23/10/2015

Programming in the future tense,

or Accept that things will change

Michael Brehler
Medical and Biological Informatics (MBI)

dkfz.
GERMAN
CANCER RESEARCH CENTER
IN THE HELMHOLTZ ASSOCIATION

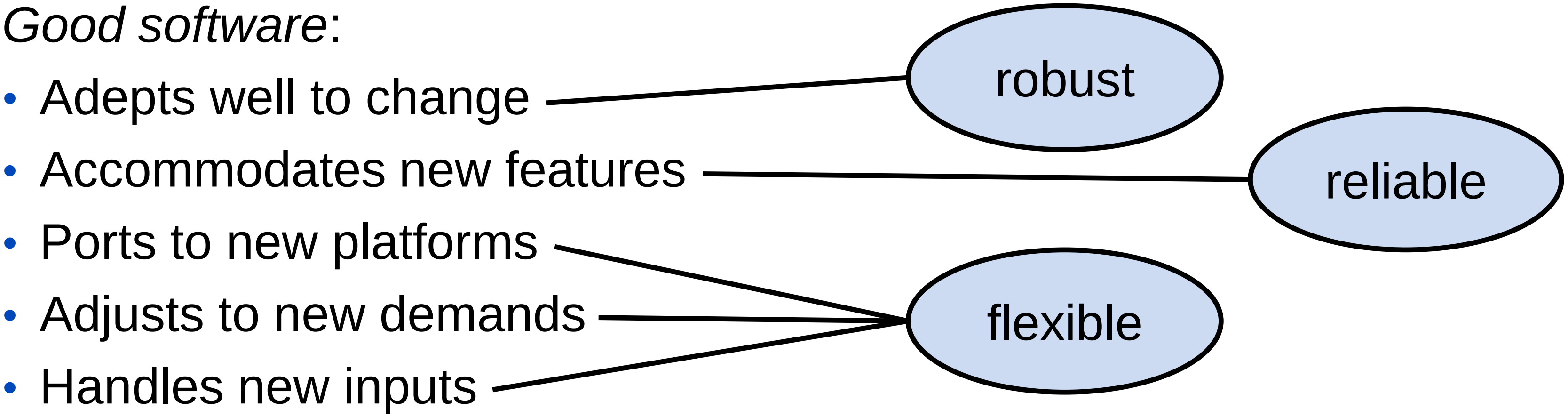
Programming in the future tense

Object-oriented programming

→ Support for incremental changes

→ add new functionality and new properties

Good software:



- Adepts well to change
- Accommodates new features
- Ports to new platforms
- Adjusts to new demands
- Handles new inputs

robust

reliable

flexible

Good software does not come about by accident,

Good software is written by **Programming in the future tense!**

Why?

What could happen…

- New classes are added to the hierarchies
- New overloading will occur
- Derived classes may be tomorrow‘s base classes
- Functions are called in new context

Why?

What ~~could~~ will happen!

- New classes are added to the hierarchies
- New overloading will occur
- Derived classes may be tomorrow‘s base classes
- Functions are called in new context

Additional problem:

“It is to remember that the programmers who modify code [fix bugs] are typically ***NOT*** *the code‘s original developers!”*

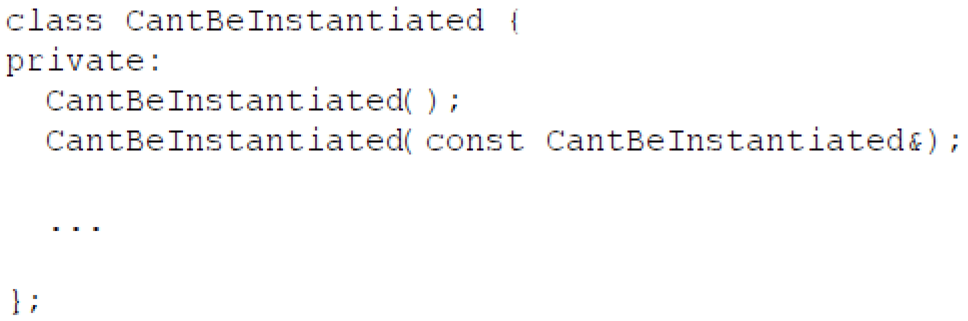
- Scott Meyers, More Effective C++, Addison-Wesley, 2011

HOW TO…

- One way to do this is to express design constraints in C++ (in addition to comments and documentation):

- A class is designed to never have derived classes

→ use C++ to prevent derivation



```
class CantBeInstantiated {
private:
   CantBeInstantiated( );
   CantBeInstantiated( const CantBeInstantiated&);

   ...

};
```

or even better, use the **final** keyword of C++ 11

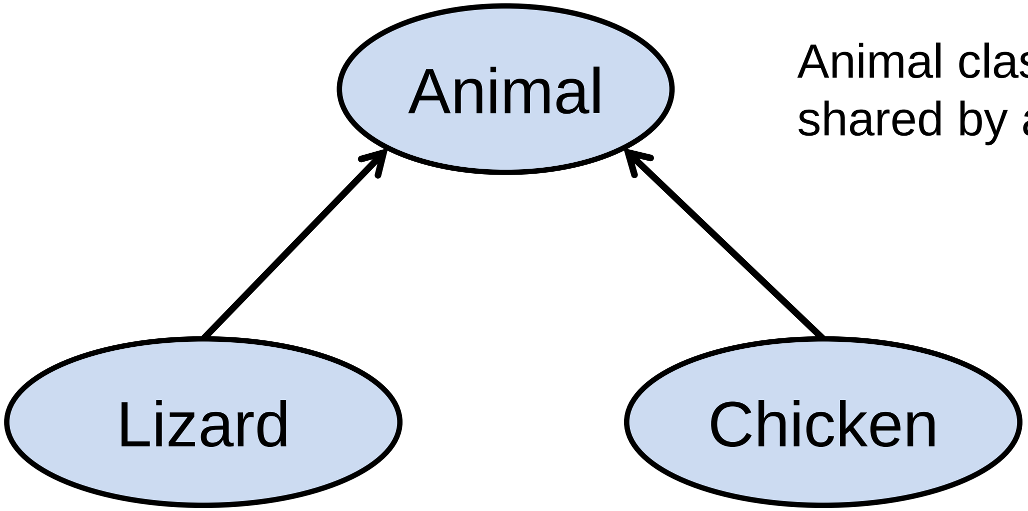
HOW TO…

- If copy and assignment make no sense for a class

→ prevent those operations by declaring the copy constructor and assignment operator private

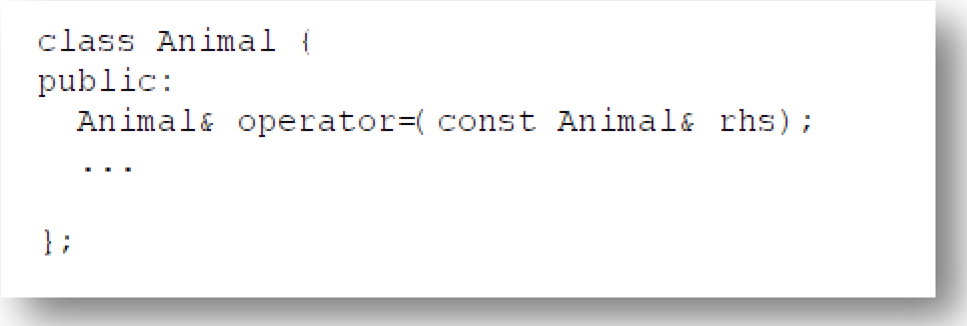
→ Prevent partial assignments

Chicken / Lizard example

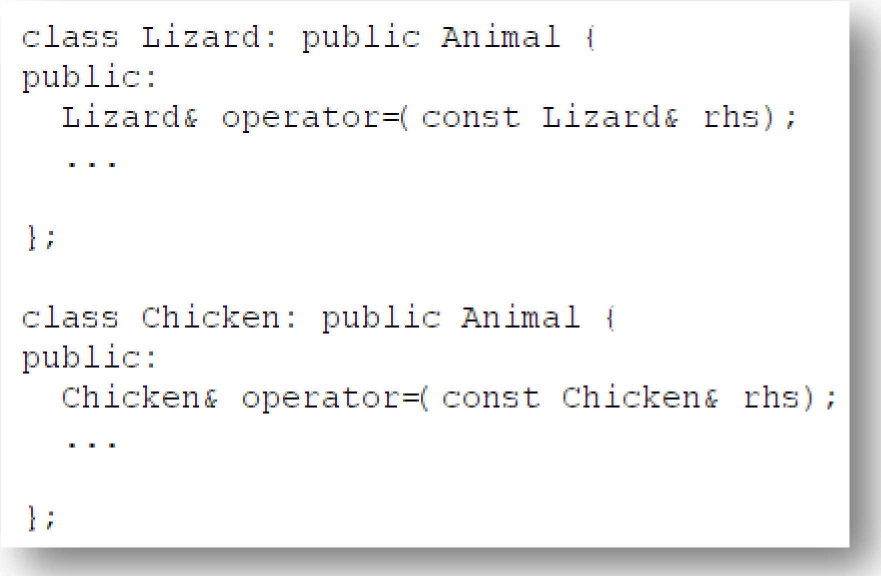


Animal class embodies all features shared by all creatures

Specialize Animal in ways appropriate for Chickens and Lizards

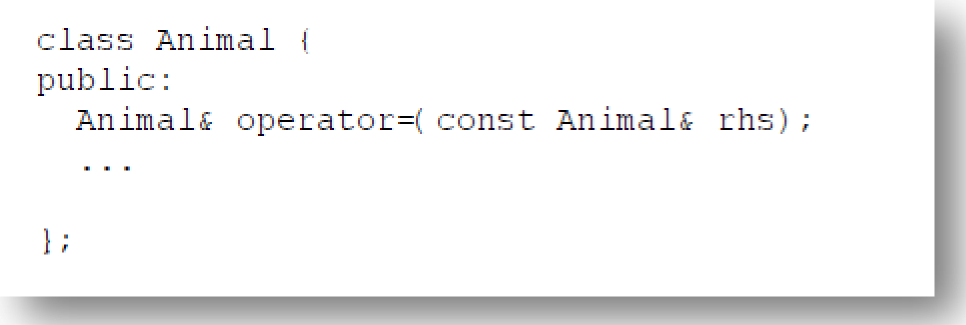


```
class Animal {
public:
  Animal& operator=( const Animal& rhs);
  ...

};
```

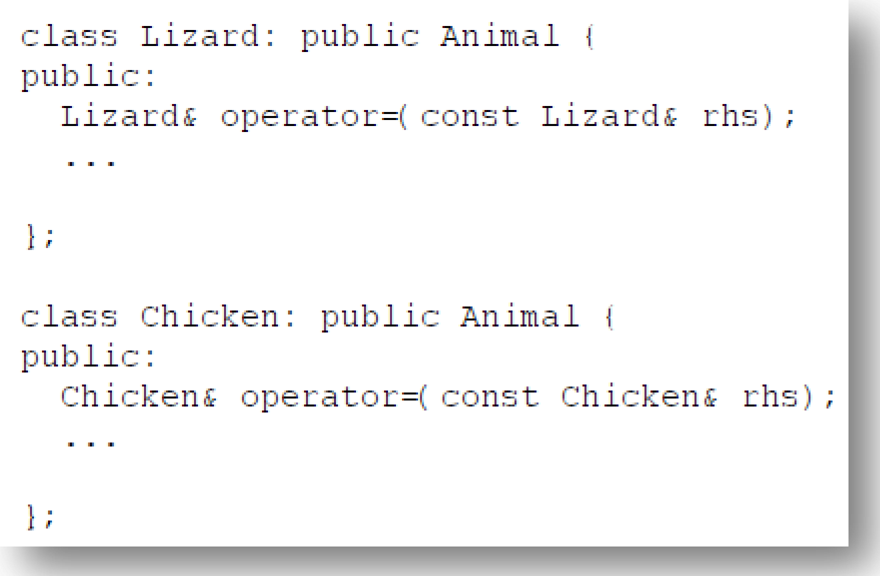


```
class Lizard: public Animal {
public:
  Lizard& operator=( const Lizard& rhs);
  ...

};

class Chicken: public Animal {
public:
  Chicken& operator=( const Chicken& rhs);
  ...

};
```

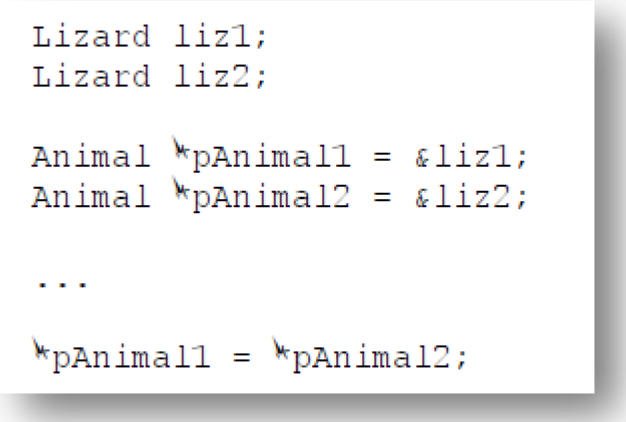
Chicken / Lizard example



```
class Animal {
public:
  Animal& operator=( const Animal& rhs);
  ...

};
```



```
class Lizard: public Animal {
public:
  Lizard& operator=( const Lizard& rhs);
  ...

};

class Chicken: public Animal {
public:
  Chicken& operator=( const Chicken& rhs);
  ...

};
```



```
Lizard liz1;
Lizard liz2;

Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &liz2;

...

*pAnimal1 = *pAnimal2;
```

Only the Animal part liz1 will be modified!
→ **Partial assignment**

HOW TO…

- Avoid „demand-paged“ virtual functions (only make functions virtual when somebody comes along and demands it)

→ Make it virtual if it makes sense

→ If it does not make sense that's ok but don‘t change it later just because it would be convenient for someone

- Handle assignments and copy construction in every class

→ Even if „nobody ever does those things“

Recognize that anything somebody CAN do, they WILL do.

Examples

Most frequent (simple) MITK examples:

- Assigning objects to themselves
- Use objects before giving them values
- Give objects values and never use them
- Give objects huge, tiny or null values

A friendly reminder:

If it will compile, somebody will do it.

Summary

- Present-tense thinking is ok
  - You can’t wait for the latest language features
  - It has to run on the current hardware
  - It has to offer acceptable performance NOW

- Provide complete classes, even if some parts aren‘t currently used.
- Design your interfaces to facilitate common operations and prevent common errors → Make the classes hard to use incorrectly!
- If there is no great penalty for generalizing your code, generalize it.

Be a renegade and program in future tense!