# Object-Oriented Programming for Scientific Computing

## Templates and Static Polymorphism

Ole Klein

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

Summer Semester 2018

# Dependent Types: Keyword typename

```
template<typename T, int dimension = 3>
class NumericalSolver
{
    ...
  private:
    typename T::SubType value_type;
}
```

- Template classes often define *dependent types* (e.g. to determine return types of methods as a function of the template parameters).
- A C++ compiler can't know what the construct `T::Name` is (for a template argument `T`), as that requires the class definition of `T`. It therefore assumes by default that this is a static variable.
- If it is instead a type defined in the class, then this must be specified with the keyword `typename`.
- This is only required within function or class templates (in other places it is clear what exactly `Name` means).
- It isn't needed in a list of base class specifications or in an initialization list (because here it can't be a static variable).

# Member Templates

Class members (methods or nested classes) can be templates as well.

```
template<typename T>
class Stack
{
  private:
    std::deque<T> elems;
  public:
    void push(const T&);
    void pop();
    T top() const;
    bool empty() const
    {
      return elems.empty();
    }

    //assignment of stack of elements of type T2
    template<typename T2>
    Stack<T>& operator=(const Stack<T2>&);
};
```

In this example, the default assignment operator is overloaded, not replaced (see the rules for overloading template functions).

# Member Templates

```
template<typename T>
template<typename T2>
Stack<T>& Stack<T>::operator=(const Stack<T2>& other)
{
  Stack<T2> tmp(other);

  elems.clear();
  while(!tmp.empty())
  {
    elems.push_front(tmp.top());
    tmp.pop();
  }
  return *this;
}
```

- We now need two `template` lines for the method definition (one for the class template, and one for the function template).
- As `Stack<T>` and `Stack<T2>` are completely different types, one can only use the interface (public part of class). To gain access to the lowest elements of the stack, a copy is created and then gradually broken down with pop.

# Member Templates

Usage:

```
int main()
{
  Stack<int>   intStack;
  Stack<float> floatStack;

  intStack.push(100);
  floatStack.push(0.);
  floatStack.push(10.);
  floatStack = intStack;   // OK, int converts to float
  intStack   = floatStack; // here information may be lost
}
```

# Keyword .template

```
class A
{
  public:
    template<class T> T doSomething() { };
};

template<class U> void doSomethingElse(U variable)
{
    char result = variable.template doSomething<char>();
}

template<class U,typename V> V doSomethingMore(U* variable)
{
    return variable->template doSomething<V>();
}
```

- There is also ambiguity regarding the $<$ character: by default, a C++ compiler assumes that "<" marks the beginning of a comparison. With templates this results in cryptic error messages like `error: expected primary-expression before ...`.
- When the $<$ is part of a method name which explicitly depends on a template parameter, then the keyword `template` must be inserted before the method name. This is needed with ".", "::", and "->".

# Template Template Parameters

Template parameters can themselves be templates instead of values or types:

- In the Stack class with interchangeable container, the user had to specify the type of container him/herself:

  ```
  Stack<int,std::vector<int> > myStack;
  ```

  This is verbose and error-prone (the types need not match).

- It is better to write this with a template template parameter:

```
template<typename T,
    template<typename> class C = std::deque>
  class Stack
  {
    private:
      C<T> elems;
      ...
  }
```

# Template Template Parameters

- *Usage:* one of the few places for using the template name (!), not a full type

```
Stack<int,std::vector> myStack;
```

- Within the class, the template provided as a template template parameter can be instantiated with any type combination, not just with the template parameters of the class.
- The arguments of the template must exactly match the arguments of the template template parameter. Here default values are *not* applied.
- While the template template parameter is necessarily a class template, starting with C++17 it is also possible to use the generic keyword `typename` instead of `class`:

```
template<typename T,
    template<typename> typename C = std::deque> ...
```

instead of

```
template<typename T,
    template<typename> class C = std::deque> ...
```

# Stack with Template Template Parameter

```
template<typename T, template<typename U,
                              typename = std::allocator<U> >
                     class C = std::deque>
class Stack
{
  private:
    C<T> elems;
  public:
    void push(const T&);
    void pop();
    T top() const;
    bool empty() const
    {
      return elems.empty();
    }

    //assignment of stack of elements of type T2
    template<typename T2, template<typename, typename> class C2>
    Stack<T,C>& operator=(const Stack<T2,C2>&);
};
```

# Stack with Template Template Parameter II

```
template<typename T, template<typename, typename> class C>
void Stack<T,C>::push(const T& elem)
{
  elems.push_back(elem);
}

template<typename T, template<typename, typename> class C>
void Stack<T,C>::pop()
{
  if(elems.empty())
    throw std::out_of_range("Stack<>::pop(): empty stack");
  elems.pop_back();
}

template<typename T, template<typename, typename> class C>
T Stack<T,C>::top() const
{
  if(elems.empty())
    throw std::out_of_range("Stack<>::pop(): empty stack");
  return elems.back();
}
```

# Stack with Template Template Parameter III

```
template<typename T,  template<typename, typename> class C>
template<typename T2, template<typename, typename> class C2>
Stack<T,C>& Stack<T,C>::operator=(const Stack<T2,C2>& other)
{
  Stack<T2,C2> tmp(other);
  elems.clear();
  while(!tmp.empty())
  {
    elems.push_front(tmp.top());
    tmp.pop();
  }
  return *this;
}
```

# Stack with Template Template Parameter IV

Usage:

```
int main()
{
  Stack<int>                intStack;   // implicitly uses deque
  Stack<float,std::list> floatStack; // uses list instead

  intStack.push(100);
  floatStack.push(0.);
  floatStack.push(10.);
  floatStack = intStack;   // OK, int converts to float
  intStack   = floatStack; // here information may be lost
```

# Initialization with Zero

- In C++, the variables of built-in types (such as `int`, `double`, or pointers) won't be initialized with default values for performance reasons.
- Each uninitialized variable has undefined content (i.e. the random entries at its memory location):

```
template<typename T>
void foo()
{
  T x; // x has undefined value if T is a built-in type
}
```

- However, it is possible to explicitly invoke a default constructor for built-in types that sets the variable to zero (or `false` in the case of `bool`):

```
template<typename T>
void foo()
{
  T x(); // x is zero (or false) when T is a built-in type
}
```

# Initialization with Zero

- To ensure that all variables in a class template are going to be initialized, a constructor has to be explicitly called for all attributes in the initialization list:

```
template<typename T>
class MyClass
{
  private:
    T x;
  public:
    MyClass() : x() //initializes x, also for built-in T
    {}
    ...
};
```

# C++11: Template Aliases

We have already discussed "template aliasing" as an alternative for `typedef`s. While `typedef`s only work for types, i.e. fully resolved templates or non-template types, template aliasing can also be used to set only some of the template arguments:

```
template <typename T, int U>
class GeneralType
{};

template <int U>    // for partially resolved templates
using IntName = GeneralType<int,U>;

int main()
{
  using int32 = int;                    // for normal types
  using Function = void (*)(double);    // for functions

  // for fully resolved templates
  using SpecialType = GeneralType<int,36>;

  // uses partially resolved template
  IntName<7> foo;
}
```

# Independent Base Classes

The combination of object-oriented programming (inheritance) and generic programming (templates) leads to complications during name lookup that have to be studied in detail. C++ treats *dependent* and *independent* base classes in different ways.

*Independent* base classes are those base classes that are completely determined without considering template parameters.

- Independent base classes behave essentially like base classes in normal (non-template) classes.
- If a name appears in a class but no namespace precedes it (an unqualified name), then the compiler will look in the following order for a definition:
  1. Definitions in the class
  2. Definitions in independent base classes
  3. Template arguments
- This order of name lookup can lead to surprising results.

*Independent* names are unqualified names that don't depend on a template parameter.

# Independent Base Classes

```
template<typename X>
class Base
{
  public:
    int basefield;
    typedef int T;
};

class D1 : public Base<Base<void> >
{
  public:
    void f()
    {
      basefield = 3; // access to inherited number
    }
};

template<class T>
class D2 : Base<double>
{ // independent base class
  public:
    void f()
    {
      basefield = 7; // access to inherited number
    }
    T strange; // has type Base<double>::T, i.e. int !!
};
```

# Independent Base Classes

The result is a class that should store a `double`, but stores an `int` instead:

```
int main()
{
  D1 d1;
  d1.f();
  D2<double> d2;
  d2.f();
  d2.strange=1;
  d2.strange=1.1; // beware: d2.strange has type int!
  std::cout << d2.strange << std::endl;
}
```

- The main issue is the fact that this can't be expected when looking only at class `D2`.
- Therefore, the naming schemes for types and for template arguments should differ (e.g. single letters, all uppercase ...).

# Dependent Base Classes

*Dependent* base classes are those that are not independent, i.e. they require the specification of at least one template argument to be fully defined.

- The C++ standard dictates that independent names appearing in a template are resolved at their first occurrence.
- The strange behavior from the last example relied on the fact that *independent* base classes have higher priority during name lookup than template parameters.
- For names defined in a *dependent* base class, the resolution would depend on one or more template parameters, unknown at that point.
- This would have consequences when an *independent* name would have its definition in a *dependent* base class.

# Dependent Base Classes

```
template<typename T>
class DD : public Base<T>
{
  public:
    void f()
    {
      // (1) would lead to type resolution
      //     and binding to int.
      basefield = 0;
      }
};

template<>
class Base<bool>
{
  public:
    // (2) Template specialization wants to
    //     define variable differently.
    enum { basefield = 42 };
};

void g(DD<bool>& d)
{
    d.f();  // (3) Conflict
}
```

1. In the definition of class `DD`, the first access to `basefield` in `f()` would lead to binding `basefield` to `int` (because of the definition in the class template).
2. Subsequently, however, the type of `basefield` would be modified into something unchangeable for the type `bool`.
3. In the instantiation (3) a conflict would then occur.

# Solution: Delayed Type Resolution

- In order to prevent this situation, C++ defines that *independent* names won't be searched in *dependent* base classes. The C++ compiler stops already at (1) and displays an error message (`error: 'basefield' was not declared in this scope`).
- The base class attributes and methods must therefore be prefixed by either "`this->`" or "`Base<T>::`".
- As a result, the name is *dependent* (contains template parameters) and so will only be resolved during instantiation.
- Example with `this->`:

```
template<typename T>
class DD : public Base<T>
{
  public:
    void f()
    {
      this->basefield = 0;
    }
};
```

# Solution: Delayed Type Resolution

or, with explicit namespace:

```
template<typename T>
class DD : public Base<T>
{
  public:
    void f()
    {
      Base<T>::basefield = 0;
    }
};
```

or shorter:

```
template<typename T>
class DD : public Base<T>
{
    using Base<T>::basefield; // name is now dependent
                              // for whole class

  public:
    void f(){ basefield = 0; } // finds Base<T>::basefield
};
```

# Unified Modeling Language (UML)

Class hierarchies can be quite complex. Therefore, it can be useful to represent them graphically.

- The Unified Modeling Language (UML) is the standard. It is used for visualization, specification, construction and documentation of object-oriented software.
- It is the result of several predecessors (e.g. Booch, OOSE and OMT).
- Version 1.0 was released in September 1997.
- UML became an official ISO standard in 2005.
- There are also plugins for development environments (e.g. Eclipse) that automatically generate code from UML diagrams.

# UML Diagram Types

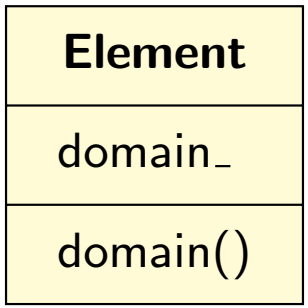
UML provides 9 different kinds of diagram:

1. Class diagram
2. Object
3. Use case
4. Sequence
5. Collaboration
6. Statechart
7. Activity
8. Component
9. Deployment

We treat only a tiny part of this extensive tool set (much more of this is considered in Software Engineering, for example).

# Classes

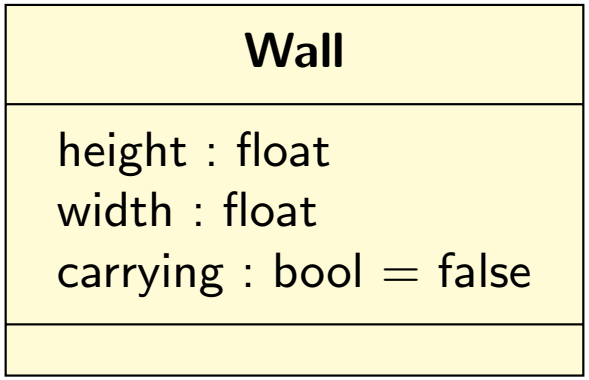
- Classes are represented by rectangular boxes, which are divided into different sections by horizontal lines.
- The name of the class comes first, followed by its attributes and then its methods.




A finite element object stores its domain internally and provides read-only access through a method.

# Attributes

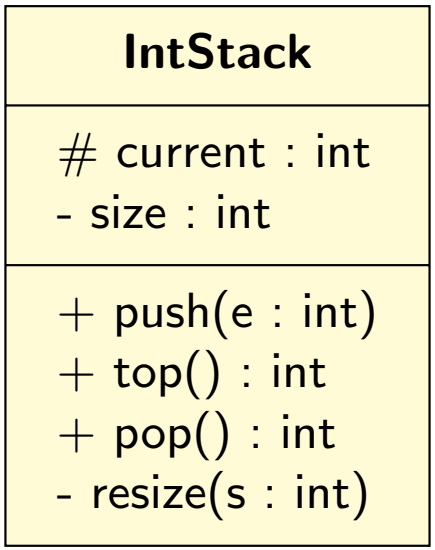
- Attributes can have a type and optionally a default value.




An object representing a wall has a specific height and width. By default a wall is not a carrying wall.

# Access Control

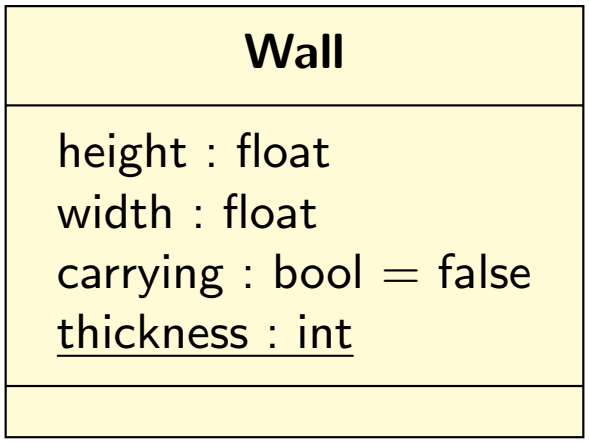
- The access rights to attributes are expressed by the leading characters + for `public`, # for `protected` and - for `private`.
- Return types are specified like types of variables, and the type of arguments of methods may also be specified.




A stack object for `int`s provides methods for storing / retrieving elements (push/pop) and reading the last one (top), and has one internal method (resize). Its maximum size (size) is private, while its current size (current) is available to derived classes.

# Static Class Members

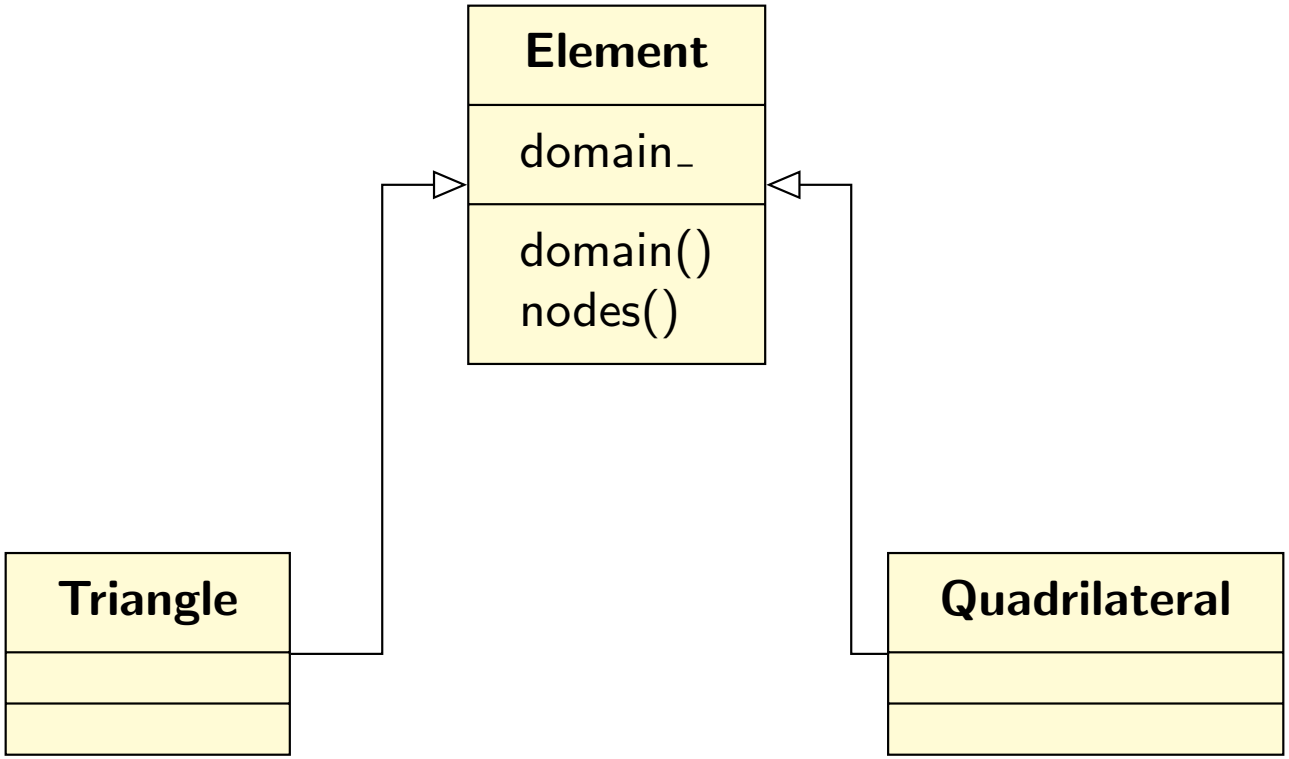
- Static class members are indicated by underlining.




While each wall has its own height and width, all walls have the same thickness.

# Inheritance (is-a)

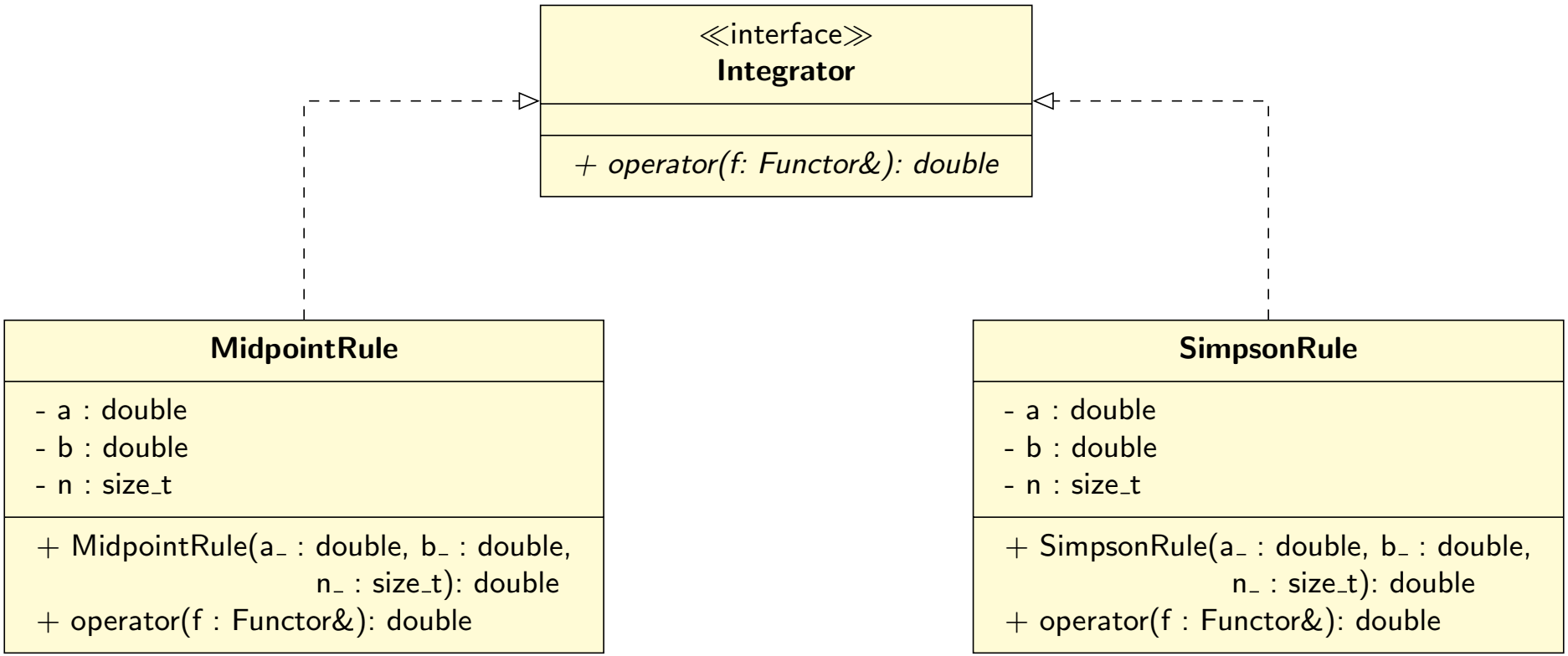
- When a class is derived from another one, this is marked by a line connecting the two classes, with an unfilled triangle at the base class opening itself in the direction of the derived class.




Triangles and quadrilaterals are two different types of finite element.

# Interface Base Classes, Pure Virtual Functions

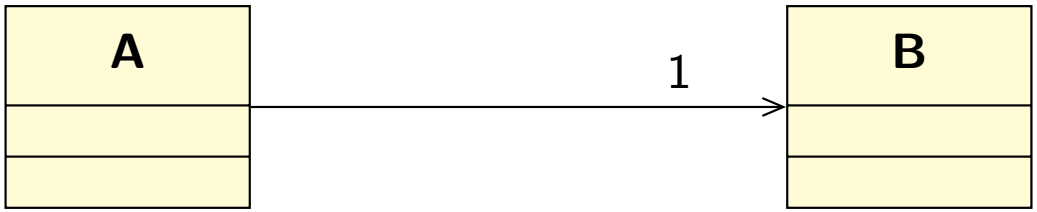
- Interface base classes are marked with the line ≪interface≫.
- The connections to classes which implement the interface are like normal inheritance, but dashed.
- Pure virtual functions are displayed using italics.



# Association (knows-a)

The association between two elements is represented by a connecting line. This can include the number of connections and a name for the connection. Association means there is a relationship between two entities, such as between a bank account and a customer.

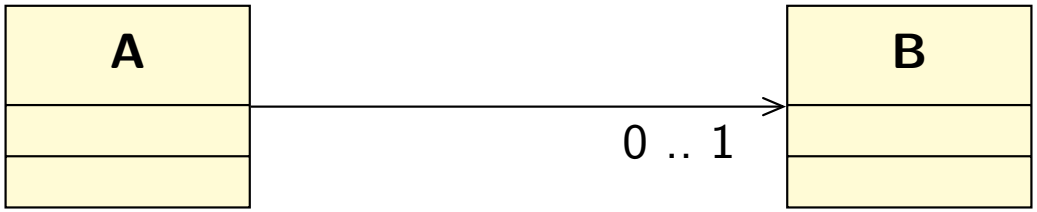
A is associated with a B.



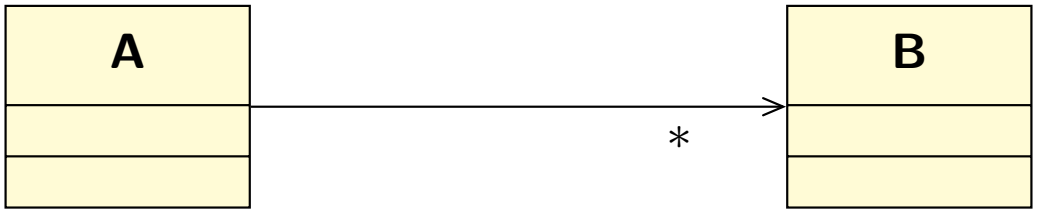

A is associated with one or more B.

# Association (knows-a)

A is associated with zero or one B.

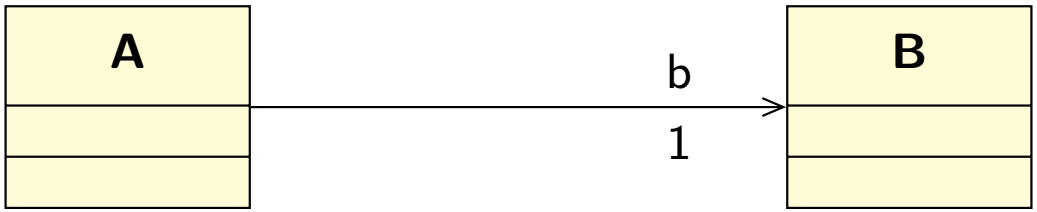



A is associated with zero, one or several B.



# Association (knows-a)

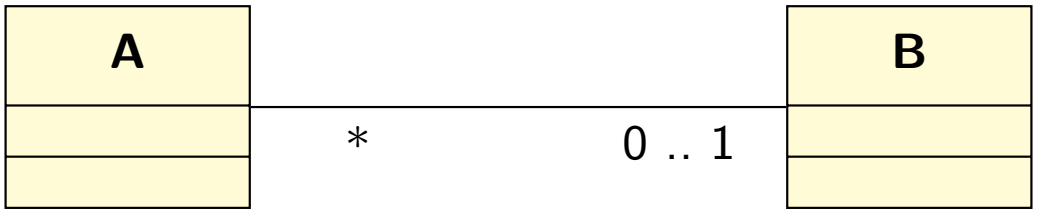
An association may have a name:




e.g.

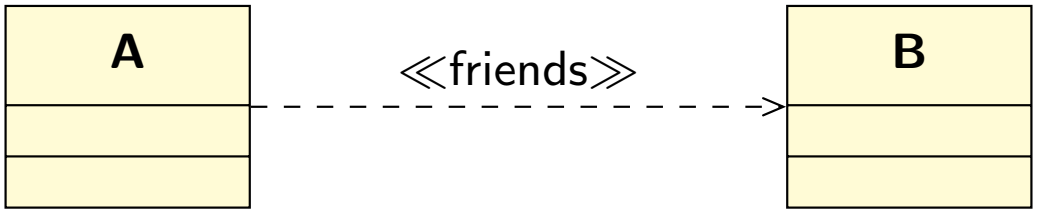
```
class B;
class A
{
    B* b;
};
```

and there is also association in both directions:



# Dependencies (uses-a)

A class can depend on another, e.g. because it is a `friend`:

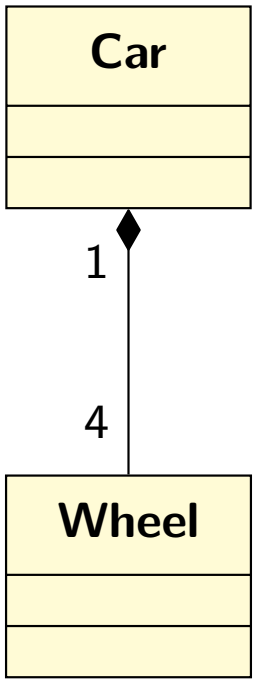



Just as with interface classes, the connecting line is dashed, because the connection is again more on the abstract level of structure (fulfills interface, has access to internals, ...), not the concrete level of interaction.

This connection can also be used to represent the fact that `A` requires knowledge about `B` in any way, e.g. because one of the methods of `A` interacts with objects of type `B`.

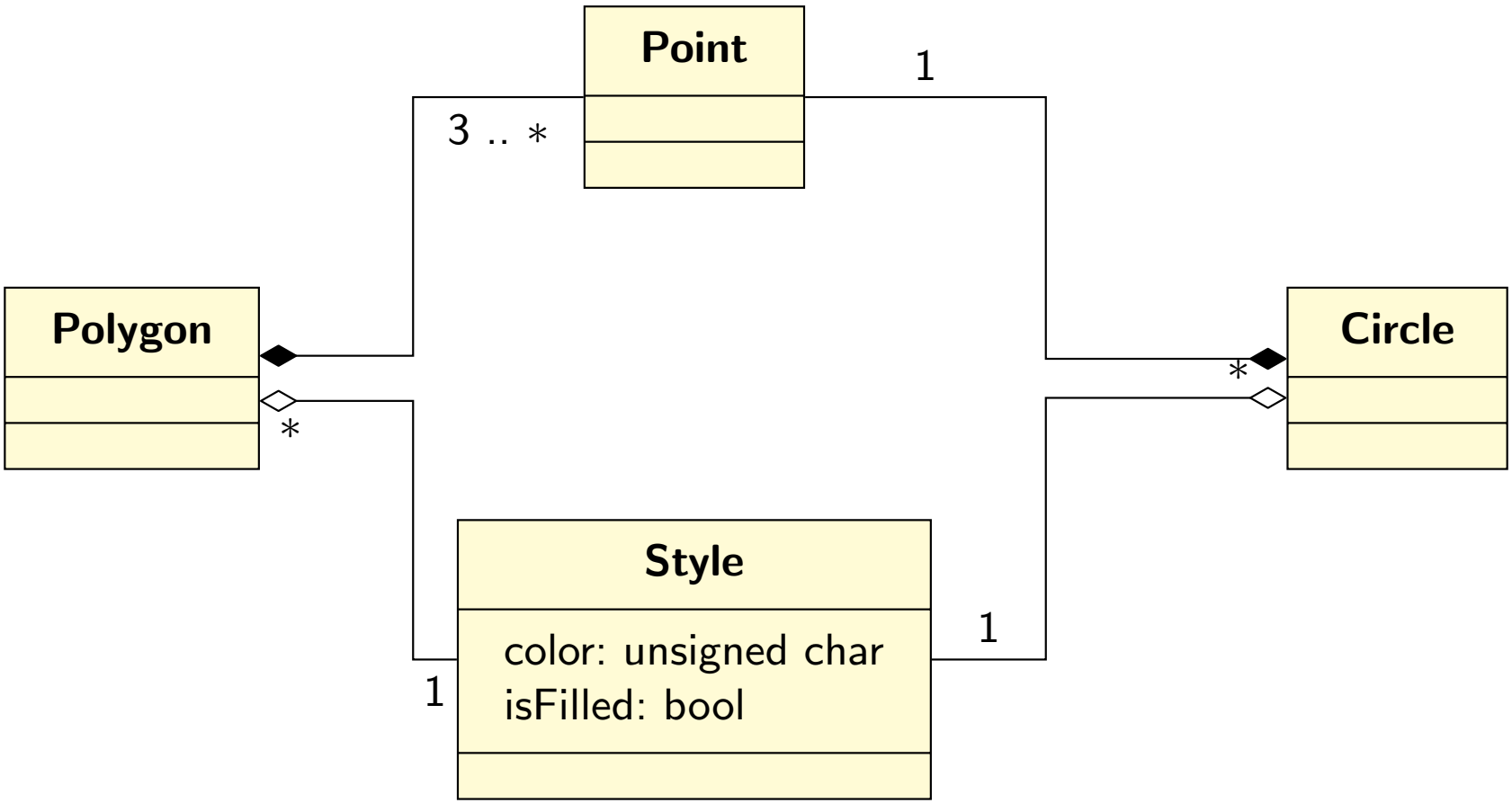
# Composition (has-a)

Composition is represented by an interconnecting line with a filled lozenge on the side of the composite class. A composition consists of parts which belong to a whole that bears the responsibility and has ownership. Compositions are usually *n*-to-one relationships.



# Aggregation (has-a)

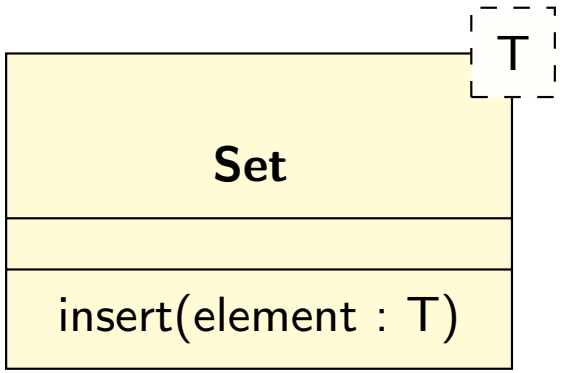
Aggregation is represented by connecting lines with an empty lozenge on the side of the aggregated class. Aggregation is a somewhat stronger relationship than association, but in contrast to composition dependent objects won't automatically be destroyed with the main object.



# Templates in UML

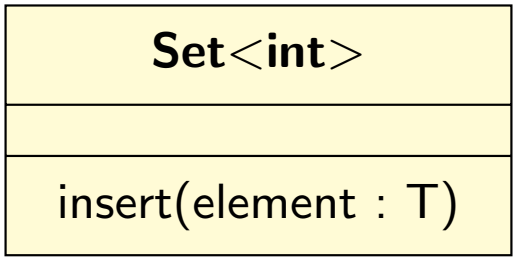
Class templates use the same representation as ordinary classes, but additionally the template parameters are listed in a small rectangle with dashed boundary.

```
template<typename T>
class Set
{
  void insert (T element);
};
```

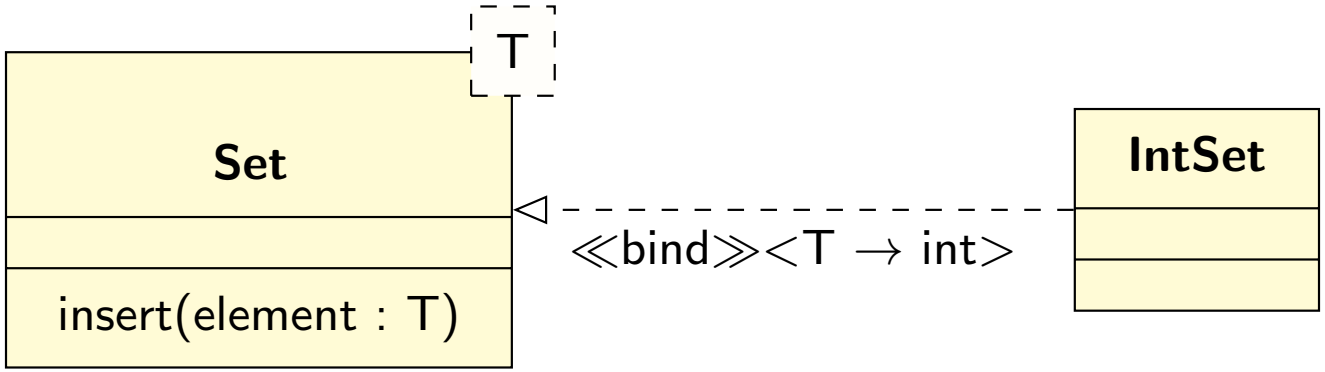


# Templates in UML

Realisation of `Set` with `T = int`:




or, also representing the template argument binding:



# Summary

- The keywords `typename` and `template` are often necessary to help the compiler parse template code (marking types and templates, respectively)
- Apart from types and values, one can also use templates as template parameters, so-called template template parameters
- C++ uses a two-phase name lookup scheme for templatized classes and distinguishes between dependent and independent base classes
- The Unified Modeling Language (UML) is an extensive toolset for the construction and documentation of object-oriented software
- Many constructs, processes and relationships in C++ can be mapped to visualizations in UML (only a small subset was shown)