Object-Oriented Programming for Scientific Computing

Namespaces and Inheritance

Ole Klein

Interdisciplinary Center for Scientific Computing
Heidelberg University
ole.klein@iwr.uni-heidelberg.de

5. Mai 2015

Namespaces

- Namespaces permit classes, functions and global variables to be grouped under one name. This way, the global namespace can be divided into subareas, each of which has its own name.
- A namespace is defined by:

```
namespace Name
{
// classes, functions etc. belonging to the namespace
}
```

  Here the `Name` is an arbitrary name which complies with the rules for variable and function names.
- In order to use a construct from a namespace, the name of the namespace followed by two colons must be written before the name of the construct, e.g. `std::max(a,b)`.
- Each class defines its own namespace.

Namespaces

- With the keyword `using` one or all of the names from another namespace are made available to the current namespace. An example that is often used is the line

  ```
  using namespace std;
  ```

  After this line, all constructs from the namespace `std` can be used without a prefix, e.g. `max(a,b)`. This must not lead to ambiguity.
- If only one name (or a small number of names) should be imported, it can be specified explicitly, e.g. `using std::cout;`
- The keyword `using` should be used sparingly.
- Namespaces may also be nested as a hierarchy.

Namespaces: Example

Namespaces are particularly useful when there is a possibility that two classes, global variables or functions with the same name (and for functions same argument list) exist in different parts of the code developed independently from each other. This leads to errors with the error message `... redefined`. Using namespaces this can be prevented:

```
// namespaces
#include <iostream>

namespace first
{
  int var = 5;
}

namespace second
{
  double var = 3.1416;
}

int main ()
{
  std::cout << first::var << endl;
  std::cout << second::var << endl;
  return 0;
}
```

Nested Classes

- Often a class needs other “auxiliary classes”.
- Those may be specific to the implementation and shouldn’t be visible from the outside.
- Example:
  - List elements
  - Iterators
  - Exceptions (Objects for error messages, next lecture)
- One can realise those as classes within the class (so-called nested classes).
- Advantages:
  - The global namespace is not “polluted”.
  - Affiliation with the other class is emphasized.

Nested Classes: Example

```
class Outer
{
  public:
    ...
    class Inner1
    {
        ...
    };
  private:
    ...
    class Inner2
    {
        void foo();
    };
};

void Outer::Inner2::foo()
{
    ...
}
```

Example: Implementation of a Set using a List

```
class Set
{
  public:
    Set();                      // empty set
    ~Set();                     // delete the set
    void Insert(double);        // insert (only once)
    void Delete(double);        // delete (if in set)
    bool Contains(double); // true if contained in set

  private:
    struct SetElem
    {
        double    item;
        SetElem* next;
    };
    SetElem* first;
};
```

`SetElem` can only be used in the `Set`, therefore all its attributes can be `public` (Remember: `struct` is `class` with `public` as default).

Inheritance

- Classes allow the definition of components that represent certain concepts of the real world or the program.
- The relationship between the various classes can be expressed through inheritance. e.g. the classes `Circle` and `Triangle` have in common that they represent a geometric shape. This should also be reflected in the program.
- In C++ it is possible to write:

```
class Shape {...};
class Circle   : public Shape {...};
class Triangle : public Shape {...};
```

  The classes `Circle` and `Triangle` are derived from `Shape`, they inherit the properties of `Shape`.
- It is thus possible to summarize common characteristics and behaviors of `Circle` and `Triangle` in `Shape`. This is a new level of abstraction.

Inheritance

- A derived class is an
  - extension of the base class. It has all the properties of the base class and adds some more.
  - specialization of the base class. As a rule, it represents a particular realization of the general concept.
- The interplay of expansion and restriction is the source of the flexibility (but also sometimes the complexity) of this technique.

Protected Members

- Next to `private` and `public` class members, there is a third category: `protected`
- It is not possible to access `protected` methods and attributes from the ouside, only from the class itself, as with `private`
- However, `protected` methods and attributes stay `protected` when using `public` inheritance, this means they can also be accessed by all derived classes.
- There is the widespread opinion that `protected` isn't needed and that the use of this type is an indication of design errors (such as missing access functions...).

Protected Members: Example

```
class A
{
  protected:
    int c;
    void f();
};

class B : public A
{
  public:
    void g();
};

B::g()
{
    int d=c;   // allowed
    f();       // allowed
}

int main()
{
    A a;
    B b;
    a.f();     // not allowed
    b.f();     // not allowed
}
```

Protected Constructors

With the help of `protected` one can prevent the creation of objects of the base class:

```
class B
{
  protected:
    B();
};

class D : public B
{
  public:
    D();      // calls B()
};

int main()
{
    B b;  // not allowed
    D d;  // allowed
}
```

Class Relations

Is-a Class `Y` has the same functionality (maybe in specialised form) as class `X`.
Object `y` (of class `Y`) can be used as an `x` (of class `X`).
Example: a VW Beetle is a car

Has-a (aggregation): Class `Z` consists of subobjects of type `X` and `Y`.
`z` has an `x` and a `y`.
Example: a car has a motor, doors, tires, ...

Knows-a (assoziation): Class `Y` has a reference (or pointer) to objects of class `X`.
`x` knows a `y`, uses a `y`.
Example: A car is registered to a person (the person possesses it, but isn't made of it, it isn't a part of her or him).

One can implement has-a (in possession of) using knows-a.

Public Inheritance

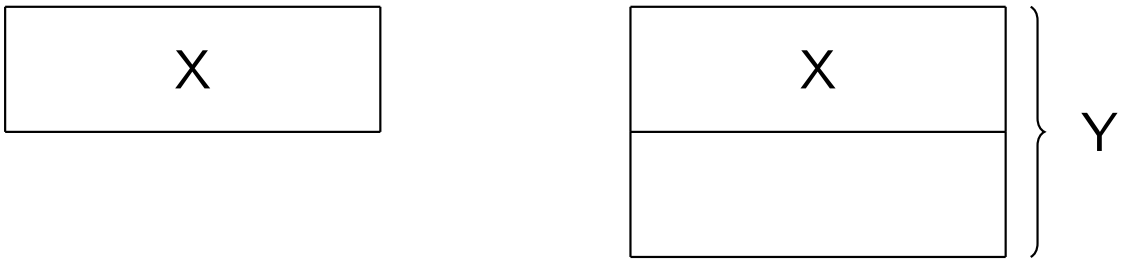
```
class X
{
  public:
    void a();
};

class Y : public X
{
  public:
    void b();
};
```

- All `public` members of `X` are `public` members of `Y`
- The implementation is inherited, i.e.

```
Y y;
y.a();      // calls method a of X
```

Public Inheritance

- Is-a-relation



- Objects of the derived class can be used as objects of the base class, but then only the base class part of the object is accessible.

Slicing

```
class X
{
  public:
    void a();
};

class Y : public X
{
  public:
    void b();
};

int main()
{
    Y y;
    y.a();      // calls method a of the X part of y
    X &x = y;
    x.a();      // calls method a of the X part of y
    x.b();      // not allowed, only methods of X accessible
}
```

If an object of the derived classe is passed call-by-value instead of an object of the base class, then only the base class part is copied.

Private Inheritance

```
class X
{
  public:
    void a();
};
```

```
class Y : private X
{
  public:
    void b();
};
```

- All `public` members of `X` are `private` members of `Y`
- The has-a relation is in principle equivalent to :

```
class Y
{
  public:
    void b();

  private:
    X x;    // aggregation
}
```

  Therefore, private inheritance is not particularly essential.
- It is used to implement a class by means of another.

Protected Inheritance

```
class X
{
  public:
    void a();
};

class Y : protected X
{
  public:
    void b();
};
```

- All `public` members of `X` are `protected` members of `Y`
- Is actually never needed.

Overview: Access Control in Inheritance

Access Rights in the Base Class	Inheritance Type: public	protected	private
public	**public**	**protected**	**private**
protected	**protected**	**protected**	**private**
private	–	–	–

Access to Methods and Attributes of the Base Class

- If there is a variable or method in the derived class with the same name as in the base class (including argument list in the case of methods), then it hides the corresponding variable or method in the base class.
- Access is still possible using the name of the base class as a prefix namespace identifier, as long as this is permitted by the access rights.

Multiple Inheritance

- A class can be derived from more than one base class.
- If there are any methods or variables with the same name in two or more base classes, then they must be identified by the namespace of the relevant class.
- The constructors of the base classes are called in the order of derivation.
- Should only be used in exceptional cases. Often this can also be solved using a has-a relation (i.e. via an appropriate attribute).

Multiple Inheritance

```
#include<iostream>

class TractionEngine
{
  public:
    float Weight()
    {
      return weight_;
    };
    TractionEngine(float weight) : weight_(weight)
    {
      std::cout << "TractionEngine initialized" << std::endl;
    };

  protected:
    float weight_;
};
```

Multiple Inheritance

```
class Trailer
{
  public:
    float Weight()
    {
      return weight_;
    };
    Trailer(float weight) : weight_(weight)
    {
      std::cout << "Trailer initialized" << std::endl;
    };
  protected:
    float weight_;
};
```

Multiple Inheritance

```
class TrailerTruck : public TractionEngine, public Trailer
{
  public:
    float Weight()
    {
        return TractionEngine::weight_+Trailer::weight_;
    }
    TrailerTruck(float wEngine, float wTrailer) :
        Trailer(wTrailer),
        TractionEngine(wEngine)
    {
      std::cout << "TrailerTruck initialized" << std::endl;
    };
};
```

Multiple Inheritance

```
int main()
{
  TrailerTruck mikesTruck(10.0,25.0);
  std::cout << "Weight trailer truck:   " << mikesTruck.Weight()
      << std::endl;
  std::cout << "Weight traction engine: " <<
      mikesTruck.TractionEngine::Weight() << std::endl;
  std::cout << "Weight trailer:         " <<
      mikesTruck.Trailer::Weight() << std::endl;
}
```

Output:

```
TractionEngine initialized
Trailer initialized
TrailerTruck initialized
Weight trailer truck:   35
Weight traction engine: 10
Weight trailer:         25
```

C++11: Final

```
class X final
{
  public:
    void a();
};

class Y : public X  // compiler error
{
  public:
    void b();
};
```

In C++11 it is allowed to mark a class as `final`. Then, further derivation from this class is no longer allowed.

Benefits of Inheritance

Software reuse Common features do not have to be written again every time. Saves time and improves security and reliability.

Code sharing Code in the base class is not duplicated in the derived class. Errors must only be corrected once.

Information hiding The class can be changed without knowing the implementation details.

Closed source extension Is also possible with classes that are only distributed as binary code and a header with declarations.

Drawbacks of Inheritance

Runtime speed Calling all constructors and destructors when creating and destroying an object, possibly a higher memory consumption when a derived class does not use all the features of the base class.

Program size When using general libraries unnecessary code may be written.

Program complexity High program complexity can be caused by excessive class hierarchies or multiple inheritance.

Summary

- Namespaces allow the separation of programs and libraries into logical parts and avoid name clashes
- Nested classes are a good way to hide classes that are only needed within one other class
- Inheritance makes code reuse and the expression of hierarchies possible
- `protected` and `private` inheritance are rarely needed / used
- Objects of derived classes can be used as objects of their base class, this may lead to slicing
- Inheritance may make coding faster and easier, but may also reduce the efficiency of the resulting code