Comp151

Generic Programming:
Container Classes

Container Classes

- Container classes are a typical use for class templates, since we need container classes for objects of many different types, and the types are not known when the container class is designed.
- Let's design a container that looks like an array, but that is a first-class type: so that assignment and call-by-value is possible.
- We want the container to be *homogeneous*: all the elements must have the same type.
- But should a container with 10 `int` elements be the same type as a container with 20 `int` elements?

  Both choices are sensible design decisions.

Remark: The `vector` type in STL is better than the classes we'll write in this lecture, so this is just for understanding. We are doing this to illustrate how C++'s actual `vector`, `list`, etc. can be implemented.

Example: Container Class – bunch.hpp

```
template<typename T, int N>
class Bunch {
public:
    Bunch();
    Bunch(const Bunch &B);
    ~Bunch();

    int size() const { return N; }
    T& operator[ ](int i) { return value_m[i]; }
    T& operator=(const Bunch &B);
private:
    T value_m[N];
};
```

Example: Use of Class Bunch

```
Bunch<int, 10> a;
cout << a[3];
a[7] = 13;
++a[2];

Bunch<string, 50> b;
b[49] = "Hello world";

Bunch<string, 50> c;
c = b;                    // Legal
Bunch<int, 20> d;
d = a;                    // Error: d and a are of different types
```

A More Flexible Container Class – array.hpp

```
#ifndef ARRAY_HPP
#define ARRAY_HPP

template<typename T>
class Array {
private:
    T* value_m;
    int size_m;
public:
    Array(int n = 10);                                  // Default/conversion constructor
    Array(const Array& A);                              // Copy constructor
    ~Array();

    int size() const { return size_m; }
    Array<T>& operator=(const Array<T>& A);             // Assignment operator
    T& operator[ ](int i) { return value_m[i]; };       // Access to an element
    const T& operator[ ](int i) const { return value_m[i]; };   // Const access to an element
};

#endif
```

Example: Use of Class Array

```
#include <iostream>
#include "array.hpp"
using namespace std;
int main()
{
    Array<int> a;
    cout << a.size() << endl;
    a[9] = 17;                    // Ok: uses non-const version of operator[ ]
    ++a[2];                       // Ok: uses non-const version of operator[ ]
    cout << a[2] << endl;

    Array<int> b(5);
    cout << b.size() << endl;

    const Array<int> c(20);
    c[1] = 5;                     // Error: assignment to read-only location
    cout << c[1] << endl;

    a = c;
    cout << a[2] << endl;
}
```

Example: Constructors/Destructor of Class Array

```
template<typename T>
Array<T>::Array(int n) : value_m( new T[n] ), size_m(n) { }

template<typename T>
Array<T>::Array(const Array<T>& A)
    : value_m( new T[A.size_m] ), _size(A.size_m)
{
    for (int i = 0; i < size_m; ++i) {
        value_m[i] = A.value_m[i];
    }
}

template<typename T>
Array<T>::~Array() { delete[ ] value_m; }
```

Shallow Copy and Deep Copy

```
Array<int> A(10);
Array<int> B(A);
```

- Shallow Copy:
  - If you don't define your own copy constructor, the copy constructor provided by the compiler simply does member-wise copy.
  - Then `A` and `B` will share to the same `value_m` array.
  - If you delete `A`, and then `B`, you will have an error as you will delete the embedded `value_m` array twice from the heap.
  - Basically, shallow copy is a bad idea if an object *owns* data.
- Deep Copy:
  - To take care of the ownership, redefine the copy constructor so that each object has its own copy of the “owned" data members.

Assignment Operator

- Idea: To assign `b = a`, first throw away the old data `b.value_m`, then create a new one and assign the elements from `a.value_m`.

```
template<typename T>
Array<T>& Array<T>::operator=(const Array<T>& A)
{
    delete [ ] value_m;
    size_m = A. size_m;
    value_m = new T[size_m];
    for (int i = 0; i < size_m; ++i) {
        value_m[i] = A.value_m[i];
    }
    return *this;
}
```

Assignment Operator (cont'd)

- There is a serious problem with the previous code. In the assignment a = a, the data in the container is lost!
- Solution: When the assignment argument is the same as the object being assigned to, don't do anything.

```
template<typename T>
Array<T>& Array<T>::operator=(const Array<T>& A)
{
    if (this != &A) {
        delete [ ] value_m;
        size_m = A. size_m;
        value_m = new T[size_m];
        for (int i = 0; i < size_m; ++i) {
            value_m[i] = A.value_m[i];
        }
    }
    return *this;
}
```

Assignment Operator (cont'd)

- Here is another way of implementing the assignment operator. Quiz: Why does this elegant trick work??

```
template<typename T>
Array<T>& Array<T>::operator=(const Array<T>& A)
{
    size_m = A.size_m;
    Array<T> temp(A);
    std::swap( value_m, temp.value_m);
    return *this;
}

// Here's what std::swap() basically looks like:
template<typename T>
void swap(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

Output Operator

- The following output operator is not a member of the `Array<T>` class, but a function template.
- Function templates and class templates work together very well: We can use function templates to implement functions that will work on any class created from a class template.

```
template<typename T>
ostream& operator<<(ostream& os, const Array<T>& A)
{
    for (int i = 0; i < A.size(); ++i) {
        os << A[i] << ' ';
    }
    return os;
}
```

Why 2 Different Subscript Operators?

- We have 2 subscript operators, and it looks as if we are violating the overloading rule. Both have the same name and the same arguments.

```
Array<int> a(3);
a[2] = 7;            // Quiz: which version of operator[ ] is called?
```

- In the above code, we need a subscript operator that returns an `int&`, not a `const int&`.
- But this subscript operator does not work in this code:

```
int last_element(const Array<int>& a)
{
    return a[a.size() - 1];
}
```

Why 2 Different Subscript Operators?

- The argument a of `last_element()` is a `const Array<int>&`.
- Therefore it can <u>only</u> call `const` member functions: in this example,
  - int size() const
  - const T& operator[ ](int i) const
- Note: On the other hand, if bad programmers are not so strict with `const` correctness (which is a <u>bad</u> idea), they could simply define one subscript function as:

T& operator[ ](int i) const { return value_m[i]; } *// This is dangerous! (Why?)*