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

2. Juni 2015

Keyword typename

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

- Template classes often define types (e.g. to determine the return type of functions as a function of the template parameters).
- A C++ compiler can not know what the construct `T::Name` is (here `T` is a typename template argument), as it does not yet know the class definition of `T`. It therefore assumes by default that this is a static variable.
- If it is a type defined in the class instead, then this must be clearly specified with the keyword `typename`.
- This is only required within function or class templates (otherwise it is clear what exactly `Name` means).
- It is not 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)
{
  if((void*)this==(void*)&other)
    return *this;

  Stack<T2> tmp(other);

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

- We now need two `template` lines at the start of the method definition.
- As `Stack<T>` and `Stack<T2>` are completely different types, one can only use the public part of the interface. 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(int argc, char** argv)
{
  Stack<int> intStack;
  Stack<float> floatStack;

  intStack.push(100);
  floatStack.push(0.0);
  floatStack.push(10.0);
  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>();
}
```

- Another ambiguity concerns the < character. A C++ compiler assumes by default that the sign < 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

- It may be necessary for a template parameter to be itself a class template.
- In the Stack class with interchangeable container, the user must specify the used type of container him/herself.

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

  In case the two types don't match, this is error-prone.

- 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;
   ...
}
```

- Usage:

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

Template Template Parameters

- Within the class, template template parameters can be instantiated with any type, not just with one of the template parameters of the class.
- The template template argument must exactly match the template template parameter for which it is used. Here default values aren't applied.

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)
{
  if((void*)this==(void*)&other)
    return *this;

  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(int argc, char** argv)
{
  Stack<int> intStack;
  Stack<float,std::deque> floatStack;

  intStack.push(100);
  floatStack.push(0.0);
  floatStack.push(10.0);
  floatStack=intStack; // OK, int converts to float
  intStack=floatStack; // here information may be lost
}
```

Initialization with Zero

- In C++ the variables of builtin types (such as `int`, `double`, or pointers) won't be initialized with default values for performance reasons.
- Each uninitialized variable has undefined content (the random entries of the 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 the type `bool`)

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

Initialization with Zero

- If it should be ensured that all the variables in a class template are going to be inintialized, then 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
    {
    }
    ...
};
```

C++11: Template Aliases

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

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

int main()
{
  using int32 = int;                      // for normal types
  using Function = void (*)(double);      // for functions
  using SpecialType = GeneralType<int,36>;  // for fully defined
      templates
  IntName<7> foo;
}
```

- In C ++11, there is an alternative method to `typedef`s to define abbreviations for long type names.
- This alternative is called “template aliasing”.
- It also allows to set some of the template arguments.

Independent Base Classes

- An independent base class is fully determined even without knowledge of a template parameter.
- Independent base classes behave essentially as base classes in normal (non-template) classes.
- If a name appears in the class but no namespace precedes it (an unqualified type), 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

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; // T has type Base<double>::T !!
};
```

Independent Base Classes

```
int main(int argc, char** argv)
{
  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;
}
```

Dependent Base Classes

- In the last example, the base class was completely defined.
- This does not apply for base classes which depend on an a template paramater.
- The C++ standard dictates that independent names appearing in a template are resolved at their first occurrence.

Dependent Base Classes

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

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

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 problem from happening, 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 and so will only be resolved during instantiation.
- Example

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

Solution: Delayed Type Resolution

or

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

or short:

```
template<typename T>
class DD : public Base<T>
{
    using Base<T>::basefield; // (1) is now dependent
                              // for whole class

  public:
    void f(){ basefield = 0; } // finds (1)
};
```

Unified Modeling Language (UML)

- Class hierarchies can be quite complex.
- It is useful to be able 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.
- There are also plugins for development environments (e.g. Eclipse) that are used to automatically generate code from UML diagrams.

UML Diagram Types

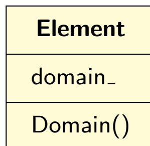
UML supplies 9 different kind of diagram:

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.

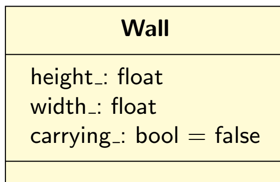
Classes

- 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.



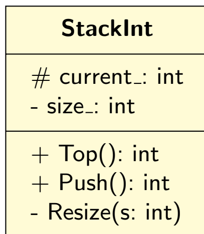
Attributes

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



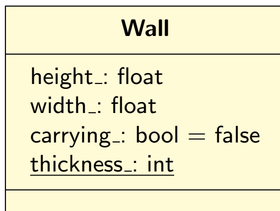
Access Control

- The access rights to attributes are expressed by the leading characters + for `public`, # for `protected` and − for `private`.



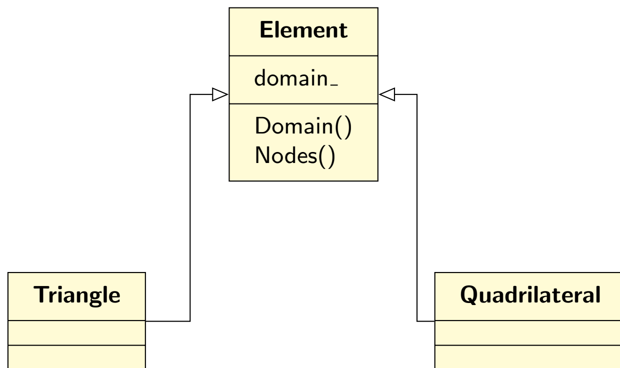
Static Class Members

- Static class members are indicated by underlining.



Inheritance

- 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:



Interface Base Classes, Purely Virtual Functions

- Interface base classes are characterized by the line <<interface>>.
- The connection to classes which implement the interface are like normal inheritance but dashed.

≪interface≫
Integrator

+ operator(f: Functor&): double

MidpointRule

- a_: double
- b_: double
- n_: size_t

+ MidpointRule(a: double, b: double, n: size_t): double
+ operator(f: Functor&): double

SimpsonRule

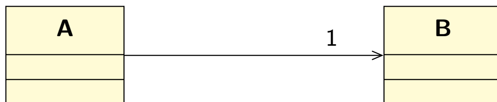
- a_: double
- b_: double
- n_: size_t

+ SimpsonRule(a: double, b: double, n: size_t): double
+ operator(f: Functor&): double

Association

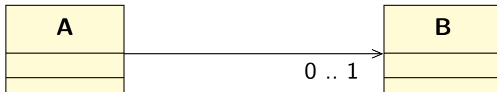
The association between two elements is represented by a connecting line. Numbers on the line can specify the number of connections and a name for the connection. An association means that there is a relationship between two elements such as between a bank account and a customer.

A is associated with a B.



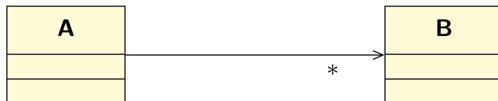
A is associated with one or more B.

A is associated with none or one B.



Association

A is associated with none, one or several B.



An association may have a name:

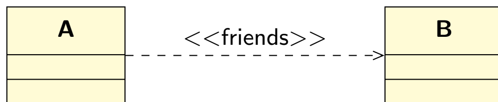
e.g.

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

and there is also association in both directions:

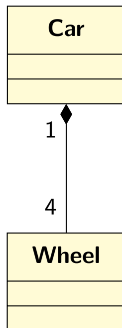
Dependencies

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



Composition

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 and for which it bears the responsibility. Compositions are usually n to one relationships.



Aggregation

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. A university for example can be represented as a composition of faculties, but these are only aggregations of professors.

Point

3 .. *

1

Polygon

*

Circle

*

Style

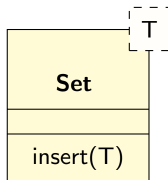
color: unsigned char
isFilled: bool

1

1

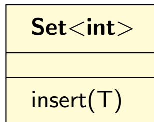
Templates in UML

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

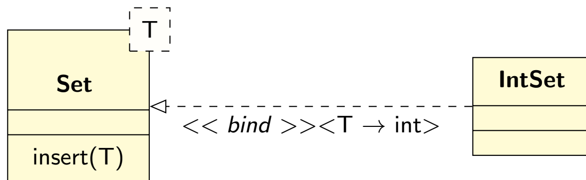


Templates in UML

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



or



The Standard Template Library(STL)

- The Standard Template Library (STL)
  - The Standard Template Library is a class library for diverse needs.
  - It provides algorithms to work with these classes.
- It also formulates interfaces, which must be provided by other collections of classes in order to be used as STL classes, or can be used to write algorithms that work with all STL-like container classes.
- The STL is a new level of abstraction, which frees the programmer from the necessity to write frequently used constructs such as dynamic arrays, lists, binary trees, search algorithms, and so on himself.
- STL algorithms are programmed as optimally as possible, i.e. if there is an STL algorithm for a problem, one should have a very good reason not to use it.
- Unfortunately, the STL is not self-explanatory.

STL Components

- The main components of the STL are:

  Containers are used to manage a particular type of object. The various containers have different properties and related advantages and disadvantages. The containers that fit best should be used in any given situation.

  Iterators make it possible to iterate over the contents of a container. They provide a uniform interface for each STL compliant container, regardless of its internal structure.

  Algorithms work with the elements of a container. They use iterators and therefore must only be written once for an arbitrary number of STL-compliant containers.

- At first sight, the structure of the STL partially contradicts the original idea of object-oriented programming that algorithms and data belong together.

Containers

STL container classes, or short containers, manage a collection of elements of the same type. Depending on the type of container, the STL gives assurances on the execution speed of certain operations.

There are two fundamentally different types of container:

Sequences
: are ordered sets of elements with freely selectable arrangement. Each element has its place, which depends on the program execution and not on the value of the element.

Associative Containers
: are ordered sets sorted according to a certain sorting criterion in which the position of an element depends only on its value.

Vector

STL sequence containers are class templates. There are two template arguments, the type of objects to be stored and a so-called allocator that can be used to change the memory management (this is useful for example when you create many small objects and don't want to pay the operating system overhead every time). The second parameter has a default value where `new()` and `delete()` are used.

Vector is a field of variable size.

- Adding and removing elements at the end of a `vector` is fast, i.e. complexity $O(1)$.
- The element can be accessed directly via an index (random access).

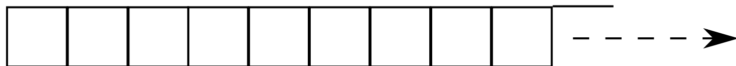


Abbildung: Structure of a `vector`

Amortized Complexity

- Typically, adding elements at the end of a `std::vector` is in $O(1)$.
- In individual cases, however, it may take much longer, especially if the allocated storage is no longer sufficient. Then new storage must be allocated, and often data has to be copied over. This is an $O(N)$ process.
- However, the standard library reserves memory blocks of increasing size for a growing vector. The overhead depends on the length of the vector. This optimizes the speed at the expense of memory usage.
- The $O(N)$-case therefore occurs very rarely.
- This is called "amortized complexity".
- If it is already known that a certain amount of elements is needed, then one can reserve space with the method `reserve(size_t size)`. This doesn't change the current size of the vector, it only reserves the right amount of memory.
- The same applies to the `deque`.

Example: STL Vector

```
#include <iostream>
#include <vector>
#include <string>

int main(){
    std::vector<double> a(7);
    std::cout << a.size() << std::endl;
    for (int i=0;i<7;++i)
        a[i] = i*0.1;
    double d = 4 * a[2];
    std::vector<double> c(a);
    std::cout << a.back() << " " << c.back() << std::endl;
    std::vector<std::string> b;
    b.resize(3);
    for (int i=2;i>=0;--i)
        std::cin >> b[i];
    b.resize(4);
    b[3] = "blub";
    b.push_back("blob");
    for (int i=0;i<b.size();++i)
        std::cout << b[i] << std::endl;
}
```

Deque

Deque is a “double-ended” queue, it is also a field of dynamic size, but:

- The addition and removal of elements is quick also at the beginning of `deque`, that is $O(1)$.
- Element access can again be achieved using an index, but the index of an element may change when elements are added to the beginning of the container.

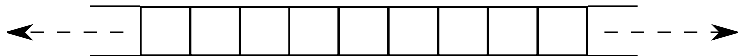


Abbildung: Structure of a `deque`

List

List is a doubly linked list of elements.

- There is no direct access to list elements.
- To reach the 10th element, one must start at the beginning of the `list` and traverse the first nine elements, access to a specific element is therefore $O(N)$.
- Adding and removing elements is fast in any location in the `list`, i.e. $O(1)$.

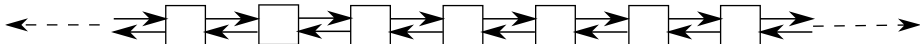


Abbildung: Structure of a `list`

Example: STL List

```
#include <iostream>
#include <list>
#include <string>

int main()
{
    std::list<double> vals;
    for (int i=0;i<7;++i)
      vals.push_back(i*0.1);
    vals.push_front(-1);
    std::list<double> copy(vals);
    std::cout << vals.back() << " " << copy.back() << std::endl;
    std::cout << vals.front() << " " << copy.front() << std::endl;
    for (int i=0;i<vals.size();++i)
    {
      std::cout << i << ": " << vals.front() << " " << vals.size() <<
           std::endl;
      vals.pop_front();
    }
    std::cout << std::endl;
    for (int i=0;i<copy.size();++i)
    {
      std::cout << i << ": " << copy.back() << " " << copy.size() <<
           std::endl;
      copy.pop_back();
    }
}
```

C++11: Array

Array is a C++11 replacement for the classical C arrays, i.e. a field of fixed size.

- The `std::array` hat two template parameters, the type of the stored objects and the number of elements of the container
- Adding and removing elements is not possible.
- Element access can be achieved directly via index.
- In contrast to C arrays, a `std::array` knows its size and can be used as the other STL containers.

Abbildung: Structure of an `array`

Example: STL Array

```
#include <iostream>
#include <array>
#include <string>

int main(){
    std::array<double,7> a;
    std::cout << a.size() << std::endl;
    for (int i=0;i<7;++i)
        a[i] = i*0.1;
    double d = 4 * a[2];
    std::array<double,7> c(a);
    std::cout << a.back() << " " << c.back() << std::endl;
    std::array<std::string,4> b;
    for (int i=2;i>=0;--i)
        std::cin >> b[i];
    b[3] = "blub";
    for (int i=0;i<b.size();++i)
        std::cout << b[i] << std::endl;
}
```