# Design Patterns

# What

- Creational, Behavioral, Structural
- Creational:
    - Manage the instantiation process.
    - Abstraction of how objects are created.
- Two types of Creational Patterns
    1. Class-based creational patterns
       Use inheritance to vary the class that is instantiated
    2. Object–based creational patterns
       Delegates instantiation to another object

# Singleton

- The singleton ensures only one instance exists

```
class Singleton
{
private:
        static Singleton* pSingInst = nullptr ;
protected:
        Singleton() {}
public:
        static Singleton* getInstance(){
                if (pSingInst ==nullptr)
                        pSingInst = new Singleton() ;
                        return pSingInst;
                }
};

Singleton* Singleton ::g_pS = nullptr;
int main() {
Singleton* g_pS = Singleton::GetInstance();
}
```

Lazy
Initialization

# Singleton – Initialization of global variables

- It replaces global variable and control the order of initialization
- The following program calls an unconstructed object
- Output:

```
Clock: Adding to global Clock
Clock: member function add 0x10f739034
Clock: Constructor 0x10f739034
```

- But wait, isn't it the same Class?

```cpp
#include "stdio.h"
struct Clock;
extern Clock globalClock;
struct Clock {
    Clock() {printf("Clock: Constructor %p\n",this);}
    void add() {  printf("Clock: member function add %p\n",this);}
    static int addToGlobal() {
        printf("Clock: Adding to global Clock\n");
        globalClock.add();
        return 0; }
};
int dummy = Clock::addToGlobal();
Clock globalClock;
int main() {
    return 0;
}
```

# Singleton – Initialization of global variables

- With two different classes
- Still, the following program calls an unconstructed object
- Output (note Clock2,Clock1):

```
Clock2: Adding to global Clock
Clock1: member function add 0x10ee49034
Clock1: Constructor 0x10ee49034
```

```
#include "stdio.h"
#include "temp.h" // here we define Clock1
extern Clock1 globalClock;
struct Clock2 {
  int clock_time;
  Clock2() {
    printf("Clock2: Constructor %p\n",this);
  }
  void add() {
    printf("Clock2: member function add %p\n",this);
  }
  static int addToGlobal() {
    printf("Clock2: Adding to global Clock\n");
    globalClock.add();
    return 0;
  }
};
int dummy = Clock2::addToGlobal();
Clock1 globalClock;
int main() {
  return 0;
}
```

# Singleton – Initialization of global variables

- Why do we have access to uninitialized objects??
- In C++ (C) we can use global objects from other files without knowing if they have been already constructed or not
  - Once we use the keyword “extern” we use forward declaration
- The order of initialization is within a file (“translation unit”) but not between different files with global variables
- By using singleton we can give access from various files while keeping the objects initialized

# Singleton – Global Vars Init

```
class Clock2 {
   static Clock2 * myClock2;
private:
   Clock2(){printf("Clock2 constructor\n");};
public:
   static Clock2* getInstance() {
      if (myClock2 == nullptr) {
         myClock2 = new Clock2();
      }
      return myClock2;
   }
   void add() {
      printf("Clock2: member function add %p\n",this);
   }
};
```

# Singleton – Global Vars Init

```
class Clock {
static Clock * myClock;
private:
   Clock(){printf("Clock constructor\n");};
public:
   static Clock* getInstance() {
      if (myClock == nullptr) {
         myClock = new Clock();
      }
      return myClock;
   }
   void add() {
      printf("Clock2: member function add %p\n",this);
   }
    int addToGlobal() {
      printf("Clock2: Adding to global Clock\n");
      Clock2::getInstance()->add();
      return 0;
   }
};
```

```
Clock * Clock::myClock;
Clock2 * Clock2::myClock2;

int main() {
   Clock::getInstance()->addToGlobal();
}
```

**Output:**

Clock constructor
Clock2: Adding to global Clock
Clock2 constructor
Clock2: member function add 0x7fca8f400690
Note: constructor before add!

# Singleton – Template version

## Given an existing class, how to convert it to a singleton?

```
// file.h
template <class T>
class Singleton : public T
          {
          public:
                    static Singleton* GetInstance()
                    {
                              if (pSingObject==nullptr){
                                        pSingObject = new
Singleton ;
                              }
                              return pSingObject ;
                    }
                    ~Singleton() { delete pSingObject ; }
          private:
                    Singleton() { } ;
                    static Singleton*    pSingObject ;
          };
```

```
// file.cpp
    template <class T>
    Singleton<T>* Singleton<T>::pSingObject = NULL ;

// MyClass.cpp
    class MyMainClass {
          void myActionFunction() ;
    } ;
// Main.cpp
    Singleton<MyMainClass>::GetInstance()->myActionFunc()
    ;
```

# Singleton Destruction

- The wrong way

```
~Singleton() {
  delete pSingInst;
  pSingInst = nullptr;
}
```

- The right way

```
static void ResetInstance() {
  delete pSingInst;
  pSingInst = nullptr;
}
```

# Factory

- We have a maze game
- And we would like to create various types of mazes
  - Regular one, enchanted, etc..

```
class MazeGame
{
public:
   Maze* CreateMaze() {
         Maze* maze  = new Maze() ;
      Room* room1 = new Room(1) ;
      Room* room2 = new Room(2) ;
      Door* door  = new Door(room1,room2) ;
         maze->AddRoom(room1) ;
      maze->AddRoom(room2) ;
             room1->SetSide(North, new
Wall()) ;
      room1->SetSide(East , door) ;
      room1->SetSide(South, new Wall()) ;
      room1->SetSide(West , new Wall()) ;
         room2->SetSide(North, new Wall())
;
      room2->SetSide(East , new Wall()) ;
      room2->SetSide(South, new Wall()) ;
      room2->SetSide(West , door) ;
         return maze ;
       }
}
```

Credit: Moshe Fresco

# Factory

- Using virtual functions to create the components of the class

```
class MazeGame
{
public:
        virtual Maze* MakeMaze() const    { return new Maze() ;
}
        virtual Room* MakeRoom(int n) { return new Room(n) ; }
        virtual Wall* MakeWall() { return new Wall() ; }
        virtual Door* MakeDoor(Room* r1, Room* r2)
                { return new Door(r1,r2) ; }
        Maze* CreateMaze() {
                Maze* maze = MakeMaze() ;
        Room* room1 = MakeRoom(1) ;
        Room* room2 = MakeRoom(2) ;
        Door* door  = MakeDoor(room1,room2) ;
                .........
                .........
                return maze ;
        }
} ;
```

# Factory

```
class BombedWall: public Wall {
       // …
} ;

class RoomWithABomb: public Room {
public:
       RoomWithABomb(int n) : Room(n) { }
} ;

class BombedMazeGame: public MazeGame {
public:
       BombedMazeGame();
       virtual Wall* MakeWall()
              { return new BombedWall() ; }
       virtual Room* MakeRoom(int n)
              { return new RoomWithABomb(n) ; }
} ;
```

# Abstract Factory

- Using an object as a parameter to create components

```
class MazeFactory {
public:
        Maze* MakeMaze() { return new Maze() ; }
        Room* MakeRoom(int n) { return new Room(n) ; }
        Wall* MakeWall()    { return new Wall() ; }
        Door* MakeDoor(Room r1, Room r2)
               { return new Door(r1,r2) ; }
} ;
class MazeGame {
public:
        Maze* CreateMaze(MazeFactory* factory) {
               Maze* maze  = factory->newMaze() ;
        Room* room1 = factory->newRoom(1) ;
        Room* room2 = factory->newRoom(2) ;
        Door* door  = factory->newDoor(room1,room2) ;
               .........
               return maze ;
        }
} ;
```

# Abstract Factory

```
class BombedWall: public Wall {
       // …
} ;

class RoomWithABomb: public Room {
public:
       RoomWithABomb(int n) : Room(n) { }
} ;

class BombedMazeFactory: public MazeFactory
{
public:
       BombedMazeGame();
       virtual Wall* MakeWall()
              { return new BombedWall() ; }
       virtual Room* MakeRoom(int n)
              { return new RoomWithABomb(n)
; }
} ;
```

# Abstract vs. Non-Abstract

- Compile time vs. Run time
- Subclass vs. Objects

# Factory-Singleton

- Often, it is best for Factory to be a Singleton

```
class MazeFactory {
       protected: MazeFactory() { }
       private: static MazeFactory* inst = null ;
       public: static MazeFactory* getInst()
              { if (inst==null) inst = new MazeFactory() ;return inst ; }
       Maze* makeMaze()
              { return new Maze() ; }
       Room* makeRoom(int n)
              { return new Room(n) ; }
       Wall* makeWall()
              { return new Wall() ; }
       Door* makeDoor(Room r1, Room r2)
              { return new Door(r1,r2) ; }
} ;
```

# Factory-Singleton

```
class MazeGame
{
public:
  Maze* createMaze() {
        Maze maze*  = MazeFactory.getInst()->MakeMaze() ;
        Room room1* = MazeFactory.getInst()->MakeRoom(1) ;
        Room room2* = MazeFactory.getInst()->MakeRoom(2) ;
        Door door*  =
          MazeFactory.getInst()->MakeDoor(room1,room2) ;

        maze->AddRoom(room1) ;
        maze->AddRoom(room2) ;
        ………
        return maze ;
  }
}
```

# Runtime-based Factory

- Suppose we would like to decide in runtime what type of factory to use?
- For example, let the user decide which type of Maze it would like to play
- We can use configuration file

# Runtime-based Factory

```
MazeFactory* MazeFactory::getInst()
        { if (inst==0) {
              const char* style = readConfigFile("MAZESTYLE") ;
              if (strcmp(style,”Complex”))
                     inst = new ComplexMazeFactory() ;
              else if (strcmp(style,”Enchanted”))
                     inst = new EnchantedMazeFactory() ;
              else
                     inst = new MazeFactory() ;
          }
          return inst ;
}
```

# Builder

- We would like to build a class with various options or stages
- Sometime we know all the defaults and sometimes not

```
public class Computer {
        private:
                String HardDiskType;
                int HardDiskSize
                String Cpu;
                int CpuClock;
                int RAMSize;
                String RAMtype;
                String KeyboardType;
                String MouseType;
        public:
                Computer(String Cpu, int CpuClock){}
                Computer(String Cpu, int CpuClock , int HardDiskSize) {}
                Computer(String Cpu, int CpuClock , int HardDiskSize, String HardDiskType) {}
                Computer(String Cpu, int CpuClock, int RAMSize) {}
};
```

# Builder

- Two issues
  - We would like to use only a subgroup of the parameters
  - We would like to define several combinations that are mandatory

# Builder

- The following constructor options can solve it
    - CreateComputerCPUsizeAndHardDiskSizeAndRamSize
    - CreateComputerCpuTypeRamTypeHardDiskSize
- And so on..
- Complicated to code and hard to use

# Builder

- The key idea is to create flexibility in the construction options
- Constructor is private, only builder can access it
- How to implement it??

```
public class Computer {
        private:
                String HardDiskType;
                int HardDiskSize
                String Cpu;
                int CpuClock;
                int RAMSize;
                String RAMtype;
                String KeyboardType;
                String MouseType;
        private Computer(Builder builder) {
                this.CpuType = builder. CpuType;
                this.CpuClock = builder. CpuClock;
                this.HardDiskType = builder. HardDiskType;
                this.HardDiskSize = builder. HardDiskSize;
        And so on..
        }
};
```

# Builder

- Builder is class within Computer, usually static
- It has all the fields of Computer

```
public static class Computer::Builder { // Can be defined inside Computer
        private String CpuType;
        private int  CpuClock;
        // more members here.. ramType,ramSize,

        public Builder cpuType(String CpuType) {
                this.CpuType = CpuType;
                return this;
        }
        public Builder cpuClock(int CpuClock) {
                this.CpuClock = CpuClock;
                return this;
        }
        // more member functions to set everything:  ramSize,ramType

        // THIS IS THE ENTRY POINT
        public Computer build() {
                return new Computer(this);
        }
} ;
```

# Builder

- How to use it?

```
Computer myComputer =
new Builder().cpuType("i-9870).ramSize(80).build();
```

# Builder

- How to make mandatory fields during construction?

```
public Builder(String cpuType, int HardDiskSize)
{
        this.cpuType = cpuType;
        this.HardDiskSize = HardDiskSize;
// more fields goes here
}
```