ECE2800J RC 6

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!

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.
- 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.

Optional

If I define the template like this:

```
template <class T>
class List {
  private:
    struct Node {
        T data;
        Node* next;
        Node(T val);
    };
    ...
  public:
    ...
};
```

How to implement the constructor of `Node` outside the template declaration?

```
template <class T>
List<T>::Node::Node(T val) : data(val), next(nullptr) {}
```

You can explore other nested structures by yourself.

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.

References

[1] Qian, Weikang, ECE2800J 25FA Lecture 20.

[2] ECE2800J 24FA Final RC Part 3.