ECE2800J Final RC Part 3

SJTU-GC

Yang, Jingwen

Warnings

- Always remember that RCs only contain some key points, they can not be used as a substitute of lectures.
- If there is any difference between RC slides and lecture slides, always refer to lecture slides.
- Optional contents are not required, and there are potential errors.
- If you find anything wrong, tell me as soon as possible!
- Please check the correctness of the related code files by yourself.

L20: Template and Container

Motivation

- Some classes are used to contain some objects and do some operations on them. When we want to handle different types of objects, we may write similar code.
- It turns out we need to write the code only once, and can reuse it for each different type we want to use it for.
- **Polymorphism / Polymorphic Code**: reusing code for different types.
- One way to achieve polymorphism in C++ is **templated containers**.

Basic Grammar

Declaration

```
template <class T>
class List {
  private:
    struct Node {
        T data;
        Node* next;
        Node(T val) : data(val), next(nullptr) {}
    };
    Node *first;
    void removeAll();
    void copyList (Node* np);
  public:
    bool isEmpty();
    void insert(T v);
    T remove();
    List();
    List(const List &l);
    List &operator=(const List &l);
```

```
    ~List();
};
```

- C++ uses `class` to mean "type" here, but that doesn't mean only class names can serve as `T`. Any valid type such as `int` and `double` can.
- Write `template <class T> class List` in one line is OK.
- (Optional) `template <typename T> class List` is also OK. However, do not use it in the exam.

Implementation

- Implement the methods directly in the class, no `template <class T>` before each method. Also, no `List<T>::` before method name.
- Implement the methods outside the class:

```
template <class T>
bool List<T>::isEmpty() {
    return first == nullptr;
}
```

  - `template <class T>` is needed before each method implementation outside the class.
  - Write `template <class T> bool List<T>::isEmpty()` in one line is OK.
  - Key components: `template <class T>`, return type `bool`, class name `List<T>`, method name `isEmpty()`.
- The format of other methods:
  - Constructor:

```
template <class T>
List<T>::List() : first(nullptr) {}
```

  - Destructor:

```
template <class T>
List<T>::~List() {
    removeAll();
}
```

  - Assignment operator:

```
template <class T>
List<T> &List<T>::operator=(const List<T> &l) {
    if (this != &l) {
        removeAll();
        copyList(l.first);
```

```
        }
        return *this;
    }
```

- Do not add `<T>` to method name. Do add `<T>` to where the compiler does not know the actual class type.

Put in Header File

You should put your class member function implementation also in the `.h` file, following class definition. So, there is no `.cpp` for member functions.

(Optional) But why? Here is an **bad** example.

- `func.h`:

```
// Header guard
template <class T>
T add(T a, T b);
```

- `func.cpp`:

```
#include "func.h"
template <class T>
T add(T a, T b) {
    return a + b;
}
```

- `main.cpp`

```
#include "func.h"
int main() {
    int x = add<int>(1, 2);
}
```

The compiler will translate the template into a specific version when it knows which type is used. Here is the problem:

- When compiling `main.cpp`, the compiler knows that `add<int>` is being called, but it cannot see the implementation (only the declaration), so it assumes the function is defined elsewhere and defers resolution to the linking stage.
- When compiling `func.cpp`, although the implementation exists, it is not instantiated (because `add<int>` is not used in that translation unit), so no machine code for `add<int>` is generated.
- During linking, `main.o` requires `add<int>`, but `func.o` does not contain it, resulting in a linker error: "undefined reference to `add<int>`".

If we put the implementation into the header file, then only `main.cpp` will be compiled. Due to `#include "func.h"`, the compiler can see both the declaration and implementation of `add<int>`, so it can instantiate the template and generate the necessary machine code.

Container of Pointers

For now, we pass parameters by value. When objects are large, it is more efficient to pass pointers.

Basic Grammar

```
template <class T>
class PtrList {
  public:
    ...
    void insert(T *v);
    T *remove();
  private:
    struct node {
    node *next;
    node *prev;
    T *o;
    };
    ...
};
```

Why?

- Clear semantic meaning. `PtrList<class T>` means a container of pointers to objects of type T.
- Keep the interfaces consistent.
- More easy to implement, avoid confusions like `T**`.

Important Concepts

One invariant and three rules:

- **At-most-once invariant**: any object can be linked to at most one container at any time through pointer.
- **Existence Rule**: An object must be dynamically allocated before a pointer to it is inserted.
- **Ownership Rule**: Once a pointer to an object is inserted, that object becomes the property of the container. It can only be modified through the methods of the container.
- **Conservation Rule**: When a pointer is removed from a container, either the pointer must be inserted into some container, or its referent must be deleted after using.

WARNINGS:

- You'd better memorize these concepts. You should be able to judge which concept is applied/violated, and you should also know what is the concept.
- For concept problems, **ONLY** consider knowledge from this course! Do not consider more advanced features of C++ in your exam.

Some examples:

- Delete the object when destructing the container to fix conservation rule violation.
- Also copy the objects when doing deep copy to keep at-most-once invariant.

Polymorphic Container

Motivation

We want a container that can contain objects of different types. We can use inheritance and virtual functions to achieve this.

Basic Grammar

We can define a base class `Object`, and let all classes we want to store inherit from it.

```
class Object {
  public:
    virtual ~Object() {}
};
```

The virtual destructor is necessary to ensure that the derived class's destructor is called when deleting an object through a base class pointer.

Then we can define a container of `Object*`:

```
struct node {
  node *next;
  Object *value;
};
class List {
  ...
  public:
    void insert(Object *o);
    Object *remove();
    ...
};
```

(Optional) Why can't we define a container of `Object`?

- That is because of **object slicing**. When we store a derived class object in a base class variable, the derived part is "sliced off", and only the base class part is stored. This leads to loss of information and incorrect behavior.

Removal

```
class BigThing : public Object {
  ...
};
BigThing *bp;
bp = l.remove(); // Type error
```

- The return type of `remove()` is `Object*`, which cannot be directly assigned to a `BigThing*` without an explicit cast.
- Use `dynamic_cast` to safely downcast:

```
Object *op;
BigThing *bp;
op = l.remove();
bp = dynamic_cast<BigThing *>(op);
```

Deep Copy

If we do deep copy like before, we will have problems:

```
void List::copyList(node *list) {
  if(!list) return;
  copyList(list->next);
  Object *o = new Object(*list->value);
  insert(o);
}
```

`Object` can not take a `BigThing` to copy, because `Object` does not know how to copy `BigThing`.

So we need to add a virtual `clone()` method in `Object` to copy itself and return an `Object` pointer to the new copy.:

```
Object *BigThing::clone() {
  BigThing *bp = new BigThing(*this);
  return bp; // Legal due to substitution rule
}
void List::copyList(node *list){
  if(!list) return;
  copyList(list->next);
  Object *o = list->value->clone();
  insert(o);
}
```

More details are left for you to explore.

L21: Operator Overloading

Motivation

Sometimes we want to define existing operators for our classes to make them more intuitive and easier to use.

Arithmetic Operators

Arithmetic Operators: `+`, `-`, `*`, `/`, `%`

```
A operator+(const A &l, const A &r); // nonmember function
// returns l "+" r
A A::operator+(const A &r); // member function
// returns *this "+" r
```

- The input parameters are `const` references, which means the input objects are not modified.
- The return type is an object, not a reference. No `const` because the return value is a new object. No reference because the new object is not a reference to any existing object.
- For class member functions, the first input is implicitly `*this`. And the only input is the right-hand side object.
- For operator overloading as class member functions, the final `const` means that the member function does not modify the input object and can be called on `const` objects.
- The input type and return type can be changed according to the actual situation. Be careful about the C++ matching rules.
- The above two functions can appear together in the same program. The compiler will choose the appropriate one based on the context.

Assignment Operators

Assignment Operators: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, ...

```
A& A::operator+=(const A& r); // member function
A& A::operator+=(A& l, const A& r); // non-member function
```

- `operator=` **must** be a non-static member function! Why?
- Pay attention to the return type. It should be a reference to the object itself. In this way, we can support chained assignments like `a += b += c;`. Note that it is equivalent to `b += c; a += b;` (right associativity).
- The input parameter is a `const` reference, meaning the input object is not modified.
- Although it is possible to overload assignment operators (except `=`) as non-member functions, it is not recommended.
- The input type of `r` can be changed according to the actual situation. Be careful about the C++ matching rules.

Friend

Sometimes we need to overload operators as non-member functions, but they need to access private members of the class. In this case, we can declare them as `friend` functions inside the class.

```
class A {
  private:
    int data;
  public:
    friend A operator+(const A &l, const A &r);
};
```

We can also declare friend classes:

```
class B; // forward declaration
class A {
  private:
    int data;
  public:
    friend class B; // B can access private members of A
};
```

- The friend function declared in a class is not a member function of the class. It is a non-member function that can access the private members of the class. Do not write `A::f` or `B::f` outside the class.
- The friend relation is not mutual. If `A` is a friend of `B`, `B` is not a friend of `A`, then `B` cannot access the private members of `A`.

Subscript Operator

Subscript Operator: `[]`

```
int& A::operator[](int index); // member function
const int& A::operator[](int index) const; // member function for const objects
```

- `operator[]` **must** be a non-static member function!
- The `const` version is used when the object is `const`, and it returns a `const` reference to prevent modification.
- The non `const` version is used when the object is non-`const`, and it returns a non-`const` reference to allow modification.
- An `const` object will only call the `const` version, and a non-`const` object will only call the non-`const` version.

Stream Operators

Stream Operators: `<<`, `>>`

```
std::ostream& operator<<(std::ostream& os, const IntSet& s) {
  // print s to os
  return os;
}
std::istream& operator>>(std::istream& is, IntSet& s) {
  // read from is to s
  return is;
}
```

- They **must** be non-member functions! The first operand is not of the class type.
- Return type is a reference to the stream object to support chained operations like `std::cout << a << b;`.
- These operators are left associative.

L22: Linear List and Stack

Linear List

- A collection of zero or more integers with duplicates possible.
- It supports insertion and removal by position.

Insertion

```
void insert(int i, int v);
// if 0 <= i <= N (N is the size of the list), insert v at position i;
// otherwise, throws BoundsError exception.
```

- Example:

```
L1 = (1, 2, 3)
L1.insert(0, 5) = (5, 1, 2, 3);
L1.insert(1, 4) = (1, 4, 2, 3);
L1.insert(3, 6) = (1, 2, 3, 6);
L1.insert(4, 0) throws BoundsError
```

Removal

```
int remove(int i);
// if 0 <= i < N, remove and return the i-th element;
// otherwise, throws BoundsError exception.
```

- Example:

```
L1 = (1, 2, 3)
L1.remove(0) = (2, 3);
L1.remove(1) = (1, 3);
L1.remove(2) = (1, 2);
L1.remove(3) throws BoundsError
```

Stack

Stack is a data structure that supports insertion and removal at one end only. It follows the Last-In-First-Out (LIFO) principle. It is a restricted version of linear list.

Basic Operations

- `size()`: number of elements in the stack.
- `isEmpty()`: checks if stack has no elements.
- `push(Object o)`: add object `o` to the top of stack.
- `pop()`: remove the top object if stack is not empty; otherwise, throw `stackEmpty`.
- `Object &top()`: return a reference to the top element.

Array Implementation

- `Array[MAXSIZE]`.
- Maintain an integer `size` to record the size of the stack.
- `size()`: return `size`.
- `isEmpty()`: return (`size` == 0).
- `push(Object o)`: add object `o` at index `size` of the array and increment `size`. Allocate more space if necessary.
- `pop()`: If `isEmpty()`, throw `stackEmpty`; otherwise, decrement `size`.
- `Object &top()`: return a reference to the top element `Array[size-1]`.

Linked List Implementation

- `size()`: `LinkedList::size();`
- `isEmpty()`: `LinkedList::isEmpty();`
- `push(Object o)`: insert object at the beginning `LinkedList::insertFirst(Object o);`
- `pop()`: remove the first node `LinkedList::removeFirst();`
- `Object &top()`: return a reference to the object stored in the first node.

L23: Queue

Queue is also a linear data structure like stack. But contrary to stack, it follows the First-In-First-Out (FIFO) principle.

Basic Operations

- `size()`: number of elements in the queue.
- `isEmpty()`: check if queue has no elements.
- `enqueue(Object o)`: add object `o` to the rear of the queue.

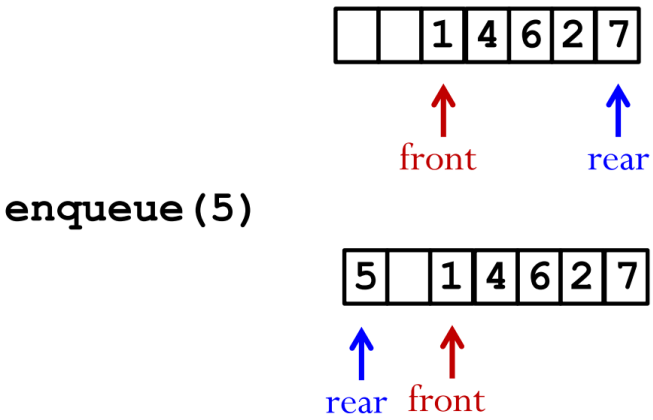
- `dequeue()`: remove the front object of the queue if not empty; otherwise, throw `queueEmpty`.
- `Object &front()`: return a reference to the front element of the queue.
- `Object &rear()`: return a reference to the rear element of the queue.

Linked List Implementation

- Double-ended singly-linked list.
- `enqueue(Object o)`: append object at the end `LinkedList::insertLast(Object o);`
- `dequeue()`: remove the first node `LinkedList::removeFirst();`
- `size()`: `LinkedList::size();`
- `isEmpty()`: `LinkedList::isEmpty();`
- `Object &front()`: return a reference to the object stored in the first node.
- `Object &rear()`: return a reference to the object stored in the last node.

Circular Array Implementation

- Use a circular array to store the elements.
- The class should have an integer to track the size of the queue, an integer to track the front index, and an integer to track the capacity of the array.
- Also, we can maintain a boolean variable to check if the queue is full. If the size of the queue is equal to the capacity of the array, the queue is full.



- `isEmpty()`: `return (count == 0);`
- `size()`: `return count;`
- `enqueue(Object o)`: add an object to the rear of the array and advance the rear index.
- `dequeue()`: remove an object from the front of the array and advance the front index.
- The indexes of the front and rear can be calculated by:

```
front = (front + 1) % capacity;
rear = (rear + 1) % capacity;
```

Tips for Final Exam

- Be familiar with data structures mentioned in this course (like list/tree from project 2 and from lectures after midterm). Know what they are, the motivation for using specific data structures (if any) and the possible operations on them. At least know the general idea on how to implement them.

- Start with project 5, at least go through it and have general ideas. There's 100% chance that you will see some questions related to data structure implementation like project 5 or exercise 6 in your final exam.
- Pay attention to base cases and edge cases, like how to handle `nullptr` in some operations.
- Be familiar with the tiny details. Grasping all these details can help you get a high score, even if you cannot solve the hardest coding problem. **Conceptual questions** also make up a part of the final exam.
- Do not review and memorize the concepts by brute force. Try to understand the logic behind them and the motivation for using them. This can help you remember them better.
- **DO NOT WAIT UNTIL THE LAST MINUTE!** Start early, and don't try to cram everything in one night.

References

[1] Qian, Weikang, ECE2800J 25FA Lectures 20-23.

[2] ECE2800J 24FA Final RC Part 3.