Object-Oriented Programming for Scientific Computing

Namespaces and Inheritance

Ole Klein

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

16. Mai 2017

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 `Name` is an arbitrary name in compliance 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`.

Common examples: classes "Vector" and "Matrix", function "mean" ...

Using namespaces this can be prevented:

```
#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” for specific tasks.

- These may be specific to the implementation and in many cases shouldn't be visible from the outside.
- Example:
    - List elements (nodes, see exercise)
    - Iterators (later)
    - Exceptions (objects for error messages, next lecture)
- One can realize 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

```cpp
class Outer
{
  public:
    ...
    class Inner1 // e.g., iterators
    {
        ...
    };
  private:
    ...
    class Inner2 // e.g., nodes
    {
        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 remove(double);      // delete (if in set)
    bool contains(double);    // true if contained in set

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

`SetElem` can only be used in `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 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`, and 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.
- It is important to make sure that derived classes retain all properties of the original class.

Rectangle and Square

Is a `Square` a `Rectangle`?

The answer depends on whether / how objects can be modified after creation:

1. Height and width are fixed during construction and constant for the whole lifetime of objects
    - Then, it is perfectly fine to define `Square` as a class derived from `Rectangle`
2. There is only a method for scaling with fixed aspect ratio
    - Same as above
3. Height and width can be modified independently in `Rectangle` (this is the standard case!)
    - Keeping this behavior for `Square` violates basic assumptions for being a square
    - Keeping the aspect ratio constant for `Square` would lead to surprising behavior when used as a rectangle

Both "solutions" violate a class invariant (being a square, resp. doing exactly what is written on the tin when calling a method)

$\Rightarrow$ class hierarchies have to take use cases into account (principle of least surprise)

Protected Members

Next to `private` and `public` class members, there is a third category: `protected`

- Just as with `private`, it is not possible to access `protected` methods and attributes from the outside, only from the class itself.
- 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 its use 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 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

There are three main relations between classes:

Is-a Class `Y` has the same functionality (maybe in specialized form) as `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`.
Object `z` has an `x` and a `y`.
Example: a car has a motor, doors, tires, ...

Knows-a (association): Class `Y` has references / pointers 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*, as long as it is guaranteed that access is exclusive.

Public Inheritance

Public inheritance is the standard way of expressing *is-a* in C++:

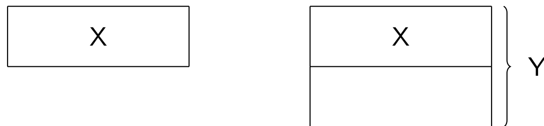
```
class X
{
  public:
    void a();
};

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

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

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

Public Inheritance

- Is-a relation:



- The derived class `Y` fulfills the interface of its base class `X`.
- 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

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.

```
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
}
```

Private Inheritance

Private inheritance is a form of *has-a*:

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

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

- All `public` members of `X` are now `private` members of `Y`
- This form of 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

There is also 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.

Rather obscure application for private / protected inheritance: need to change virtual function in a component (later: Dynamic Polymorphism)

Overview: Access Control in Inheritance

Summary of visibility changes:

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

Public inheritance: derived class can be used like base class (is extension)

Private inheritance: inheritance is hidden implementation detail (simply use member variables instead)

Protected inheritance: same as private, but implementation detail is exposed to further derived classes down the line

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.

```cpp
class A
{
  public:
    void foo() {...}
};

class B : public A
{
  public:
    void foo() {...}
};
```

```cpp
int main()
{
  B b;
  A& a = b;
  b.foo();     // calls foo of B
  b.A::foo(); // explicitly calls foo of A
  b.B::foo(); // redundant, but allowed
  a.foo();     // calls foo of A
  a.A::foo(); // redundant, but allowed
  a.B::foo(); // not allowed (slicing)
}
```

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 appropriate attributes).

Application: Design by Contract

- Generate multiple abstract base classes (later: Dynamic Polymorphism)
- Abstract base class (ABC) defines interface
- Inherit from all ABCs to ensure that derived class is fully functional

Main danger of multiple inheritance: Deadly Diamond of Death (later)

Multiple Inheritance

```cpp
#include<iostream>

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

  protected:
    float weight;
};
```

Multiple Inheritance

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

  protected:
    float weight;
};
```

Multiple Inheritance

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

Multiple Inheritance

```
int main()
{
  TrailerTruck mikesTruck(10.,25.);
  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: Inheriting Constructors

The keyword `using` can be used to import from the namespace of a base class, just like any other namespace. This is especially useful when the base class is templatized and names are not resolved automatically (later).

```
// make max available without cumbersome namespace declaration
using Base<T>::max();
```

Before C++11, this was not allowed for constructors, these had to be reimplemented even if they just pass arguments along:

```
Derived(int long_, int list_, int of_, int arguments_)
  : Base(long_,list_,of_,arguments_)
{}

// several other constructors that pass different args along
```

Since C++11, the following is valid:

```
// implicitly defines Derived(int,int,int,int)
// (and all other constructors that aren't overloaded locally)
using Base::Base;
```

C++11: Final

In C++11 and above it is allowed to mark a class as `final`. This means further derivation from this class is no longer allowed.

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

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

Marking classes as `final` restricts potential users of the class and should only be used if there's a good reason to do it.

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 extended without knowing its 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 higher memory consumption when the derived class does not use all features of the base class.

Program size
: Unnecessarily long code may have to be written to use general libraries in a given context.

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 (usually not an issue)