STL Algorithms & Functors

BugSquashing Seminar

dkfz.
DEUTSCHES
KREBSFORSCHUNGSZENTRUM
IN DER HELMHOLTZ-GEMEINSCHAFT

std::generate

```
template <class ForwardIterator, class Generator>
    void generate (ForwardIterator first, ForwardIterator last, Generator gen);
```

- generator = function pointer or function object
  - for example use the std::rand()

```
std::vector<int> v(5);
std::generate(v.begin(), v.end(), std::rand);
```

  - another version `std::generate_n`
    - no need for last position, n steps from start
  - also to be used with own functions, return type must match the vector<T> type

```
std::generate_n(v.begin(), count, Fibonacci);
```

```
int Fibonacci(void)
{
    static int r;
    static int f1 = 0;
    static int f2 = 1;
    r = f1 + f2 ;
    f1 = f2 ;
    f2 = r ;
    return f1 ;
}
```

(STL) Functors

- **Functor** = function object
  - *functions*, *function pointers*, generally: an object defining `operator()`

- **Generators**, **Unary** and **Binary Operators** `f(), f(x), f(x,y)`

- special case with return type `bool` : **Predicate**
- for use with templates : **Adaptable Functors**

```
struct less_mag : public binary_function<double, double, bool> {
  bool operator()(double x, double y) { return fabs(x) < fabs(y); }
   };
```

- STL offers predefined function objects
  - `plus, minus, multiplies, divides ...`
  - `equal_to, not_equal_to, greater, greater_equal ...`
  - `logical_and, logical_or, logical_not`

std::generate – Example 2

- **Example:** generate a set of 3D Points covering a regular grid and store them in a vector
- **Solution:** use own class **`AngleGridPoint`**

```
struct AngleGridPoint
{
    itk::Point<double> operator()() { [...] }
};

// create the generator object
AngleGridPoint gen_coarse_grid( minAngle, maxAngle, angleStep );

// declare the vector and generate
std::vector < itk::Point<double> > CoarseGrid ( coarseGridSize );
std::generate( CoarseGrid.begin(), CoarseGrid.end(), gen_coarse_grid );
```

std::transform

```
template <class InputIterator, class OutputIterator, class UnaryOperation>
  OutputIterator transform (InputIterator first1, InputIterator last1,
                            OutputIterator result, UnaryOperation op);
```

- applies the operation specified to the input and stores them into the result

- UnaryOperation signature : `T_out some_function( T_in value )`
- BinaryOperation signature : `T_out some_function( T_in value1, T_in value2)`

```
// foo: 10 20 30 40 50

std::transform (foo.begin(), foo.end(), bar.begin(), op_increase);
// bar: 11 21 31 41 51

std::transform (foo.begin(), foo.end(), bar.begin(), foo.begin(),
                std::plus<int>());
// foo: 21 41 61 81 101
```

std::transform – Example 2

- for special tasks – define own class, the operator() will be used to access the input

```
EvaluateMetricOnTranslation<NCorrMetricType> evaluateMetricFunctor( metric)
std::vector<double> metric_values( dx_values.size() );

std::transform(  dx_values.begin(), dx_values.end(), metric_values.begin(),
 evaluateMetricFunctor )
```

```
template <typename MetricType>
struct EvaluateTranslation{
[...]
double operator()( double delta)
  {
    MetricType::TransformParametersType
         initialParams( 3 );

    initialParams.Fill(0);
    initialParams[2] = delta;

    m_Metric->Initialize();
    return m_Metric->GetValue( initialParams );
  }
[...]};
```