9. C++ advanced (II): Templates

Motivation

Goal: generic vector class and functionality.

Examples

```
vector<double> vd(10);
vector<int> vi(10);
vector<char> vi(20);

auto nd = vd * vd; // norm (vector of double)
auto ni = vi * vi; // norm (vector of int)
```

Types as Template Parameters

1. In the concrete implementation of a class replace the type that should become generic (in our example: `double`) by a representative element, e.g. `T`.
2. Put in front of the class the construct `template<typename T>`[10] Replace `T` by the representative name).

The construct `template<typename T>` can be understood as "for all types `T`".

[10] equally: `template<class T>`

Types as Template Parameters

```
template <typename ElementType>
class vector{
    size_t size;
    ElementType* elem;
public:
    ...
    vector(size_t s):
        size{s},
        elem{new ElementType[s]}{}
    ...
    ElementType& operator[](size_t pos){
        return elem[pos];
    }
    ...
}
```

Template Instances

`vector<typeName>` generates a type instance `vector` with `ElementType=typeName`.

Notation: Instantiation

Examples

```
vector<double> x;          // vector of double
vector<int> y;             // vector of int
vector<vector<double>> x; // vector of vector of double
```

Type-checking

Templates are basically replacement rules at instantiation time and applied compilation. It is checked as little as necessary and as much as possible.

Example

```
template <typename T>
class vector{
...
  // pre: vector contains at least one element, elements comparable
  // post: return minimum of contained elements
  T min() const{
    auto min = elem[0];
    for (auto x=elem+1; x<elem+size; ++x){
      if (*x<min) min = *x;
    }
    return min;
  }
...
}
```

Example

```
template <typename T>
class vector{
...
  // pre: vector contains at least one element, elements comparable
  // post: return minimum of contained elements
  T min() const{
    auto min = elem[0];
    for (auto x=elem+1; x<elem+size; ++x){
      if (*x<min) min = *x;
    }
    return min;
  }
...
}
```

```
vector<int> a(10); // ok
auto m = a.min(); // ok
vector<vector<int>> b(10); // ok;
auto n = b.min(); no match for operator< !
```

Generic Programming

Generic components should be developed rather as a generalization of one or more examples than from first principles.

```
using size_t=std::size_t;
template <typename T>
class vector{
public:
  vector();
  vector(size_t s);
  ~vector();
  vector(const vector &v);
  vector& operator=(const vector&v);
  vector (vector&& v);
  vector& operator=(vector&& v);
  T operator[] (size_t pos) const;
  T& operator[] (size_t pos);
  int length() const;
  T* begin();
  T* end();
  const T* begin() const;
  const T* end() const;
}
```

Function Templates

1. In a concrete implementation of a function replace the type that should become generic by a replacement, .e.g `T`,
2. Put in front of the function the construct `template<typename T>`[11](Replace `T` by the replacement name)

[11]equally:`template<class T>`

Function Templates

```
template <typename T>
void swap(T& x, T&y){
    T temp = x;
    x = y;
    y = temp;
}
```

Types of the parameter determine the version of the function that is (compiled) and used:

```
int x=5;
int y=6;
swap(x,y); // calls swap with T=int
```

Limits of Magic

```
template <typename T>
void swap(T& x, T&y){
    T temp = x;
    x = y;
    y = temp;
}
```

An inadmissible version of the function is not generated:

```
int x=5;
double y=6;
swap(x,y); // error:  no matching function for ...
```

Limits of Magic

Separation of declaration and definition is possible ...

```
template <typename T>
class Pair{
    T left; T right;
public:
    Pair(T l, T r):left{l}, right{r}{}
    T Min();
    //...
};

template <typename T>
T Pair<T>::Min(){
    return left < right ? left : right;
}
```

Limits of Magic

Hiding implementations common in OOP is limited. The definition cannot be provided in separate, non-included file.

```
template <typename T>
class Pair{
    T left; T right;
public:
    Pair(T l, T r):left{l}, right{r}{}
    T Min();
    //...
};

template <typename T> // cannot be hidden from the user of Pair !
T Pair<T>::Min(){
    return left < right ? left : right;
}
```

Useful!

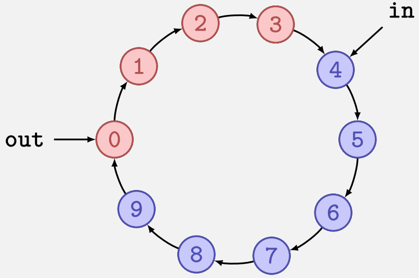
```
// Output of an arbitrary container
template <typename T>
void output(const T& t){
    for (auto x: t)
        std::cout << x << " ";
    std::cout << "\n";
}

int main(){
  std::vector<int> v={1,2,3};
  output(v); // 1 2 3
}
```

Powerful!

```
template <typename T> // square number
T sq(T x){
    return x*x;
}
template <typename Container, typename F>
void apply(Container& c, F f){ // x <- f(x) forall x in c
    for(auto& x: c)
        x = f(x);
}
int main(){
  std::vector<int> v={1,2,3};
  apply(v,sq<int>);
  output(v); // 1 4 9
}
```

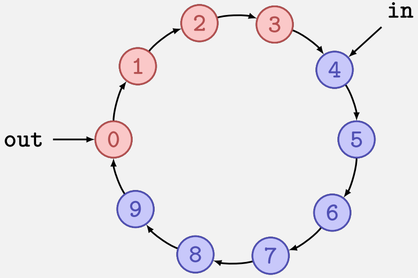
Template Parameterization with Values

```
template <typename T, int size>
class CircularBuffer{
  T buf[size] ;
  int in; int out;
```



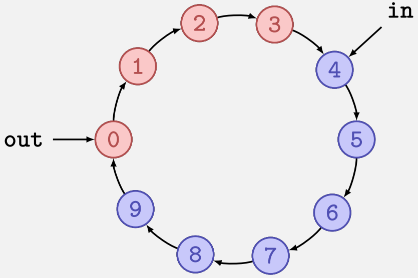
Template Parameterization with Values

```
template <typename T, int size>
class CircularBuffer{
  T buf[size] ;
  int in; int out;
public:
  CircularBuffer():in{0},out{0}{};
  bool empty(){
    return in == out;
  }
  bool full(){
    return (in + 1) % size == out;
  }
```



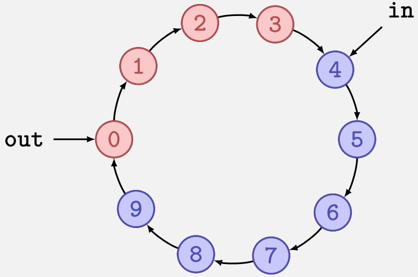
Template Parameterization with Values

```
template <typename T, int size>
class CircularBuffer{
  T buf[size] ;
  int in; int out;
public:
  CircularBuffer():in{0},out{0}{};
  bool empty(){
    return in == out;
  }
  bool full(){
    return (in + 1) % size == out;
  }
  void put(T x); // declaration
  T get();        // declaration
};
```



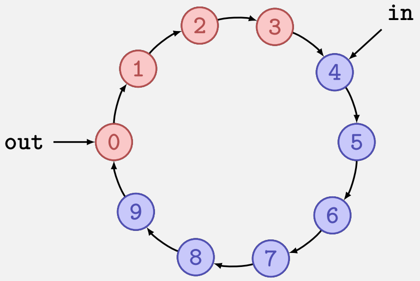
Template Parameterization with Values

```
template <typename T, int size>
void CircularBuffer<T,size>::put(T x){
    assert(!full());
    buf[in] = x;
    in = (in + 1) % size;
}
```



Template Parameterization with Values

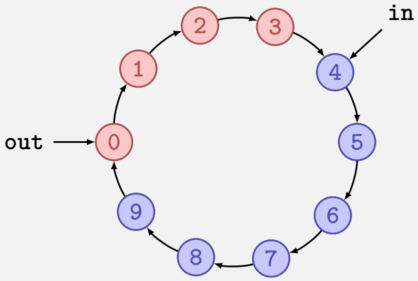
```
template <typename T, int size>
void CircularBuffer<T,size>::put(T x){
    assert(!full());
    buf[in] = x;
    in = (in + 1) % size;
}

template <typename T, int size>
T CircularBuffer<T,size>::get(){
    assert(!empty());
    T x = buf[out];
    out = (out + 1) % size;
    return x;
}
```



Template Parameterization with Values

```
template <typename T, int size>
void CircularBuffer<T,size>::put(T x){
    assert(!full());
    buf[in] = x;
    in = (in + 1) % size;
}

template <typename T, int size>
T CircularBuffer<T,size>::get(){
    assert(!empty());
    T x = buf[out];
    out = (out + 1) % size;
    return x;
}
```

$\longleftarrow$ Potential for optimization if size $= 2^k$.


in
out
0
1
2
3
4
5
6
7
8
9