleton pattern: Applicability

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.
Singleton pattern: Participants

• Singleton
  – Defines an Instance operation that lets clients access its unique instance.
  – May be responsible for creating its own unique instance.
Singleton pattern: Collaborations

- Clients access a Singleton instance solely through Singleton’s Instance operation.
Singleton pattern: Consequences

- Controlled access to sole instance.
- Reduced name space
  - no pollution to the name space of global variables
- Permits refinement of operations and representation
  - may be subclassed
- Permits a variable number of instances
  - only `Instance()` needs to change
- More flexible than class operations
  - C++ static member functions
    - hard to change design to allow more than one instance
    - non-virtual: overriding not allowed
Singleton pattern: Implementation (1)

Ensuring a unique instance.

```cpp
class Singleton {
    public:
        static Singleton* Instance();
        void Operation();
    protected:
        Singleton();
    private:
        static Singleton* _instance;
}

Singleton* Singleton::_instance = 0;
Singleton* Singleton::Instance() {
    if (_instance == 0) {
        _instance = new Singleton;
    }
    return _instance;
}

// Client
// Singleton::Instance()->Operation();
```

- protected constructor
- * _instance

Who: delete _instance? (memory leak)
Singleton pattern: Implementation (2)

- C++ global or static object
  - can’t guarantee single instance
  - automatic initialization may not work (a single might require values that are computed after initialization
  - order of constructors of global objects are not defined (no dependencies can exist between singletons)
  - created whether they are used or not
Singleton pattern: Implementation (3)

Subclassing the Singleton class.

class Singleton {
public:
    static void Register(char* name, Singleton*);
    static Singleton* Instance();
protected:
    static Singleton* Lookup(const char* name);
private:
    static Singleton* _instance;
    static List<NameSingletonPair>* _registry;
};
Singleton pattern: Implementation (4)

```cpp
Singleton* Singleton::Instance() {
    if(_instance==0) {
        const char* singletonName = getenv("SINGLETON");
        _instance = Lookup(singletonName);
    }
    return _instance;
}

MySingleton::MySingleton() {
    Singleton::Register("MySingleton", this);
}

static MySingleton theSingleton;
```
Singleton pattern: Related patterns

- Many patterns can be implemented using singleton pattern
  - Abstract factory
  - Builder
  - Prototype