16 Container ADT

16.1 Write a container ADT example: set

我们需要 set 这个 Container ADT 满足的条件：

1. 大小是 immutable 的，并且可以获取
2. 每个元素是 unique 的，并且可以查找
3. 可以插入和移除元素
4. 可以打印
5. ...

下面为 expect 的操作：

```
int main() {
    Set s;                      // s is empty
    s.insert(5);
    s.insert(12);
    s.insert(5);                // ignored – 5 already in set
    cout << s << endl;          // {5, 12} or {12, 5}
    cout << s.size() << endl;   // 2
    s.remove(5);
    cout << s << endl;          // {12}
    if (s.contains(14))         // should return false
        cout << "Where did this come from??" << endl;

    return 0;
}
```

所以一部分 interface 应该是长这个样子：

```
class Set{
```

```
  public:
    // EFFECTS: Constructs a Set, with size 0.
    Set();

    // EFFECTS: Adds element e to the Set if not already present
    void insert(int e);

    // EFFECTS: Removes element e from the Set if present
    void remove(int e);

    // EFFECTS: Returns true if e is in the set, false otherwise
    bool contains(int e) const;

    // EFFECTS: Returns the number of elements in the Set
    int size() const;

    void print(std::ostream &os) const;
};
```

16.1.1 `Static` member variables

我们先假设这是一个容纳 int 型的 set.

set 的容量有一定限制（因为这是计算机里，和现实的 set 不同，不能是无限的）

我们可以设置一个最大容量：

```
private:
    static const int CAPACITY = 10; // set 能够容纳的最大容量
    int elts[CAPACITY]; // 当前 set 的外观, 用 array 容纳
    int elts_size; // 当前 set 的大小
};
```

这里 `static` 是一个 keyword，它的意义是：

1. 这个 class 的所有 instances 都 share 这个 member variable.
2. static storage duration，lives throughout the whole program，就像 global variable，只不过仅限于这个 class 可使用.
3. lives inside a class's scope，比起 global variable lives in the global scope 而言更加 organized 一点.

（复习：global variable）

1. global variable 就是 scope 为整个 program 的 variable，可以在任何 scope，比如任何函数体内被访问。

```
#include <iostream>
int globalVar = 10; // 这是一个全局变量

void demoFunction() {
    std::cout << globalVar << std::endl; // 可以访问
}

int main() {
    std::cout << globalVar << std::endl; // 可以访问
    demoFunction();
    return 0;
}
```

`static` 就是 throughout 某个 class 的，对于这个 class 而言全局的变量。

16.1.2 private helper function

我们发现这三个 member function 都要做一件事情:

1. `contains`: **search** the array looking for a specific number
2. `remove`: **search** the array for a number; if it exists, remove it
3. `insert`: **search** the array for a number; if it doesn't exist, add it

我们这个时候肯定要写一个 `search` 的 function 来防止 code reproduction，否则如果我们要更改代码就要改三次，这在大工程里是很不好的。

我们需要这个 helper function 是 `private` 的，因为它在 class 之外是不需要的。这样也可以避免 exposing implementation details。

```
int Set::index_of(int e) const {  // serch helper function
    for (int i = 0; i < elts_size; ++i) {
      if (elts[i] == e)
        return i; //found
    }
    return -1; //not found
}

Set::Set() // constructor
  : elts_size(0) {}

int Set::size() const {
    return elts_size;
}

bool Set::contains(int e) const {
    return index_of(e) != -1;
}
```

```
void Set::insert(int e) {
    if (contains(e)) // e already in the set
        return;
    assert(elts_size < CAPACITY); //REQUIRES!
    elts[elts_size] = e;  // add e to the array
    elts_size ++;
}

void Set::remove(int e) {
    int target = index_of(e);
    if (target == -1) // not found
        return;

    elts[target] = elts[elts_size - 1];
    --elts_size;      // we have one fewer elements now!
}
```

16.2 Operator overloading

我们想要随时能够灵活地打印一个 container ADT 的结构，就需要 overload operator.

比如 `<`, `==`, `<<`, `>>` ..... 这些 operator 都可以 overload. 标准库只写了参数为标准数据类型（`int`, `string`....) 的这些 operator.

我们可以在**class 内声明一个 friend function. 表示这个 function 是 class 外的 （并不是`set::...` 的 function），但是它被允许 access 这个 class 的 private 和 protected 的 members.**

```
class Set {
public:
  ...
private:
  ...
    friend std::ostream &operator<<(ostream &os, const Set &s);
    // friend 关键词的 function, 表示这个 function 是来自 class 外部的, 并不是一个 member, 但是它被允许 access 这个 class 的 private 和 protected 的 members.
};

void Set::print(ostream &os) const {
    os << "{";
    for (int i = 0; i < elts_size - 1; ++i)
        os << elts[i] << ", ";

    os << elts[elts_size - 1] << "} ";
}


std::ostream &operator<<(ostream &os, const Set &s) {
```

```
  s.print(os);
  return os;
}
```

这个 overloading 还是很讲究的。

1. 传进去的参数依次分别是**左右两边的 object 的 ref**。通常我们都使用**引用传递**，因为我们不想随便创造 copy，因为可能 obejct 很大，会影响性能。
2. 既然参数都是引用变量，那么我们需要注意要不要加上 `const`. 比如 `<<`，左边是 `ostream`, 右边是要写的东西。我们要修改 `ostream`，所以不加 `const`；但是写的东西我们不要修改，所以要加上 `const`.
3. 最后**返回的是我们传进去的 ostream `os`**, 比如 `cout`，这样之后又可以 chainly 继续 write. **return 的 ostream 也是 pass by reference 的!**

最后的效果是：

```
Set s;
cout << s;  // 传进去的参数是 &cout, &s

Set s1, s2, s3, s4;
cout << s1 << s2 << s3 << endl;
// 其实就是 (((cout << s1) << s2) << s3) << endl;
// 这里 (cout << s1) 改变了 cout 这个 ostream 并把 cout 返回出来了
// 因而之后就是 ((cout << s2) << s3) << endl;
// 之后: (cout << s3) << endl;
// cout << endl;
// 最后只剩一个 cout, 结束
```

16.3 Template: generic programming

Generic programming(泛型编程) 指 implement algorithm without data types，而 specify data types later.

C++ 中支持 generic programming 的 mechanism 是 **template(模版)**.

实际上我们一直都在使用一个 template container ADT，就是 vector. vector 的成员可以是任何类型的，比如 `int`, `double`, ..., 甚至是我们自己定义的 ADT.

```
vector<double> data;
```

我们可以把我们的 set 也加上 template，这样它就可以是一个广泛类型的 set，而不局限于某个特定类型。

Template 也是一种 polymorphism. 之前的 Dynamic type 是 Subtype polymorphism 而 template 是 parametric polymorphism.

16.3.1 Template Notation

```
template <typename T>
```

在 function 或者 class 的 declaration 的前加上这一行，表示下面这个 function/class 中所有出现的 `T` 作为 typename，都用你传进来的代替。

Template 可以用于 class 或者 function. **注意：**

1. **当用于 class 的时候，class 的所有 member functions 如果 implement 在 class declaration 外，前面都要重新加上 `template <typename T>`！！**
2. 所有你使用 class 名称的地方都要加上 `<T>`. 比如所有 `Set` 出现的地方都要变为 `Set<T>`，**除了 constructor！！**

   ```
   Set<T>::Set()
     :... {}
   ```

用于 function:

```
template <typename T>
T max(T a, T b) {
// 比如如果你传进去的 x, y 为 int, 则是 int max(int a, int b);
    return a > b ? a : b;
    //condition? expression1 : expression2;
    // 这个 > ? :是个三元运算符, 表示条件如果为真就返回1的结果, 假就返回2的结果
}

int main() {
    int x = 1;
    int y = 2;
    cout << max(x, y);
}
```

用于 class:

```
template <typename T>
class Set{
  public:
    // EFFECTS: Constructs a Set, with size 0.
    Set();

    // EFFECTS: Adds element e to the Set if not already present
    void insert(T e);

    // EFFECTS: Removes element e from the Set if present
    void remove(T e);
```

```
    // EFFECTS: Returns true if e is in the set, false otherwise
    bool contains(T e) const;

    // EFFECTS: Returns the number of elements in the Set
    int size() const;

    void print(std::ostream &os) const;

private:
    //EFFECTS: returns the index of e if it is in the
    //  set, -1 otherwise.
    int index_of(T e) const;

    static const int CAPACITY = 10;
    T elts[CAPACITY];
    int elts_size;
};

template <typename T>
std::ostream &operator<<(std::ostream &os, const Set<T> &s);

template <typename T>
int Set<T>::index_of(T e) const {
    for (int i = 0; i < elts_size; ++i) {
      if (elts[i] == e)
        return i; // found
    }
    return -1;    // not found
}

template <typename T>
Set<T>::Set()
  : elts_size(0) {
}

template <typename T>
int Set<T>::size() const {
    return elts_size;
}

template <typename T>
bool Set<T>::contains(T e) const {
    return index_of(e) != -1;
}

template <typename T>
void Set<T>::insert(T e) {
    if (contains(e)) // e already in the set
```

```
        return;
    assert(elts_size < CAPACITY); //REQUIRES!
    elts[elts_size] = e;  // add e to the array
    elts_size++;
}

template <typename T>
void Set<T>::remove(T e) {
    int target = index_of(e);
    if (target == -1) // not found
        return;

    elts[target] = elts[elts_size - 1];
    --elts_size;      // we have one fewer elements now!
}

template <typename T>
void Set<T>::print(std::ostream &os) const {
    os << "{";
    for (int i = 0; i < elts_size - 1; ++i)
        os << elts[i] << ", ";

    os << elts[elts_size - 1] << "} ";
}

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

#endif
```

使用例：

```
int main() {
    Set<int> si;                // si is empty
    si.insert(5);
    si.insert(12);
    si.insert(5);               // ignored — 5 already in set
    cout << si << endl;         // {5, 12} or {12, 5}
    cout << si.size() << endl;  // 2
    si.remove(5);
    cout << si << endl;         // {12}
    if (si.contains(14))        // should return false
        cout << "Where did this come from??" << endl;

    cout << "=========================" << endl;
```

```
    Set<string> ss;             // ss is empty
    ss.insert("apple");
    ss.insert("banana");
    ss.insert("apple");         // ignored — apple already in set
    cout << ss << endl;         // {apple, banana} or {banana, apple}
    cout << ss.size() << endl;  // 2
    ss.remove("apple");
    cout << ss << endl;         // {banana}
    if (ss.contains("cranberry"))        // should return false
        cout << "Where did this come from??" << endl;

    return 0;
}
```

16.4 注意事项

16.4.1 template class 的 declarations 和 definitions 必须在同一个文件

我们至今为止平时在写非 template class 的时候总是将 declarations 放在 .hpp 中，将定义放在 .cpp 中，为了更简洁的 interface. 但是**一旦使用 template，那么我们就必须把 declarations 和 definitions 都放在同一个 (.hpp) 文件里。**

原因是：当使用 template 定义一个 class 或 function 时，实际上**是在定义一个模板，而不是一个实际可编译的东西。**而 Initialization 却是在 compiler 遇到 template 的具体使用时进行的。compiler 在 initializaiton 时的具体做法是根据 template 为每种类型生成专门的代码。

例如，当你创建一个 `std::vector<int>` 时，compiler **会对照着 template 自动生成一个把 template 中的 `T` 换成 `int` 的函数，然后对应这段自动生成的代码来 instantiate 这个 object。**

由于 template 的这种工作原理，**compiler 在 compile template code 时必须有 template 的完整 definitions。**

我们平时写的非 template 的 code

1. compilation: 当 compiler compile 一个 source file (如 `set.cpp`，`main.cpp`)的时候，如果该文件 `#include` 了一个 class 的 `hpp` declaration，compiler 会检查 source file 中对该 class 的成员的引用是否与 `hpp` 中的 declaration 匹配，只要语法正确就会编译成功，生成 `.o` 的 object file. 这个object file 包含了未被解析的 declarations 的符号引用，这些引用指向我们在 source file 中写的 definitions.
2. linking: linker 把所有 object files 合并成一个 `.exe` 文件。在这个过程中，linker 会解析之前 compilation 阶段留下的未解析的 declarations 的符号引用，查找其指向的 definitions.

这一过程允许我们将 non-template class 的 declaration 和 definition 分离开。

Non-template class 可以分离 declaration 和 definitions，是因为 **compilation 和 linking 过程是分开的**，compiler 生成的是对符号的 ref，在 linking 阶段这些 refs 才被解析到具体的 definitions。

而相反，**template 的 instantiation 是一个 compile-time process，在 linking 前就要完成，需要在模板被使用时具有可访问的完整定义。** 只有当 compiler 看到 template 的完整定义时，它才能为特定的 typename 实例化 template。否则会产生 linker error。

16.4.2 Include Guards

include 同一个 header file 超过一次会导致 compiler error. 尤其是当 A include B 和 C，B include C，那么 A 就等同于 include 了 C 两次，导致了 compiler error。

为了防止这个问题，我们一般会在 header files 里加上 **include guards**。作用是让 compiler **ignore the code if the header is included a second time.**

```
#ifndef SOMETHING.hpp
#define SOMETHING.hpp

//...
#endif
```