Programming Templates

Martin Heck

Remarks

- I’m the short term replacement for the original speaker.

- Rough Target Group for this course:

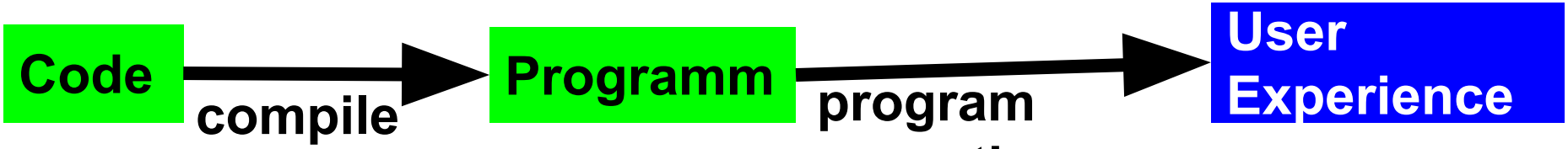
  BEGINNERS _____ **ADVANCED** _____ EXPERT

Overview

- (Reminder) on Template Basics
- Policies
- Some Examples of Usage

Motivation

- Runtime and Compile-time polymorphism can often be used alternatively.
- Pro template:
  - **Better compiler optimization**, as the compiler knows what code will run.
  - Easier to handle, if you have a **large number of very similar classes** (e.g. container classes, where you would need a subclass per data type)



Templates work similar to text replacements (not identical).

v-table look up at run time;

```
int main ()
{
  std::vector<int> myints;
  std::cout << "0. size: " << myints.size() << '\n';

  for (int i=0; i<10; i++) myints.push_back(i);
```

```
class CRectangle: public CPolygon {
  public:
    int area ()
      { return (width * height); }
  };
```

(Reminder) on Template Basics

```cpp
#include <iostream>
/** Getter for maximum value of two values of equal type.
 *
 *  T can a priori be any type, that has a > operator.
 *  As objects of type T can therefore be very large,
 *  we use references to them as inputs.
 */
template <typename T>    // let's use "typename" instead class, when used as template parameter;
T getMaxValue(const T& input1, const T& input2)
{
  if (input1 > input2) { // most critical thing: the specified type needs to have the > operator;
    return input1;
  }
  return input2;
}

using namespace std;

int main()
{
// Test function with int's
  int testInt1 = 3;
  int testInt2 = 5;
// Modern compilers are often able to deduce the used type without explicit specification.
  cout << getMaxValue(testInt1, testInt2) << endl;

// Test function with float's
  float testFloat1 = 1.2;
  float testFloat2 = 1.4;

  cout << getMaxValue(testFloat1, testFloat2) << endl;

// The following line of code would fail to compile;
// cout << getMaxValue(testFloat1, testInt2) << endl;

//Explicit determination of the function solves the issue, by triggering a cast.
  cout << getMaxValue<float> (testFloat1, testInt2) << endl;

}
```

Function- and Class- Templates allow to create semantically similar functions and classes with different parameters.
Example function template:

Compilation and execution yields:

```
 ./functionTemplatesProgram1
5
1.4
5
```

```cpp
#include <GetMaxValue.h>

#include <iostream>
#include <array>


using namespace std;

int main()
{
// Test function with pointers
  array <int*, 2> intPtrArray = {new int(2), new int(1)};

// Probably the result here is not really what one wants, but we still want to
// be able to program quite generic --> Specialisation
  cout  << "Maximum is " <<   getMaxValue(intPtrArray.at(0), intPtrArray.at(1)) << endl;
  cout  << "Pointer to " << *(getMaxValue(intPtrArray.at(0), intPtrArray.at(1))) << endl;
// Result, that we probably want:
  cout << "Highest Value that is pointed to in the array is: "
       << getMaxValue(*(intPtrArray.at(0)), *(intPtrArray.at(1))) << endl;

//Now let's use the smart function, that will give us the pointer to the largest value:
  cout  << "Maximum is " <<   getMaxValueSmart(intPtrArray.at(0), intPtrArray.at(1))  << endl;
  cout  << "Pointer to " << *(getMaxValueSmart(intPtrArray.at(0), intPtrArray.at(1))) << endl;

//Only with correct specialisation, not with overloading this is equal to the previous.
  cout  << "Maximum is " <<   getMaxValueSmart<int*>(intPtrArray.at(0), intPtrArray.at(1))  << endl;
  cout  << "Pointer to " << *(getMaxValueSmart<int*>(intPtrArray.at(0), intPtrArray.at(1))) << endl;


}
```

Specialisation helps in situations, where you need different behaviour for a special type.

Compilation and execution yields:

```
./functionTemplatesProgram2
Maximum is 0xb44030
Pointer to 1
Highest Value that is pointed to in the array is: 2
Maximum is 0xb44010
Pointer to 2
Maximum is 0xb44010
Pointer to 2
```

Our old getMaxValue function compares the pointer values, not the values pointed to.

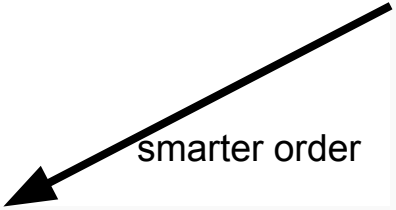
```
1  /** Getter for maximum value of two values of equal type.
2   *
3   *  T can a priori be any type, that has a > operator.
4   *  As objects of type T can therefore be very large,
5   *  we use references to them as inputs.
6   */
7  template <typename T>    // let's use "typename" instead, when used as template parameter;
8  T getMaxValue(const T& input1, const T& input2)
9  {
10   if (input1 > input2) { // most critical thing: the specified type needs to have the > operator;
11     return input1;
12   }
13   return input2;
14 }
15
16 /** Getter for maximum value of two values of equal type.
17  *
18  *  At first the smart version does the same as the "non-smart" one.
19  */
20 template <typename T>    // let's use "typename" instead, when used as template parameter;
21 T getMaxValueSmart(const T& input1, const T& input2)
22 {
23   if (input1 > input2) { // most critical thing: the specified type needs to have the > operator;
24     return input1;
25   }
26   return input2;
27 }
28
29 template<>               // removing this line is dangerous, leads to overloading!!!
30 int* getMaxValueSmart(int* const& input1, int* const& input2)
31 {
32   if (*input1 > *input2) {
33     return input1;
34   }
35   return input2;
36 }
```

Partial specialisation unfortunately works only for classes, not functions...

Note the interface:
const int *&
is something different than
int * const &

```
isation, not with overloading this is
:   getMaxValueSmart<int*>(intPtrArray
: *(getMaxValueSmart<int*>(intPtrArray
```

Deduction of template types is tried as good as possible as default values, but as with default values at functions, non-deducible values need come first.




```cpp
#include <iostream>          // Ein und Ausgabe

using namespace std;

// Unsere Vorlage für die Summenberechnung
template <typename R, typename Arg1, typename Arg2>
R summe(Arg1 obj1, Arg2 obj2){
    return obj1 + obj2;
}

int main(){
    double a = 5.4;                      // Gleitkommazahl-Variable
    int   b = -3;                       // Ganzzahlige Variable

    cout << summe<int>(a, b)             << endl; // int           summe(double obj1, int obj2);
    cout << summe<double>(a, b)           << endl; // double       summe(double obj1, int obj2);
    cout << summe<double>(a, b)          << endl; // double        summe(double obj1, int obj2);
    cout << summe<unsigned long>(a, b)   << endl; // unsigned long summe(double obj1, int obj2);

    cout << summe<double, int, int>(a, b) << endl; // double         summe(int  obj1, int obj2);

    cout << summe<double, long>(a, b)     << endl; // double         summe(long obj1, int obj2);

    return 0;                            // Programm erfolgreich durchlaufen
}
```

```cpp
#include <iostream>          // Ein und Ausgabe

using namespace std;

// Unsere Vorlage für die Summenberechnung
template <typename Arg1, typename Arg2, typename R> // Rückgabetyp als letztes angegeben
R summe(Arg1 obj1, Arg2 obj2){
    return obj1 + obj2;
}

int main(){
    double a = 5.4;                      // Gleitkommazahl-Variable
    int   b = -3;                       // Ganzzahlige Variable

    cout << summe<int>(a, b) << endl; // ??? summe(int obj1, int  obj2);
    cout << summe<double>(a, b) << endl; // ??? summe(double obj1, int  obj2);
    cout << summe<double>(a, b) << endl; // ??? summe(double obj1, int  obj2);
    cout << summe<unsigned long>(a, b) << endl; // ??? summe(unsigned long obj1, int  obj2);

    cout << summe<double, int, int>(a, b) << endl; // int summe(double obj1, int  obj2);

    cout << summe<double, long>(a, b) << endl; // ??? summe(double obj1, long obj2);

    return 0;                            // Programm erfolgreich durchlaufen
}
```

Instead of types, integer values can as well be used to specialise a function.

```
1  #include <cstddef>          // Für den Typ size_t
2
3  template <std::size_t N, typename T>
4  void array_init(T (&array)[N], T const &startwert){
5      for(std::size_t i=0; i!=N; ++i)
6          array[i]=startwert;
7  }
```

Practice of Template Basics 1

- Have a look into ***practiceSimpleTemplate.cc***
  e.g. “wget https://wiki.scc.kit.edu/gridkaschool/upload/3/31/PracticeSimpleTemplate.cc ./”
  - Change the global functions in such a way, that they can deal with different types of containers with different type specialisations.
  - Use them!
  - Compile the example with SCons.

```cpp
#include <iostream>
#include <vector>
#include <list>

using namespace std;

/** Return twice the input value.
 */
template <typename T>
T doubleValue (T AValue){
  return 2 * AValue;
}

/** Return triple the input value.
 */
template <typename T>
T tripleValue (T AValue){
  return 3 * AValue;
}

/** A function to apply some function on each element of a cont
 */
template <typename T, typename U>
void applyOnEachElement (T& AContainer, U AFunction(U)){
  for (auto iterator = AContainer.begin(); iterator != AContainer.end(); iterator++){
    *iterator = AFunction(*iterator);
  }
}

/** Prints the content of a container.
 */
template <class T>
void print (T& Acontainer){
  for (auto& value : Acontainer){
    cout << value << ", ";
  }
  cout  << endl;
}
```

```cpp
int main(){
  cout << "Hello World" << endl;

  vector<int>   intVector   = {1, 2, 3, 4, 5};
  list  <float> floatList   = {6.1, 7.2, 8.3, 9.4, 10.5};

  cout << "Printing the vector" << endl;
  print(intVector);
  //The newest compilers can deduce the template type themselves.
  //The following is equivalent to:
//applyOnEachElement< vector<int>, int> (intVector, doubleValue<int>);
  applyOnEachElement (intVector, doubleValue<int>);
  cout << "Printing the vector" << endl;
  print(intVector);

  cout << "Printing the list" << endl;
  print(floatList);
  //If you specify at the start, the type of double Value can be deduced...
  applyOnEachElement<list<float>, float>(floatList, tripleValue);
  cout << "Printing the list" << endl;
  print(floatList);
}
```

tripleValue

Note the power of ***iterators*** and the ***auto*** keyword!

```
./practiceSimpleTemplateProgram
Hello World
Printing the vector
1, 2, 3, 4, 5,
Printing the vector
2, 4, 6, 8, 10,
Printing the list
6.1, 7.2, 8.3, 9.4, 10.5,
Printing the list
18.3, 21.6, 24.9, 28.2, 31.5,
```

Partial Specialisation of class Templates

```cpp
#include <iostream>

using namespace std;
//Good old template
template <typename T>
struct Form {
  Form() {cout << "Form <T>" << endl;}
};

//Specialisation for all double
template<>
struct Form <double> {
  Form() {cout << "Form <double>" << endl;}
};

//Specialisation for all const
template <typename T>
struct Form <T const> {
  Form() {cout << "Form <T const>" << endl;}
};


//Specialisation for all pointers to functions with two parameters
template < typename Result, typename T1, typename T2 >
struct Form< Result(*)(T1, T2) > {
  Form() { cout << "Result (*)(T1, T2)" << endl; }
};
//Further specialisation for functions with a char and a double.
template < typename Result >
struct Form< Result(*)(char, double) > {
  Form() { cout << "Result (*)(char, double)" << endl; }
};

//THIS IS FOR YOU
template <typename T>
struct NumberGetter {
  NumberGetter(T number): m_number(number){}

  int getNumber()
  {
//    "return number"
  }
  private:
  T m_number;
};


int main()
{
  Form<int>             a;
  Form<double>          b;
  Form<const double>    c;

  Form <int (*)(char, double)> d;
  Form <int (*)(int, int)>     e;

  NumberGetter<int> myNumber1 (2);
  cout << myNumber1.getNumber() << endl;

  NumberGetter<int* > myNumber2 (new int(2));
  cout << myNumber2.getNumber()  << endl;

}
```

It is on the Wiki.

We had the case of functions before, where partial specialisation is impossible.

Here we can still be generic and the thing would work for double/double*, too.

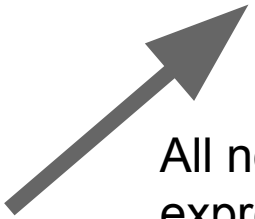
```
36 template <typename T>
37 struct NumberGetter {
38   NumberGetter(T number): m_number(number){}
39
40   int getNumber()
41   {
42     return m_number;
43   }
44   private:
45   T m_number;
46 };
47
48 template<typename T>
49 struct NumberGetter <T*> {
50   NumberGetter(T* number): m_number(number){}
51
52   int getNumber()
53   {
54     return *m_number;
55   }
56   private:
57   T* m_number;
58 };
59
60 int main()
61 {
62   Form<int>             a;
63   Form<double>          b;
64   Form<const double>    c;
65
66   Form <int (*)(char, double)> d;
67   Form <int (*)(int, int)>     e;
68
69   NumberGetter<int> myNumber1 (2);
70   cout << myNumber1.getNumber() << endl;
71
72   NumberGetter<int* > myNumber2 (new int(2));
73   cout << myNumber2.getNumber()  << endl;
74
75 }
```

Policies

- Policies [Introduced in “Modern C++ Design” by Andrei Alexandrescu] are a template based way to implement the Strategy Pattern [Introduced in “Desing Patterns” by Gamma et al.].
    - Let’s first have a look at the Strategy Pattern;

Short Diagram Explanation

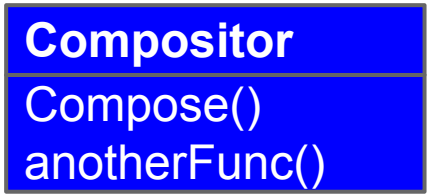
- In short(er than Wikipedia):
  Certain Structures, that solve fairly common problems;
- Often expressed as UML (Universal Modelling Language) Diagram;



All normal arrows express a conceptual reference, e.g. a C++ reference or pointer.

Filled arrows usually express inheritance.

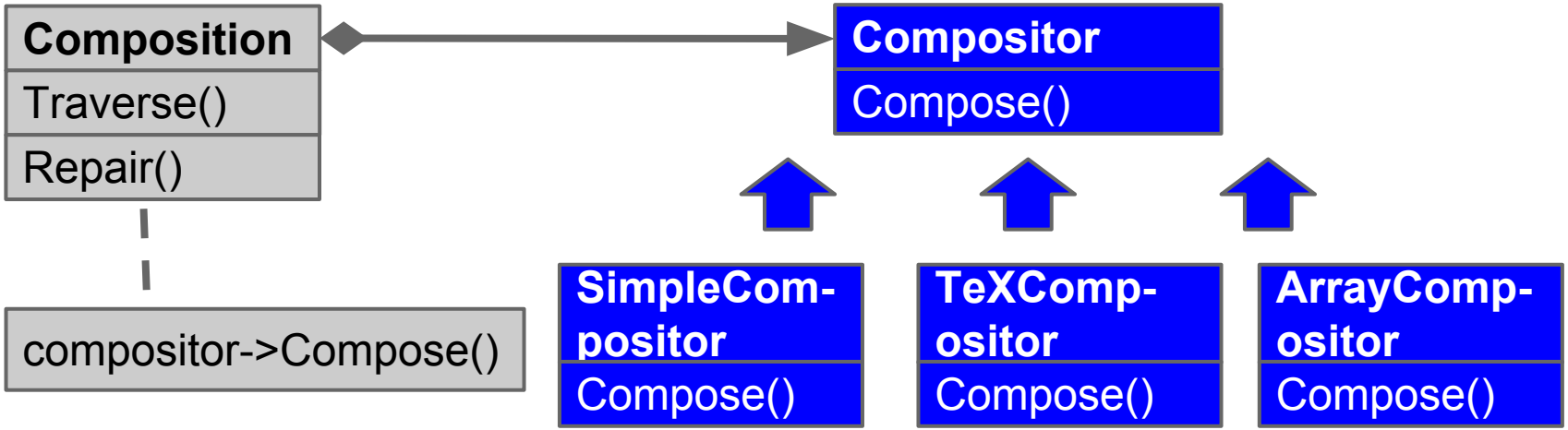
Implementation details.



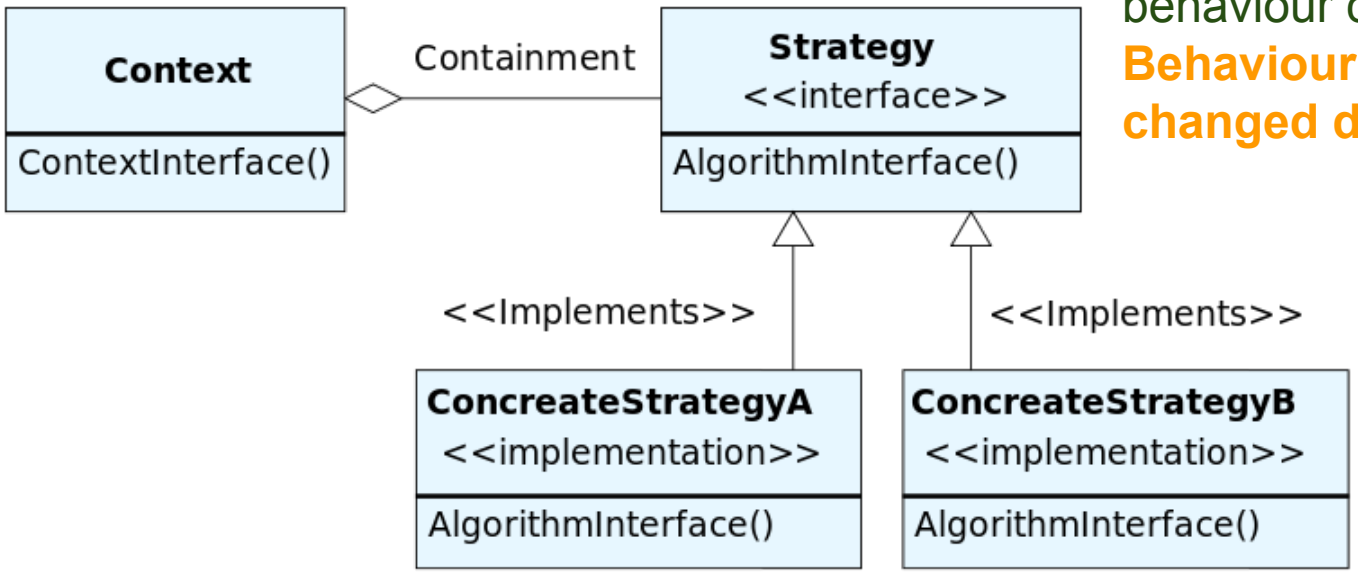
The class name is written bold, below are the important public functions.

Strategy Pattern (using inheritance)

- Assume you want to program an editor, and whenever something is changed, it checks, if it has to recompose line breaks.



Strategy Pattern - Structure



Depending on the type of object, the reference points to, the behaviour changes.

Behaviour can be dynamically changed during running.

When to Use Recommendations

E. Gamma et al.; Design Patterns

- Many related classes differ only in their behaviour (but are called in equal ways).
- You want to employ **Different Variants of an algorithm**, e.g. with different space/time constraints.
- You can **hide data structures** in specific strategy implementations.
- **Multiple conditional statements in a class** --> Move to Strategy class for greater flexibility and better overview.

Consequences

- *You create families of related algorithms*.
  Inheritance from the abstract Strategy may help factor out common behaviour.
- *Alternative to subclassing*.
  Allows dynamic switching of the algorithm, reasonable naming (you need a name for the algorithm, not the class, that uses it), and **no exponential increases of subclasses for combinations of orthogonal behaviour**.
- *Alternative to conditional statements*,
  which make code less easily maintainable, and may lead to branch misprediction etc.

More Consequences

- ***Clients must be aware of different Strategies.***
  A default behaviour is not foreseen in this simple case. Clients need to choose something. Sometimes a reasonable default behaviour may be implemented, when there is no Strategy object at all.
- Strategy interface probably provides ***very generous interface*** with information for all types of concrete strategies. This may produce performance issues. Or the context may pass itself (see below).
- If you have stateless concrete strategy objects, they may be shared among several contexts (and can be flyweights; state information might be storable in the context).

Policies

http://en.wikipedia.org/wiki/Policy-based_design

```
template <typename OutputPolicy, typename LanguagePolicy>
class HelloWorld : private OutputPolicy, private LanguagePolicy
{
    using OutputPolicy::print;
    using LanguagePolicy::message;

public:
    // Behaviour method
    void run() const
    {
        // Two policy methods
        print(message());
    }
};

class OutputPolicyWriteToCout
{
protected:
    template<typename MessageType>
    void print(MessageType const &message) const
    {
        std::cout << message << std::endl;
    }
};
```

- private inheritance means: you inherit the implementation, not the interface;
- has nothing to do with an “is a” relationship;

```
class LanguagePolicyEnglish
{
protected:
    std::string message() const
    {
        return "Hello, World!";
    }
};

class LanguagePolicyGerman
{
protected:
    std::string message() const
    {
        return "Hallo Welt!";
    }
};
```

```cpp
int main()
{
    /* Example 1 */
    typedef HelloWorld<OutputPolicyWriteToCout, LanguagePolicyEnglish> HelloWorldEnglish;

    HelloWorldEnglish hello_world;
    hello_world.run(); // prints "Hello, World!"

    /* Example 2
     * Does the same, but uses another language policy */
    typedef HelloWorld<OutputPolicyWriteToCout, LanguagePolicyGerman> HelloWorldGerman;

    HelloWorldGerman hello_world2;
    hello_world2.run(); // prints "Hallo Welt!"
}
```

- You specify as template parameter from which class you inherit.
- Still more efficient than runtime-polymorphism, as you will usually not have pointers to the base type/ virtual functions etc.

- If the policy is inherited publicly, you can have additional methods.
- As stated before, policies as a strategy-like thing work best, when they are fully orthogonal. If they are not, maybe traits can help you work around…
- Price to pay: Essentially you are creating a new class for each combination of policies, so in that sense it is more like subclassing than the original Strategy, but you are creating the Subclasses with very little effort.

One more policy for the sake of it...

```cpp
#pragma once
/**Template abstraction for 2D Forms using inheritance based policies.*/
template <class T>
class Form : private T{
public:
  /**Constructor fixing the length scale. */
  Form(const float scale): m_scale(scale) {
  }
  /** Implemented using the base class. */
  float getArea()const {
    return T::getArea(m_scale);
  }
  /** Common ... */
  float getScale () const {
    return m_scale;
  }
private:
  float m_scale;
};
```

```cpp
#pragma once
/**Class providing area calculation for circles. */
class CirclePolicy{
protected:
 float getArea(const float radius) const {
    return (3.14 * radius * radius);
  }
};
```

```cpp
#include <iostream>
#include <Shape.h>
#include <CircleUtility.h>
#include <Form.h>
#include <CirclePolicy.h>
int main()
{
  std::cout << "Hello World" << std::endl;
  Shape<CircleUtility> circleShape(1.);
  std::cout << "The area is: " << circleShape.getArea() << std::endl;
  Form<CirclePolicy> circleForm(2.);
  std::cout << "The area is: " << circleForm.getArea() << std::endl;
}
```

If I don't have the private inheritance, I need here a member to have an instance of the object.
If the Policy has no state, I could of course use simply function specifications as shown later.

```cpp
#pragma once
/** Circle Utility Object for the template abstraction of 2D Shapes.*/
class CircleUtility {
public:
  CircleUtility (){}
  /** Calculates the area of the shape;
   *
   *  @param radius Scale parameter for the circle; Would diameter
   *                be the better choice?
   */
  float getArea(const float radius) const{
    return (3.14 * radius * radius);
  }
};
```

```cpp
#pragma once
/** Template abstraction for 2D Shapes. */
template <class T>
class Shape{
public:
  /** Constructor fixing the length-scale of the shape. */
  Shape(const float scale): m_scale(scale), m_specificShape(){
  }
  /** Passes the area calculation to an object of the specified type.*/
  float getArea() const {
    return m_specificShape.getArea(m_scale);
  }
  /** Common function to all shapes implemented in this abstraction.*/
  float getScale() const {
    return m_scale;
  }
private:
  //Alternative to private inheritance: Object Composition
  T m_specificShape;
  float m_scale;
};
```

Remark on a Pitfall using Templates

The compiler parses templates for looking at syntactic correctness early on. In case of possibly misinterpretation, the non-template is assumed, e.g. here value multiplication.

```
1  #include <iostream>
2
3  using namespace std;
4
5  template <typename T>
6  void dependentFunction(const T& t){
7    //This doesn't compile, because compiler thinks
8    //Type... is a value and * is a multiplication
9    //instead of a pointer declaration.
10   T::TypeDependentOnT* typePtr;
11   //Solution: Tell the compiler!
12   //typename T::TypeDependentOnT* typePtr;
13 }
14
15 struct TypeDefiningStruct {
16   typedef int TypeDependentOnT;
17 };
18
19 //----------------------------------------
20 int main()
21 {
22   TypeDefiningStruct myStruct;
23   dependentFunction(myStruct);
24 }
```

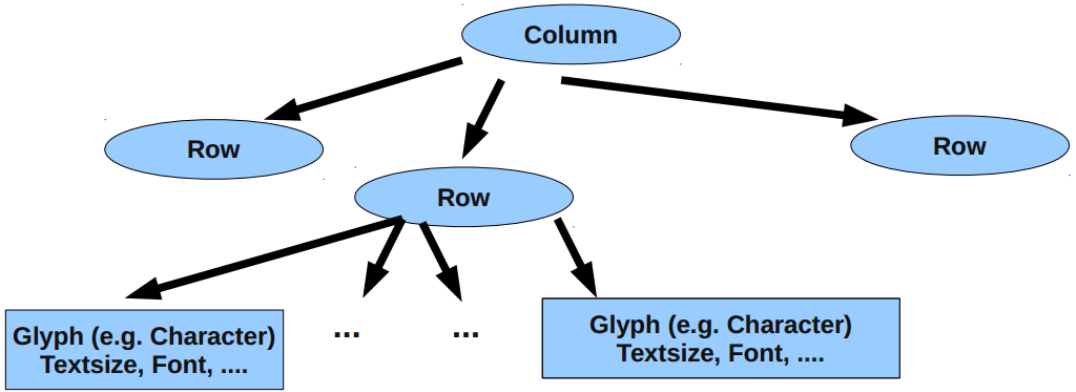
```
g++ -o templatePitFalls.o -c -std=c++11 -ggdb -I. templatePitFalls.cc
templatePitFalls.cc: In function 'void dependentFunction(const T&)':
templatePitFalls.cc:10:24: error: 'typePtr' was not declared in this scope
   T::TypeDependentOnT* typePtr;
                        ^
templatePitFalls.cc: In instantiation of 'void dependentFunction(const T&) [with T = TypeDefiningStruct]':
templatePitFalls.cc:23:29:   required from here
templatePitFalls.cc:10:22: error: dependent-name 'T:: TypeDependentOnT' is parsed as a non-type, but instantiation yields a type
   T::TypeDependentOnT* typePtr;
                      ^
templatePitFalls.cc:10:22: note: say 'typename T:: TypeDependentOnT' if a type is meant
scons: *** [templatePitFalls.o] Error 1
```

For Practice on Policies...

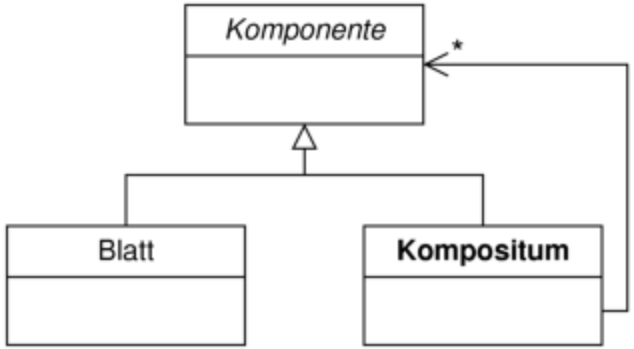
- I would like you to work on a bigger toy project rather than just a few lines of code, so you can have a better feeling on using templates in a more realistic setup.
    - As the this bigger project uses another Design Pattern, I would like to introduce it, the Composite.

Composite

- Tree-like structure, that allows treating individual objects and sub-composites in a similar way.
- E.g. a text editor program may have glyphs, that can again have sub-glyphs.

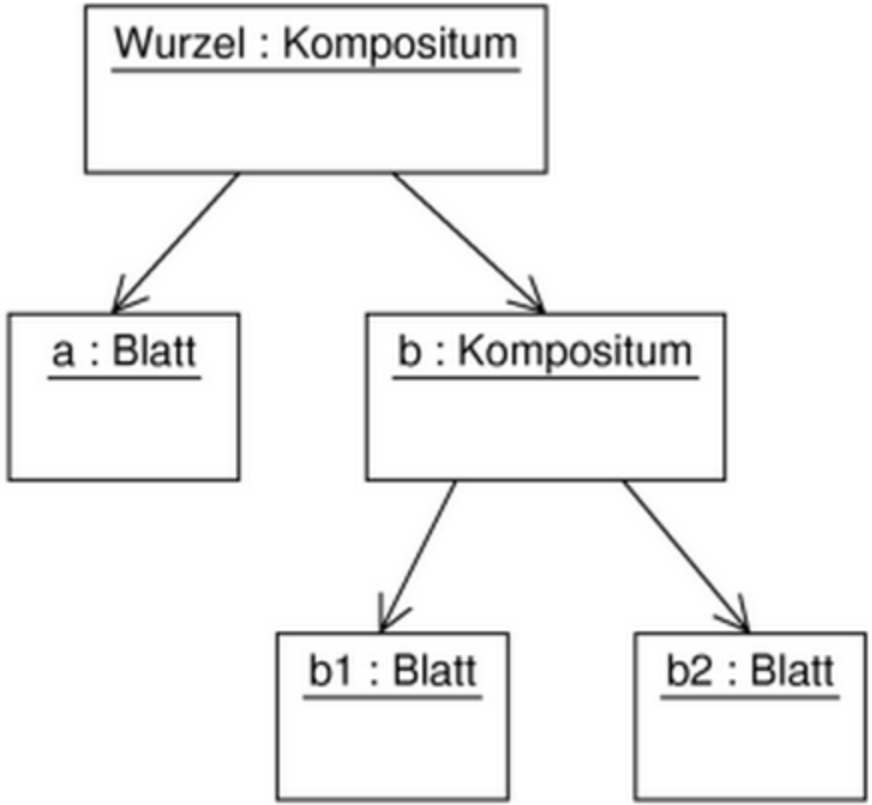


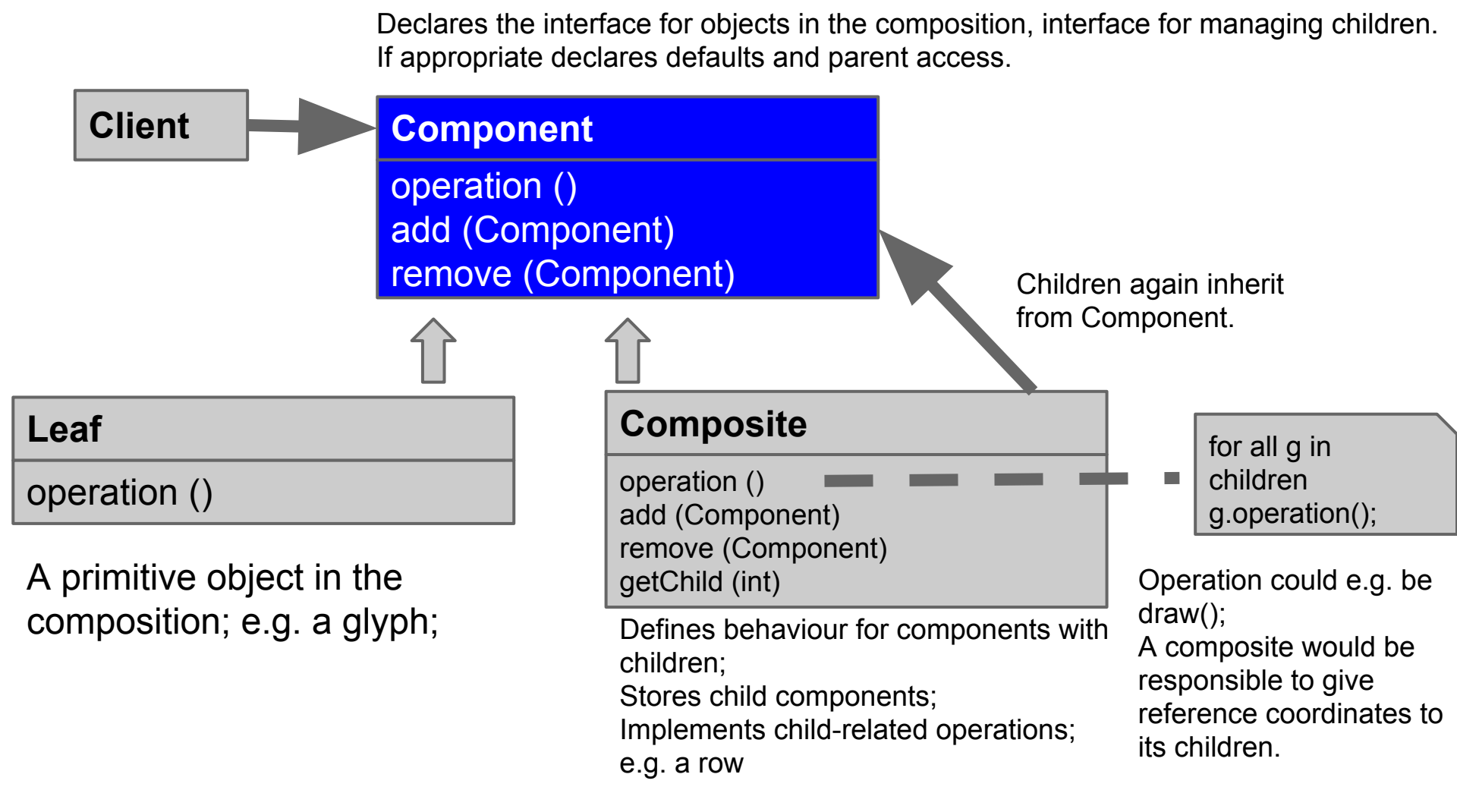
Klassendiagramm [Bearbeiten]



Sometimes the flexibility of possible operations is enhanced, if a child has as well a reference to the parent (or some subclasses have...). This pattern is fairly broad in terms of what additional features might be there.

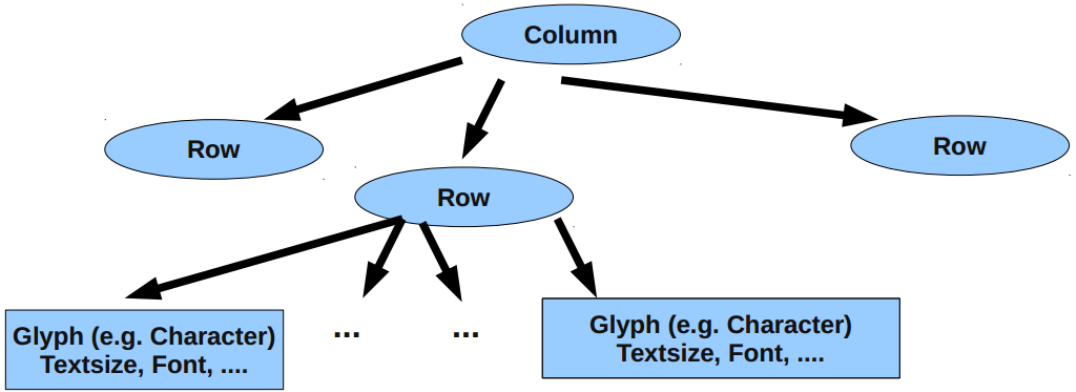
Objektdiagramm [Bearbeiten]




Declares the interface for objects in the composition, interface for managing children.
If appropriate declares defaults and parent access.
Client
Component
operation ()
add (Component)
remove (Component)
Children again inherit from Component.
Leaf
operation ()
A primitive object in the composition; e.g. a glyph;
Composite
operation ()
add (Component)
remove (Component)
getChild (int)
for all g in children
g.operation();
Defines behaviour for components with children;
Stores child components;
Implements child-related operations;
e.g. a row
Operation could e.g. be draw();
A composite would be responsible to give reference coordinates to its children.

Remarks

- It is not strictly necessary, that the leaf handles all operations.
  In this example, e.g. if the document is asked about its length, the document may add the column length, the columns the row lengths, but the row would search for the single largest length.



Consequences

- Composite object and primitive object have a common interface to client requests.
- Simplifies the client, which only operates on the provided component interface.
- Easy to add new kinds of components.
- Difficult to control on what you are running. The composite may contain components you don’t expect.

Practice for Policies

- A little project to be able to test the practicability of policies in a “real” situation.
  - Try first to make the WidthStrategy into a policy.
  - Next make the ColourStrategy into a policy.
  - If you think your construction is very ugly now, try to make it nicer.

```
int main()
{
  cout << "Creating a composite practicePolicyProject!" << endl;

  // Config info:
  unsigned xSize = 100;
  std::vector<unsigned> nLayers = {8, 6, 6, 6, 6};

  //Building of the Composite.
  CompositeChamber compositeChamber(xSize);
  for (int ii = 0; ii < nLayers.size(); ii++) {
    SuperLayer* superLayer = 0;
    if (ii % 2) {
      superLayer = compositeChamber.addSuperLayer(new ColourStrategyBlue());
    } else {
      superLayer = compositeChamber.addSuperLayer(new ColourStrategyRed());
    }

    for (int jj = 0; jj < nLayers[ii]; ++jj) {
      if (jj < nLayers[ii] / 2) {
        superLayer->addLayer(new WidthStrategyOne());
      } else {
        superLayer->addLayer(new WidthStrategyTwo());
      }
    }
  }
  compositeChamber.fillCells();
```

```
wc *
  21    55   424 BackgroundVisitor.h
   7    10   147 Cell.cc
  67   205  1602 Cell.h
  79   198  1872 ChamberComponent.cc
 112   280  2303 ChamberComponent.h
  16    23   230 ChamberIterator.cc
  34   102   737 ChamberIterator.h
  21    44   393 ColourStrategy.h
  66   189  1620 CompositeChamber.h
  50   150  1307 Layer.h
  46   164   964 Particle.h
  83   277  2536 practicePolicy.cc
   5    16   146 SConstruct
  52   122  1279 SpecialCell.h
  36   128  1109 SuperLayer.h
  53   119  1039 WidthStrategy.h
 748  2082 17708 total
```

```
// Config info.
unsigned xSize = 100;
std::vector<unsigned> nLayers = {8, 6, 6, 6, 6};

//Building of the Composite.
CompositeChamber compositeChamber(xSize);
for (int ii = 0; ii < nLayers.size(); ii++) {
  SuperLayer* superLayer = 0;
  if (ii % 2) {
    superLayer = compositeChamber.addSuperLayer(new ColourStrategyBlue())
  } else {
    superLayer = compositeChamber.addSuperLayer(new ColourStrategyRed())
  }

  for (int jj = 0; jj < nLayers[ii]; ++jj) {
    if (jj < nLayers[ii] / 2) {
      superLayer->addLayer<WidthStrategyOne>();
    } else {
      superLayer->addLayer<WidthStrategyOne>();
    }
  }
}
compositeChamber.fillCells();
```

```
class Layer : public ChamberComponent, private WidthType {
  using WidthType::getWidth;
public:
  Layer(ChamberComponent* parent, const ColourStrategy* colourStrategy) :
    ChamberComponent(parent),
    m_colourStrategy(colourStrategy)
  {
  }

  void fillCells() override
  {
    unsigned width = getWidth();
    for (int ii = m_children.size() * width; ii < getMaxX(); ii += width) {
      m_children.emplace_back(new SpecialCell<WidthType>(ii, getMyY(), this, m_col
```

```
template <typename WidthType>
class SpecialCell : public Cell, private WidthType {
  using WidthType::getWidth;
  using WidthType::getEmptyDisplayElement;
  using WidthType::getHitDisplayElement;
public:
  /** Constructor with position and strategies.
   *
   *  Note that in the current version
   */
  SpecialCell(unsigned xPosition, unsigned yPosition,
              ChamberComponent* parent,
              const ColourStrategy* colourStrategy
             ) :
    Cell(xPosition, yPosition, parent),
    m_colourStrategy(colourStrategy)
  {}

  bool xIsInside(unsigned x) override
  {
    if (x >=  m_xPosition && x <= m_xPosition + getWidth()) {
      return true;
    }
    return false;
  }

  unsigned getMaxY() const override
  {
    return 0;
  }

  std::string visualize() override
  {
    std::string returnString = m_colourStrategy->getColourString();
    if (m_eDepositionCount == 0) {
      returnString += getEmptyDisplayElement();
    } else {
      returnString += getHitDisplayElement();
    }
    return (returnString += "\x1B[0m");
  }
```

```
// Config info:
unsigned xSize = 100;
std::vector<unsigned> nLayers = {8, 6, 6, 6, 6};

//Building of the Composite.
CompositeChamber compositeChamber(xSize);
for (int ii = 0; ii < nLayers.size(); ii++) {
  if (ii % 2) {
    auto superLayer = compositeChamber.addSuperLayer<ColourStrategyBlue>();

    for (int jj = 0; jj < nLayers[ii]; ++jj) {
      if (jj < nLayers[ii] / 2) {
        superLayer->addLayer<WidthStrategyOne>();
      } else {
        superLayer->addLayer<WidthStrategyOne>();
      }
    }

  } else {
    auto superLayer = compositeChamber.addSuperLayer<ColourStrategyRed>();

    for (int jj = 0; jj < nLayers[ii]; ++jj) {
      if (jj < nLayers[ii] / 2) {
        superLayer->addLayer<WidthStrategyOne>();
      } else {
        superLayer->addLayer<WidthStrategyOne>();
      }
    }

  }

}
compositeChamber.fillCells();
```

```
template <typename SuperLayerType>
void fillWithAlternatingWidths(SuperLayerType superLayer, unsigned nLayers)
{
  for (int jj = 0; jj < nLayers; ++jj) {
    if (jj % 2) {
      //removing the keyword template here can cause problems with the compiler trying to
      //interprete the < and > as less than/ greater than symbols.
      superLayer->template addLayer<WidthStrategyOne>();
    } else {
      superLayer->template addLayer<WidthStrategyTwo>();
    }
  }
}

int main()
{
  cout << "Creating a composite practicePolicyProject!" << endl;

  // Config info:
  unsigned xSize = 100;
  std::vector<unsigned> nLayers = {8, 6, 6, 6, 6};

  //Building of the Composite.
  CompositeChamber compositeChamber(xSize);
  for (int ii = 0; ii < nLayers.size(); ii++) {
    if (ii % 2) {
      auto superLayer = compositeChamber.addSuperLayer<ColourStrategyBlue>();
      fillWithAlternatingWidths(superLayer, nLayers[ii]);
    } else {
      auto superLayer = compositeChamber.addSuperLayer<ColourStrategyRed>();
      fillWithAlternatingWidths(superLayer, nLayers[ii]);
    }

  }
  compositeChamber.fillCells();
```

What have we learned from the project?

- It was only easy to change things, because SpecialCell inherits from Cell, which is the interface for most of the algorithms.
- Is it better to use Strategy or Policy here?
  - I think the Strategy is easier to read.
  - Theoretically there is a minor speed improvement.
  - Policies prevent the detector being changed during runtime.

  In total I think here it is mostly a matter of taste.

Some Usage Examples

- Let’s have a look into some examples, just so you see people use templates in HEP software.

From basf2: RelationObject

- Use templates to avoid diamond inheritance, for objects, that should inherit from RelationsObject (interfacing to relations) and some other object that already inherits from TObject

```
template <class BASE> class RelationsInterface: public BASE {
public:

  /** Default constructor.
   */
  RelationsInterface(): m_cacheDataStoreEntry(NULL), m_cacheArrayIndex(-1) {}
#if defined(__CINT__) || defined(__ROOTCLING__) || defined(R__DICTIONARY_FILENAME)
#else
  /** Constructor, forwards all arguments to BASE constructor. */
  template<class ...Args> explicit RelationsInterface(Args&& ... params):
    BASE(std::forward<Args>(params)...),
    m_cacheDataStoreEntry(NULL), m_cacheArrayIndex(-1) { }
#endif

  /** Copy constructor.
   *
   *  Cached values are cleared.
   *  @param relationsInterface  The object that should be copied.
   */
  RelationsInterface(const RelationsInterface& relationsInterface):
    BASE(relationsInterface),
    m_cacheDataStoreEntry(NULL), m_cacheArrayIndex(-1) { }

  /** Assignment operator.
   *
   *  cached values of 'this' are not touched, since position does not change.
   *  @param relationsInterface  The object that should be assigned.
   */
  RelationsInterface& operator=(const RelationsInterface& relationsInterface)
  {
    if (this != &relationsInterface)
      this->BASE::operator=(relationsInterface);
    return *this;
  }


  /** Add a relation from this object to another object (with caching).
   *
   *  @param object  The object to which the relation should point.
   *  @param weight  The weight of the relation.
   */
  void addRelationTo(const RelationsInterface<BASE>* object, double weight = 1.0) const
  {
    if (object)
      DataStore::Instance().addRelation(this, m_cacheDataStoreEntry, m_cacheArrayIndex,
                                        object, object->m_cacheDataStoreEntry, object->m_cacheArrayIndex, weight);
  }
```

Use template for automatic name deduction, check equality of type in I/O with expected type...

```
template <class T>
class StoreArray : public StoreAccessorBase {
public:
  /** STL-like iterator over the T objects (not T* ). */
  typedef ArrayIterator<StoreArray<T>, T> iterator;
  /** STL-like const_iterator over the T objects (not T* ). */
  typedef ArrayIterator<StoreArray<T>, const T> const_iterator;

  /** Register an array, that should be written to the output by default, in the data store.
   *  This must be called in the initialization phase.
   *
   *  @param name        Name under which the TClonesArray is stored.
   *  @param durability  Specifies lifetime of array in question.
   *  @param errorIfExisting  Flag whether an error will be reported if the array was already registered.
   *  @return            True if the registration succeeded.
   */
  static bool registerPersistent(const std::string& name = "", DataStore::EDurability durability = DataStore::c_Event,
                                 bool errorIfExisting = false)
  {
    return DataStore::Instance().registerEntry(DataStore::arrayName<T>(name), durability, T::Class(), true,
                                               errorIfExisting ? DataStore::c_ErrorIfAlreadyRegistered : 0);

  }
```

```cpp
#pragma once

#include <tracking/trackFindingCDC/legendre/quadtree/QuadTreeItem.h>
#include <tracking/trackFindingCDC/legendre/quadtree/CDCLegendreQuadTree.h>

namespace Belle2 {
  namespace TrackFindingCDC {

    /**
     * This abstract class serves as a base class for all implementations of track processors.
     * It provides some functions to create, fill, clear and postprocess a quad tree.
     * If you want to use your own class as a quad tree item, you have to overload this processor (not the quad tree itself as it is templated).
     * You have provide only the two functions insertItemInNode and createChildWithParent.
     */
    template<typename typeX, typename typeY, class typeData, int binCountX, int binCountY>
    class QuadTreeProcessorTemplate {

    public:
      typedef QuadTreeItem<typeData> ItemType; /**< The QuadTree will only see items of this type */
      typedef std::vector<typeData*> ReturnList; /**< The type of the list of result items returned to the lambda function */
      typedef QuadTreeTemplate<typeX, typeY, ItemType, binCountX, binCountY> QuadTree;  /**< The used QuadTree */
      typedef std::function< void(const ReturnList&, QuadTree*) >
      CandidateProcessorLambda;   /**< This lambda function can be used for postprocessing */
      typedef typename QuadTree::Children QuadTreeChildren; /**< A typedef for the QuadTree Children */

      typedef std::pair<typeX, typeX> rangeX;   /**< This pair describes the range in X for a node */
      typedef std::pair<typeY, typeY> rangeY;   /**< This pair describes the range in Y for a node */
      typedef std::pair<rangeX, rangeY> ChildRanges; /**< This pair of ranges describes the range of a node */

      friend QuadTree;

    public:
      /**
       * Constructor is very simple. The QuadTree has to be constructed elsewhere.
       * @param lastLevel describing the last search level for the quad tree creation.
       */
      QuadTreeProcessorTemplate(unsigned char lastLevel, const ChildRanges& ranges, bool debugOutput = false, bool setUsedFlag = true) :
        m_lastLevel(lastLevel), m_debugOutput(debugOutput), m_debugOutputMap(), m_param_setUsedFlag(setUsedFlag)
      {
        createQuadTree(ranges);
      }
```

Algorithms, that need to run very fast.

Objects for generic algorithms

Possibilities, not necessarily recommendations

```
1 #include <iostream>
2
3 using namespace std;
4
5 //Abstract Form, saving only the position.
6 class Form {
7 public:
8   Form(float positionX):
9     m_positionX(positionX)
10  {}
11
12  float getPositionX()const
13  {
14    return m_positionX;
15  }
16 private:
17  float m_positionX;
18 };
19
20 template <float T()>
21 //Class taking form (e.g. circle etc.) specific members.
22 class SpecialForm {
23 public:
24  SpecialForm(): m_form(new Form(10.)) {}
25  ~SpecialForm() {delete m_form;}
26
27  //The area calculation depends on the specific form chosen.
28  float calculateArea()const
29  {
30    return T();
31  }
32
33  //Make the object act as a pointer to itself.
34  //I don't recommend this, but you should have seen it.
35  SpecialForm const* operator->() const
36  {return this;}
37
38  //Make the object "auto casting" to Form in certain circumstances.
39  operator const Form* () const
40  {
41    return m_form;
42  }
43
44 private:
45  Form* m_form;
46 };
47
48 //Area calculation for circles.
49 float circleArea()
50 {
51  return 3.14;
52 }
53
54 //just a function to take a Form pointer.
55 void doSomethingWithForm(Form const* form)
56 {
57  cout << form->getPositionX() << endl;
58 }
59
60 //----------------------------------------
61 int main()
62 {
63  SpecialForm<circleArea> specialFormCircle;
64  //Test, that object indeed acts as pointer to itself;
65  cout << specialFormCircle. calculateArea() << endl;
66  cout << specialFormCircle->calculateArea() << endl;
67
68  //Test, that I can use SpecialForm in functions, where Pointer to Form is asked;
69  doSomethingWithForm(specialFormCircle);
70 }
```

Avoiding Inheritance

- To avoid inheritance (and that is part of the usage of templates), you may use interface forwarding, e.g.
  struct Foo {
  get_size (){return bar.get_size()}
  private:
  vector<> bar
  }

```
#pragma once
#include <iostream>
#include <vector>

template <int key, int... constants>
class VectorOfConstants
{
public:
  VectorOfConstants(){
    m_vector.reserve(sizeof... constants);
    int temp[] = {constants...};
    for (int& ii : temp) {
      m_vector.push_back(ii);
    }
  }
  void print() const {
    for (const int& ii : m_vector){
      std::cout << "Number is: " << ii << std::endl;
    }
  }
private:
  std::vector<int> m_vector;
};
```

Elipsis operator is part of C++11

```
./main2
Hello Welt
Number is: 3
Number is: 4
Number is: 5
Number is: 65
Number is: 87
```

Templates can be used as well to instantiate something with a specific value, though this is uncommon.

```
#include <iostream>
#include <VectorOfConstants.h>
using namespace std;
int main ()
{
  cout << "Hello Welt"  << endl;
  VectorOfConstants<42, 3, 4, 5, 65, 87> vectorOfConstants;
  vectorOfConstants.print();
}
```

```
void printf(const char *s)
{
    while (*s) {
        if (*s == '%') {
            if (*(s + 1) == '%') {
                ++s;
            }
            else {
                throw std::runtime_error("invalid format string: missing arguments");
            }
        }
        std::cout << *s++;
    }
}

template<typename T, typename... Args>
void printf(const char *s, T value, Args... args)
{
    while (*s) {
        if (*s == '%') {
            if (*(s + 1) == '%') {
                ++s;
            }
            else {
                std::cout << value;
                s += 2;
                printf(s, args...); // call even when *s == 0 to detect extra arguments
                return;
            }
        }
        std::cout << *s++;
    }
}
```

from wikipedia

Proof of Reality Usability

Helper for conversion of Python objects into C++ objects in the basf2 Framework

```
template< typename T, typename... Types> struct VariadicType;
/** Recursively convert multiple types to type names (used for tuples). */
template< typename T> struct VariadicType<T> { /** type name. */ static std::string name() { return Type<T>::name(); } };
/** Recursively convert multiple types to type names (used for tuples). */
template< typename T, typename... Types> struct VariadicType {
  /** type name. */
  static std::string name()
  {
    return Type<T>::name() + ", " + VariadicType<Types...>::name();
  }
};
```