10. C++ advanced (II): Templates

What do we learn today?

- templates of classes
- function templates
- Smart Pointers

Motivation

Goal: generic vector class and functionality.

```
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>` Replace `T` by the representative name).

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

Types as Template Parameters

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

Template Instances

`Vector<typeName>` generates a type instance `Vector` with
`ElementType=typeName`.
Notation: Instantiation

```
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 during compilation. The compiler always checks as little as necessary and as much as possible.

Example

```cpp
template <typename T>
class Pair{
   T left; T right;
public:
   Pair(T l, T r):left{l}, right{r}{}
   T min(){
     return left < right ? left : right;
   }
};
```

Example

```cpp
template <typename T>
class Pair{
    T left; T right;
public:
    Pair(T l, T r):left{l}, right{r}{}
    T min(){
      return left < right ? left : right;
    }
};

Pair<int> a(10,20); // ok
auto m = a.min(); // ok
Pair<Pair<int>> b(a,Pair<int>(20,30)); // 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.

```cpp
template <typename T>
class Vector{
public:
  Vector();
  Vector(std::size_t);
  ~Vector();
  Vector(const Vector&);
  Vector& operator=(const Vector&);
  Vector (Vector&&);
  Vector& operator=(Vector&&);
  const T& operator[] (std::size_t) const;
  T& operator[] (std::size_t);
  std::size_t size() const;
  T* begin();
  T* end();
  const T* begin() const;
  const T* end() const;
}
```

Function Templates

1. To make a concrete implementation generic, replace the specific type (e.g. int) with a name, e.g. `T`,
2. Put in front of the function the construct `template<typename T>` (Replace `T` by the chosen name)

Function Templates

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

The actual parameters' types 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
```

Safety

```cpp
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:

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

.. also with operators

```cpp
template <typename T>
class Pair{
    T left; T right;
public:
    Pair(T l, T r):left{l}, right{r}{}
    T min(){ return left < right? left: right; }
    std::ostream& print (std::ostream& os) const{
        return os << "("<< left << "," << right<< ")";
    }
};

template <typename T>
std::ostream& operator<< (std::ostream& os, const Pair<T>& pair){
    return pair.print(os);
}
```

.. also with operators

```cpp
template <typename T>
class Pair{
    T left; T right;
public:
    Pair(T l, T r):left{l}, right{r}{}
    T min(){ return left < right? left: right; }
    std::ostream& print (std::ostream& os) const{
        return os << "("<< left << "," << right<< ")";
    }
};

template <typename T>
std::ostream& operator<< (std::ostream& os, const Pair<T>& pair){
    return pair.print(os);
}
```

```cpp
Pair<int> a(10,20); // ok
std::cout << a; // ok
```

Useful!

```cpp
// 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
}
```

Explicit Type

```
// input of an arbitrary pair
template <typename T>
Pair<T> read(){
  T left;
  T right;
  std::cin << left << right;
  return Pair<T>(left,right);
}
...

auto p = read<double>();
```

Explicit Type

```cpp
// input of an arbitrary pair
template <typename T>
Pair<T> read(){
  T left;
  T right;
  std::cin << left << right;
  return Pair<T>(left,right);
}
...

auto p = read<double>();
```

If the type of a template instantiation cannot be inferred, it has to be
provided explicitly.

Powerful!

```cpp
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
}
```
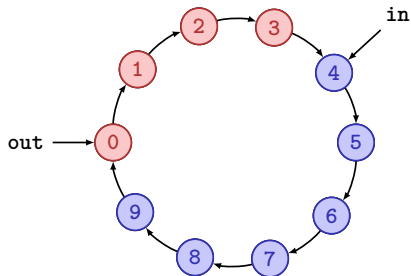
Specialization

```
template <>
class Pair<bool>{
    short both;
public:
    Pair(bool l, bool r):both{(l?1:0) + (r?2:0)} {};
    std::ostream& print (std::ostream& os) const{
        return os << "("<< both % 2 << "," << both /2 << ")";
    }
};
```

Specialization

```cpp
template <>
class Pair<bool>{
    short both;
public:
    Pair(bool l, bool r):both{(l?1:0) + (r?2:0)} {};
    std::ostream& print (std::ostream& os) const{
        return os << "("<< both % 2 << "," << both /2 << ")";
    }
};
    Pair<int> i(10,20); // ok -- generic template
    std::cout << i << std::endl; // (10,20);
    Pair<bool> b(true, false); // ok -- special bool version
    std::cout << b << std::endl; // (1,0)
```
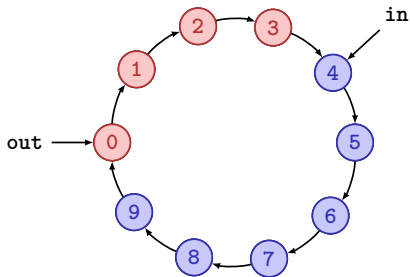
Template Parameterization with Values

```cpp
template <typename T, int size>
class CircularBuffer{
  T buf[size] ;
  int in; int out;
```
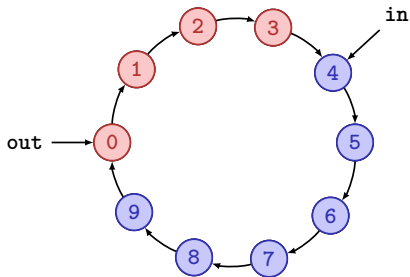
Template Parameterization with Values

```cpp
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;
  }
```
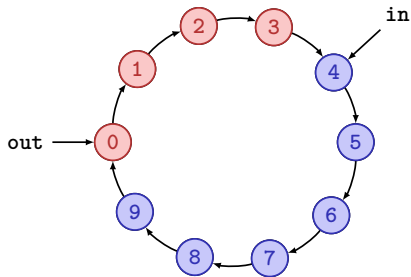
Template Parameterization with Values

```cpp
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
};
```


287

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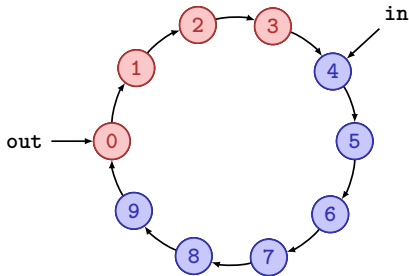
```cpp
template <typename T, int size>
void CircularBuffer<T,size>::put(T x){
    assert(!full());
    buf[in] = x;
    in = (in + 1) % size;
}
```

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;
}
```
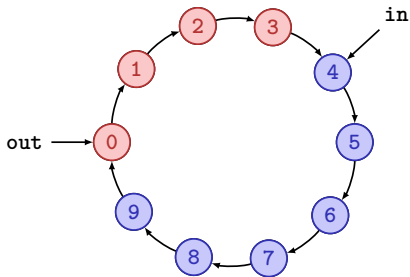
Template Parameterization with Values

```cpp
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; ⟵⟶ Potential for optimization if size = $2^k$.
    return x;
}
```

Memory Management Revisited

Guideline "Dynamic Memory"

For each **new** there is a matching **delete**!

Avoid:
- Memory leaks: old objects that occupy memory
- Pointer to released objects: **dangling pointers**
- Releasing an object more than once using **delete**.

How?

Smart Pointers

- Can make sure that an object is deleted if and only if it is not used any more
- Are based on the RAII (Resouce Acquisition is Initialization) paradigm.
- Can be used instead of a normal pointer: are implemented as class templates.
- There are **std::unique_ptr<>**, **std::shared_ptr<>** (and **std::weak_ptr<>**)

```cpp
std::unique_ptr<Node> nodeU(new Node()); // unique pointer
std::shared_ptr<Node> nodeS(new Node()); // shared pointer
```

Unique Pointer

- The deconstructor of a `std::unique_ptr<T>` deletes the pointer contained.

- `std::unique_ptr<T>` has exclusive ownership for the contained pointer on `T`.

- Copy constructor and assignment operator are deleted. A unique pointer cannot be copied by value. The move constructor is implemented: the pointer can be moved.

- No additional runtime overhead in comparison to a normal pointer

```cpp
std::unique_ptr<Node> nodeU(new Node()); // unique pointer
std::unique_ptr<Node> node2 = std::move(nodeU); // ok
std::unique_ptr<Node> node3 = nodeU; // error
```

Shared Pointer

- **`std::shared_ptr<T>`** Counts the numbers of owners of a pointer (reference count). When reference count goes to 0, the pointer is deleted.

- Shared pointers can be copied.

- Shared pointers provide additional space- and runtime overhead: they manage the reference counter at runtime and contain a pointer to the reference.

`std::`

R

`std::`

Shared Pointer

```
std::shared_ptr<Node> nodeS(new Node()); // shared pointer, rc = 1
std::shared_ptr<Node> node2 = std::move(nodeS); // ok, rc unchanged
std::shared_ptr<Node> node3 = node2; // ok, rc = 2
```

Smart Pointers

Some rules

- Never call **delete** on a pointer contained in a smart pointer.
- Avoid **new**, instead:

  ```
  std::unique_ptr<Node> nodeU = std::make_unique<Node>()
  std::shared_ptr<Node> nodeS = std::make_shared<Node>()
  ```

- Where possible, use **std::unique_ptr**
- If using **std::shared_ptr** make sure there are no cycles in the pointer graph.