# 11. Templates and STL

## 11.1. Motivation: Revisiting the Queue class

The concept of a queue is characterized by its interface and behavior, independent of the type (or class) of elements stored inside.

But: The `Queue` and `UQueue` classes in the previous chapter can only store integers; dealing with other types or objects in the queue would require re-writing the `QueueElement` and `Queue` or `UQueue` classes.

Templates allow implementing a container class independent of the type of data elements.

## 11.1. Motivation: Revisiting the Queue class

Templates are expanded at compile-time, where the compiler does type-checking before the template expansion happens. The source code contains only the function or class, the compiled code can afterwards contain multiple copies of the same function or class. For a function for example:

```
template <class T>  T maxOf(T x, T y) {
  return (x > y)? x : y;
}
int main() {
  std::cout << maxOf<int>(12,24) << ' ';
  std::cout << maxOf<char>('d','e') << '\n';
}
```

```
int maxOf(int x, int y) {
  return (x > y)? x : y;
}
```

```
char maxOf(char x, char y) {
  return (x > y)? x : y;
}
```

## 11.1. Motivation - Template Queue Example 00

example00.cpp

```
template <class Data> class Queue {  // Template class for a queue
 public:
  Queue(int size = 100) : maxSize(size), tail(0), head(0), filled(0) {items = new Data[size];}
  ~Queue() { delete[] items; items = nullptr; };
  void put(Data data);
  Data get();
  bool isFull() const { return filled == maxSize; }
  bool isEmpty() const { return filled == 0; }
  void clear() { filled = 0; head = 0; tail = 0; }  // clear whole queue
 private:
  Data *items;     // array of Data
  int maxSize;    // size of items
  int tail;       // position in array to put
  int head;       // position in array to get from
  int filled;     // number of elements in queue
};
```

## 11.1. Motivation - Template Queue Example 00

example00.cpp

```
// put element at the tail of the queue, for example put(d) updates:
// items:  [  ][  ][  ][  ][  ][ d][  ] … [  ][  ]
//               head                tail→
template <class Data>
void Queue<Data>::put(Data data) {   // put element data at tail
  if (!isFull()) {
    items[tail] = data;
    tail = (tail+1) % maxSize;
    filled++;
  } else {
    throw std::runtime_error("queue: full on put");
  }
}
```

## 11.1. Motivation - Template Queue Example 00

example00.cpp

```
// gets element at the head of the queue, for example get() updates:
// items:  [  ][  ][  ][  ][  ][  ][  ] … [  ][  ]
//                head→             tail
template <class Data>
Data Queue<Data>::get() {  // get and remove element from head
  Data retval;
  if (!isEmpty()) {
    retval = items[head];
    head = (head+1) % maxSize;
    filled--;
  } else {
    throw std::runtime_error("queue: empty on get");
  }
  return retval;
}
```

## 11.1. Motivation - Template Queue Example 00

Queue are used while specifying the data type when creating the queue objects:

example00.cpp

```
int main() {
  Queue<int> intQueue(127);
  Queue<char> charQueue(21);

  // fill int and char data in both queues:
  for (auto i = 1; i <= 9; i++)  intQueue.put(i);
  for (auto i = '1'; i <= '9'; i++)  charQueue.put(i);

  // get earliest data from both queues nine times:
  for (auto i = 0; i < 9; i++) {
    std::cout << intQueue.get() << ' ' << charQueue.get() << '\n';
  }
}
```

# 11.1. Motivation - Template UQueue Example 01

example01.cpp

```
template <class Data> class QueueElement;  // element class, hidden from users

template <class Data> class UQueue {   // Template class for an unlimited queue
 public:
  UQueue() { head = tail = nullptr; }
  ~UQueue() { clear(); };
  void put(Data data);
  Data get();
  bool isEmpty() const;
  void clear();
  Data get();
 private:
  QueueElement<Data> * head;  // pointer to element to put
  QueueElement<Data> * tail;  // pointer to element to get from
};
```

## 11.1. Motivation - Template UQueue Example 01

example01.cpp

```
template <class Data> class QueueElement {  // element class, hidden from users
 public:
  QueueElement(Data data) : data(data) , next(nullptr) {}
  Data data;
  QueueElement<Data> * next;
};

template <class Data>
void UQueue<Data>::clear() {  // iteratively clear the queue of all elements
  QueueElement<Data> * elem, * elem_next;
  for (elem = head; elem != nullptr; elem = elem_next) {
    elem_next = elem->next;
    delete elem;
  }
  head = tail = nullptr;
}
```

## 11.1. Motivation - Template UQueue Example 01

example01.cpp

```
template <class Data>
bool UQueue<Data>::isEmpty() const {  // check whether the queue is empty
  return head == nullptr;
}

template <class Data>
void UQueue<Data>::put(Data data) {  // put in new data element at tail
  QueueElement<Data> * node = new QueueElement<Data>(data);
  if (isEmpty()) {
    head = tail = node;
  } else {
    tail = tail->next = node;  // tail is guaranteed to be valid
  }
}
```

## 11.1. Motivation - Template UQueue Example 01

example01.cpp

```
template <class Data>
Data UQueue<Data>::get() {  // get and remove element from head
  Data retVal;
  if (!isEmpty()) {
    retVal = head->data;
    QueueElement<Data> * second = head->next;
    delete head;
    head = second;
  } else {
    throw std::runtime_error("uqueue: empty on get");
  }
  return retVal;
}
```

## 11.1. Motivation - Template UQueue Example 01

UQueue is used while specifying the data type when creating the queue objects:

example01.cpp

```
int main() {
  Queue<int> intQueue(127);
  Queue<char> charQueue(21);

  // fill int and char data in both queues:
  for (auto i = 1; i <= 9; i++)  intQueue.put(i);
  for (auto i = '1'; i <= '9'; i++)  charQueue.put(i);

  // get earliest data from both queues nine times:
  for (auto i = 0; i < 9; i++) {
    std::cout << intQueue.get() << ' ' << charQueue.get() << '\n';
  }
}
```

## 11.2. Templates

`template <class Data>` designates Data as the parameter. This is a placeholder for a type to be specified later, when an object of the class that follows this declaration of a template. `class` is almost interchangeable with `typename` .

The later use (e.g., instantiation) of a template is often called specialization:
`UQueue<double> q;` or `Queue<Data> * n = new Queue<Data>(250);`
or `int getSize( Queue<int> & q );`

Templates tell the compiler that the class will use a type that is to be specified when the objects are created, which is why templates are *put in the header file*. The compiler will create the actual class only when it is specialized.

## 11.3. Using templates, inheritance, friends

Templates can have more than one parameter:

```
template <class XData, class YData>
```

Templates can have, like functions or methods, default parameters:

```
template <class XData, class YData = char>
```

Templates can have non-type parameters, too:

```
template <class XData, int max>
```

Inheritance is possible:

- Templates can inherit from other templates and from non-template classes
  ```
  template <class x> class SortableQueue<x> : public UQueue<x> {
  ```
- Non-template classes can inherit from templates:
  ```
  class IntQueue : public UQueue<int> {
  ```

## 11.3. Using templates, inheritance, friends

Template parameters for functions can be deduced by the compiler, i.e., without specifying the type. This is possible for template classes since C++17.

example06.cpp

```
#include <iostream>

template <class T>  T maxOf(T x, T y) {
  return (x > y)? x : y;
}

int main() {
  std::cout << maxOf<int>(12,24) << ' ';
  std::cout << maxOf<char>('d','e') << '\n';
  std::cout << maxOf(12,24) << ' ';      // these two lines have the same
  std::cout << maxOf('d','e') << '\n';  // effect as above, types are deduced
}
```

## 11.3. Using templates, inheritance, friends

Template classes can declare a non-template friend class or function, a general template friend class or function, or a type-specific template friend class or function:

```
template <class T> class A {  //
 public:
  template<class U> friend class B;
 private:
  T data;
};
template<class U> class B {
 public:
  B() { A<U> a; std::cout << "A's data: " << a.data << '\n'; };
};

int main() {
  B<int> b;
}
```

## 11.4. Templates and operator overloading

Overloading operators allows to customize the C++ operators ( such as `+`, `-`, `*`, `/`, `%`, `^`, `&`, `|`, `~`, `!`, `=`, `<`, `>`, `+=`, `-=`, `*=`, `/=`, `%=`, `^=`, `&=`, `|=`, `<<`, `>>`, `>>=`, `<<=`, `==`, `!=`, `<=`, `>=`, `&&`, `||`, `++`, `--`, `,`, `->*`, `->`, `( )`, `[ ]`, `<=>`(since C++20) ) for operands of user-defined types. Example:

```cpp
#include <iostream>
class A {
 public:
  char a;
  A(char a): a(a) {}
};
std::string operator + (const A & a1, const A & a2) {
  std::string s;
  s += a1.a; s += a2.a;
  return s;
}
// A a1('a'), a2('b'); std::cout << a1+a2 << '\n';
```

## 11.4. Templates and operator overloading -- friend methods

example03.cpp

```
class Complex;
Complex operator + ( const Complex& obj1, const Complex& obj2);

class Complex {  // Class representing a complex number, e.g. 2.3 + 4.5 i
 public:
  Complex() : real(0), img(0) {}
  Complex(double real, double imag) : real(real), imag(imag){}
  friend Complex operator + ( Complex& obj1, const Complex& obj2);
 private:
  double real, imag;
};

Complex operator + ( const Complex& obj1, const Complex& obj2) {
  Complex temp;
  temp.real = obj1.real + obj2.real;
  temp.imag = obj1.imag + obj2.imag;
  return temp;
}
```

## 11.4. Templates and operator overloading -- conversions

Conversion operators can be used to convert one type to another type:

example04.cpp

```
#include <iostream>

// class representing a fraction through numerator and denominator, e.g. 7/9
class Fraction {
 public:
  Fraction(int n, int d) : n(n), d(d) {}
  // note that conversion does not have a return type:
  operator double() const { return double(n)/double(d); }
 private:
  int n, d;
};

int main() {
  Fraction frac(7, 9);
  double val = frac;  // conversion to double, double val = (double) frac;
  std::cout << val << '\n';
  return retVal;
}
```

## 11.4. Templates and operator overloading

With templates, now

example05.cpp

```
#include <iostream>

template <class T> class MyClass {
 public:
  MyClass(T c) : c(c) {}
  T c;
};
template <class U>
std::ostream& operator<<(std::ostream& out, const MyClass<U>& classObj) {
  out.put(classObj.c);
  return out;
}

int main() {
  MyClass<char> t('?');
  std::cout << t << '\n';
}
```

## 11.5. Standard Template Library (STL)

STL is a set of C++ template classes that:

- implement most common data structures for *containers* (queues, vectors, lists, stacks, maps, arrays, etc.), *algorithms* (sorting, search, etc.), *iterators* (to go through containers) , and *function objects or functors*
- in the form of a library of container classes, algorithms, and iterators,
- using components that are parameterized through templates.

## 11.5. The Standard Template Library (STL) - Vector

Dynamic array that resizes when elements are added or removed

```
std::vector<double> myVector( { 2.0, 4.0, 7.0 } );
std::vector<double> largeVector( 200, 1.0 );  // 200 elements, all ones
std::vector<double> copiedVector( myVector );  // exact copy of myVector
```

example07.cpp

```
#include <iostream>
#include <vector>
int main() {
  std::vector<std::string> name( {"John", "George", "Paul", "Smith"} );
  std::cout << "name size=" << name.size() << " of " << name.max_size();
  std::cout << " capacity=" << name.capacity() << \n';
  for (auto i = name.begin(); i < name.end(); i++)  // use iterators
    std::cout << *i << '\n';
  name.insert(name.begin()+2, "Tom");  // insert another name at third element
  for (const std::string & i : name)  // use range based for loop
    std::cout << i << '\n';
}
```

## 11.5. The Standard Template Library (STL) - Map

Container for ( key value, mapped value) pairs. Mapped values cannot have the same key values. Initialization can be done (since C++11) with initializer lists:

```
std::map<int, std::string> names = {{1,"Ann"}, {2,"Ames"}, {9,"Asa"}};

std::map<std::string, int> students;
students["Aaron"] = 173923;    // insert two new map elements
students["Zachary"] = 183211;

for (auto const & x : names)  // since C++11
  std::cout << "key:" << x.first << ",val:" << x.second << "\n";

for (auto const & [key, val] : students)  // since C++17
  std::cout << "key:" << key << ",val:" << val << "\n";
```

## 11.5. The Standard Template Library (STL) - Map

example08.cpp

```
#include <iostream>
#include <map>

int main() {
  std::map<std::string, int> students;
  students["Aaron"] = 173923;    // insert new map elements
  students["Zachary"] = 183211;
  students.insert( {"Patrick", 172932} );
  students.insert( std::pair<std::string,int>("Arnold", 161010) );

  for (auto const & x : names)  // since C++11
    std::cout << "key:" << x.first << ",val:" << x.second << '\n';

  for (auto const & [key, val] : students)  // since C++17
    std::cout << "key:" << key << ",val:" << val << '\n';

}
```

## 11.5. The Standard Template Library (STL) - transform

```
int increase(int x) { return (x+1); } // operation to execute

int array[] = {1, 2, 3};  // a simple container

std::transform(array, array+3, array, increase);

for ( const int & i : array )
  std::cout << i << ", ";  // output: 2, 3, 4,
```

Apart from the unary use above, transform also allows two input containers to be used (e.g., summing up elements of two containers). A functor can be passed for the given operation to be done on all elements of container, with the additional benefit of keeping its state.

## 11.5. The Standard Template Library (STL) - transform

Example of a functor:

```
#include <cassert>
class addX {  // this is a functor
 public:
  addX(int val) : x(val) {}  // Constructor sets how much to add
  int operator()(int y) const { return x + y; }  // ()
 private:
  int x;
};

int main() {
  addX add5(5);  // create object of functor class
  int i = add5(3);  // "call" it through the () operation
  assert(i == 8);
}
```

## 11.5. The Standard Template Library (STL) - transform

example09.cpp

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

int main() {
  std::vector a( { 16.0, 17.0, 18.0 } );  // init. lists since C++17
  std::vector b( { 10.0, 30.0, 60.0 } );
  std::vector<double> c;
  double n = 2.7;
  std::transform(a.begin(), a.end(),                    // iterators to input and
                 b.begin(), std::back_inserter(c),   // output vector elements
         [=](double x, double y) {return(x - y)/n; });
        // [=] : capture all external variables (n) in lambda by value
        // [&] : capture all external variables (n) in lambda by reference

  for (const double & i : c) {
    std::cout << i << "\n";
  }
}
```

## 11.5. The Standard Template Library (STL)

More reference information about the C++ STL is given at these locations:

- https://en.cppreference.com/w/cpp/standard_library
- https://cplusplus.com/reference/stl/