Templates

Rupesh Nasre.

OOAIA
January 2025

Queue Interface

```
class Queue {
public:
        Queue();
        ~Queue();
        void insert(int x);
        int remove();

private:
        int a[100];
        int head, tail;
};
```

Queue Implementation

```
class Queue {
public:
        Queue() {
                head = 0;
                tail = 0;
        }
        ~Queue() { }
        void insert(int x);
        int remove();

private:
        int a[100];
        int head, tail;
};
```

Queue Implementation

```
class Queue {
public:
        Queue() {
                head = 0;
                tail = 0;
        }
        ~Queue() { }
        void insert(int x);
        int remove();

private:
        int a[100];
        int head, tail;
};
void Queue::insert(int x) {
        // insert code.
}
```

Name resolution

This allows us to separate interface from its implementation.

Queue Implementation

```
class Queue {
public:
        Queue() {
                head = 0;
                tail = 0;
        }
        ~Queue() { }
        void insert(int x);
        int remove();

private:
        int a[100];
        int head, tail;
};
void Queue::insert(int x) {
        // insert code.
}
int Queue::remove() {
        // ...
}
```

Can we do anything about the dependence on int?

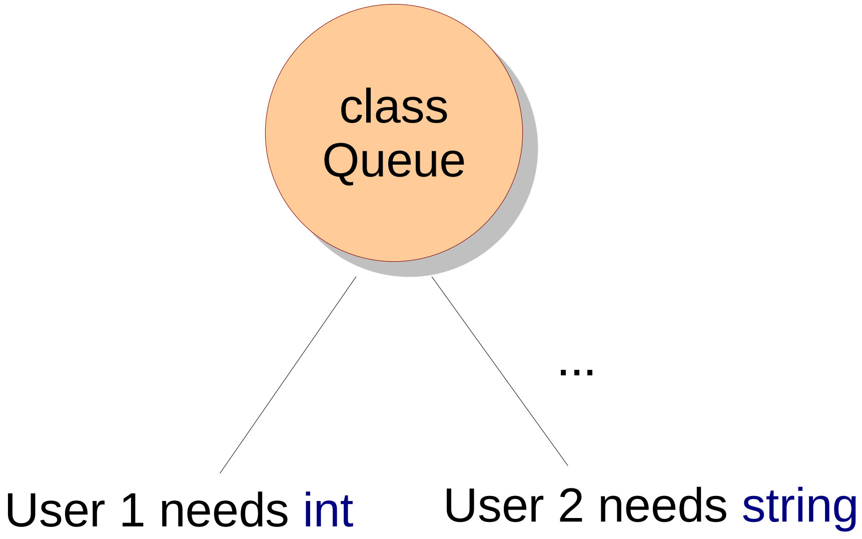
Type Generality

```
#define TYPE int
class Queue {
public:
        Queue() {
                head = 0;
                tail = 0;
        }
        ~Queue() { }
        void insert(TYPE x);
        TYPE remove();

private:
        TYPE a[100];
        int head, tail;
};
void Queue::insert(TYPE x) {
        // insert code.
}
TYPE Queue::remove() {
        // ...
}
```

I need to change the interface for different users.

Type Generality



class Queue

...

User 1 needs int

User 2 needs string

Type Generality

Tricks / Hacking

```
#define TYPE int
#include “queue”

void main() {
  Queue q;
  q.insert(10);
  ...
}
```

User 1

```
#define TYPE string
#include “queue”

void main() {
  Queue q;
  q.insert(“ooaia”);
  ...
}
```

User 2

User also needs to know which variable to define (TYPE).

Type Generality

```
#include “queue”

void main() {
  Queue<int> q;
  q.insert(10);
  ...
}
```

User 1

```
#include “queue”

void main() {
  Queue<string> q;
  q.insert(“ooaia”);
  ...
}
```

User 2

Templates

```
template <typename TYPE>
class Queue {
public:
        Queue() {
                head = 0;
                tail = 0;
        }
        ~Queue() { }
        void insert(TYPE x);
        TYPE remove();

private:
        TYPE a[100];
        int head, tail;
};
void Queue::insert(TYPE x) {
        // insert code.
}
TYPE Queue::remove() {
        // ...
}
```

I need NOT change the interface for different users.

These don't compile.

Templates

```
template <typename TYPE>
class Queue {
public:
        Queue() {
                head = 0;
                tail = 0;
        }
        ~Queue() { }
        void insert(TYPE x);
        TYPE remove();

private:
        TYPE a[100];
        int head, tail;
};
template <typename TYPE>
void Queue::insert(TYPE x) {
        // insert code.
}
template <typename TYPE>
TYPE Queue::remove() {
        // ...
}
```

Still don't compile.

Templates

```
template <typename TYPE>
class Queue {
public:
    Queue() {
        head = 0;
        tail = 0;
    }
    ~Queue() { }
    void insert(TYPE x);
    TYPE remove();

private:
    TYPE a[100];
    int head, tail;
};
template <typename TYPE>
void Queue<TYPE>::insert(TYPE x) {
    // insert code.
}
template <typename TYPE>
TYPE Queue<TYPE>::remove() {
    // ...
}
```

Compiles successfully.

Classwork

- Create a class **Group** templatized with the type of elements to be stored in the group.
- Implement methods: add and find.
- Instantiate int **Group** and check add+find.
- Instantiate string **Group** and check add+find.

Classwork

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

template<typename T>
class Group {
public:
        void add(T element);
        bool find(T element);
        T findwrapper(T element);

private:
        std::vector<T> group;
};
...
```

```
template<typename T>
void Group<T>::add(T element) {
        group.push_back(element);
}
template<typename T>
bool Group<T>::find(T e) {
        std::find(group.begin(), group.end(), e)
        != group.end();
}
template<typename T>
T Group<T>::findwrapper(T e) {
        std::cout <<
           (find(e) ? "Found " : "Not found ");
        return e;
}
```

```
int main() {
        Group<int> group;

        group.add(5);
        group.add(6);
        group.add(8);
        group.add(5);
        std::cout << group.findwrapper(5) << std::endl;
        std::cout << group.findwrapper(2) << std::endl;
        std::cout << group.findwrapper(6) << std::endl;

        Group<std::string> groupstr;
        groupstr.add("one");
        groupstr.add("two");
        groupstr.add("three");
        groupstr.add("five");

        std::cout << groupstr.findwrapper("two") << std::endl;
        std::cout << groupstr.findwrapper("four") << std::endl;
        std::cout << groupstr.findwrapper("five") << std::endl;

        return 0;
}
```

Multiple Template Arguments

```
template<typename T1, typename T2>
class Group {
public:
        Group() { std::cout << "class instantiated.\n"; }
        void add(std::pair<T1, T2> e);
        bool present(std::pair<T1, T2> e);
private:
        std::vector<std::pair<T1, T2> > elements;
};

template<typename T1, typename T2>
void Group<T1, T2>::add(std::pair<T1, T2> e) {
        elements.push_back(e);
}
template<typename T1, typename T2>
bool Group<T1, T2>::present(std::pair<T1, T2> e) {
        return (find(elements.begin(), elements.end(), e) != elements.end());
}
```

Function Templates

```
template <typename T>
T findMax(T a, T b) {
    // Works with any type that supports the > operator
    return (a > b) ? a : b;
}
int main() {
    // Same function works with different types
    int maxInt = findMax(10, 20);
    double maxDouble = findMax(3.14, 2.71);
    std::string maxString = findMax("apple", "banana");
    return 0;
}
```

Compared to polymorphism, a template has the same implementation.

This and the following slides are credited to Karan Agrawal.

Template Specialization

```
template <typename T>
T findMax(T a, T b) {
    // Works with any type that supports the > operator
    return (a > b) ? a : b;
}
int main() {
    // Same function works with different types
    int maxInt = findMax(10, 20);
    double maxDouble = findMax(3.14, 2.71);
    std::string maxString = findMax("apple", "banana");

    MyType one, two;
    MyType maxVar = findMax(one, two);
    return 0;
}
```

One way is to define
> for MyType.
Another is to specialize.

Template Specialization

```
template <typename T>
T findMax(T a, T b) {
    // Works with any type that supports the > operator
    return (a > b) ? a : b;
}
template <>
MyType findMax(MyType a, MyType b) {
    if (a.x > b.x || (a.x == b.x && a.y > b.y))
        return a;
    return b;
}
int main() {
    // Same function works with different types
    int maxInt = findMax(10, 20);
    double maxDouble = findMax(3.14, 2.71);
    std::string maxString = findMax("apple", "banana");

    MyType one, two;
    MyType maxVar = findMax(one, two);
    return 0;
}
```

Default Arguments

```
void fun(int x = 100, int y = 200) {
        cout << x << " " << y << endl;
}
int main() {
        fun();                  // prints 100 200
        fun(1);                 // prints 1 200
        fun(1, 2);              // prints 1 2

        return 0;
}



// Default arguments can only be at the end.
// For instance, the following is not permitted:
//    void fun(int x = 100, int y) {...}
```

Templates with Default Arguments

```
template <typename T, typename Container =
                                  std::vector<T>>
class Stack {
private:
    Container elements;
public:
    void push(const T& item) {
        elements.push_back(item);
    }
    T pop() {
        if (elements.empty()) {
            error("Stack is empty");
        }
        T last = elements.back();
        elements.pop_back();
        return last;
    }
};
```

```
int main() {
  // Using default container (vector)
  Stack<int> intStack;

  // Explicitly specifying a
  // different container
  Stack<double, std::deque<double>>
            doubleStack;

  return 0;
}
```