

Object Oriented Programming C++

Chapter: Templates

Coding Questions and Solutions (Set 5)

Contents: Part A: Function Template Programs. Part B: Class Template Programs. Part C: Predict the Output. Part D: Find the Error. Every program is complete, with its output. These problems are different from Sets 1 to 4.

Part A: Function Template Programs

Q1. Write a function template for binary search on a sorted array. It should return the index of the key, or -1 if the key is not found.

Solution:

```
#include <iostream>
using namespace std;

template <class T>
int binarySearch(T a[], int n, T key)
{
    int lo = 0, hi = n - 1;
    while (lo <= hi)
    {
        int mid = (lo + hi) / 2;
        if (a[mid] == key)
            return mid;
        else if (a[mid] < key)
            lo = mid + 1;
        else
            hi = mid - 1;
    }
    return -1;
}

int main()
{
    int a[] = {2, 5, 8, 12, 16, 23};
    double b[] = {1.5, 2.5, 3.5};

    cout << "Index: " << binarySearch(a, 6, 12) << endl;
    cout << "Index: " << binarySearch(b, 3, 4.5) << endl;
    return 0;
}
```

Output:

```
Index: 3
Index: -1
```

The array must already be sorted in ascending order for binary search to work.

Q2. Write a recursive function template to find the GCD (HCF) of two integers. Use it with `int` and `long`.

Solution:

```
#include <iostream>
using namespace std;

template <class T>
T findGCD(T a, T b)
{
    return (b == 0) ? a : findGCD(b, a % b);
}

int main()
{
    cout << findGCD(48, 18) << endl;
    cout << findGCD(100L, 75L) << endl;
    cout << findGCD(17, 5) << endl;
    return 0;
}
```

Output:

```
6
25
1
```

The template uses the % operator, so it can be used only with integer types such as `int` and `long`. Using it with `double` gives a compile-time error.

Q3. Write a function template to merge two sorted arrays into a third sorted array.

Solution:

```
#include <iostream>
using namespace std;

template <class T>
void mergeArrays(T a[], int n, T b[], int m, T c[])
{
    int i = 0, j = 0, k = 0;
    while (i < n && j < m)
    {
        if (a[i] < b[j])
            c[k++] = a[i++];
        else
            c[k++] = b[j++];
    }
    while (i < n) c[k++] = a[i++];
    while (j < m) c[k++] = b[j++];
}

int main()
{
    int a[] = {1, 4, 7};
    int b[] = {2, 3, 9, 10};
    int c[7];
    mergeArrays(a, 3, b, 4, c);
    for (int i = 0; i < 7; i++)
        cout << c[i] << " ";
    cout << endl;

    double x[] = {0.5, 2.5};
}
```

```

double y[] = {1.5};
double z[3];
mergeArrays(x, 2, y, 1, z);
for (int i = 0; i < 3; i++)
    cout << z[i] << " ";
cout << endl;
return 0;
}

```

Output:

```

1 2 3 4 7 9 10
0.5 1.5 2.5

```

Q4. Write a function template that counts how many elements of an array are greater than a given value.

Solution:

```

#include <iostream>
using namespace std;

template <class T>
int countGreater(T a[], int n, T value)
{
    int count = 0;
    for (int i = 0; i < n; i++)
        if (a[i] > value)
            count++;
    return count;
}

int main()
{
    int a[] = {5, 12, 8, 20, 3};
    char c[] = {'a', 'm', 'z'};

    cout << countGreater(a, 5, 7) << endl;
    cout << countGreater(c, 3, 'k') << endl;
    return 0;
}

```

Output:

```

3
2

```

Q5. Write a function template that checks whether an array is sorted in ascending order.

Solution:

```

#include <iostream>
using namespace std;

template <class T>
bool isSorted(T a[], int n)
{
    for (int i = 0; i < n - 1; i++)
        if (a[i] > a[i + 1])
            return false;
}

```

```

        return true;
    }

    int main()
    {
        int a[] = {1, 2, 5, 9};
        int b[] = {4, 2, 7};
        double c[] = {1.1, 2.2, 3.3};

        cout << boolalpha;
        cout << isSorted(a, 4) << endl;
        cout << isSorted(b, 3) << endl;
        cout << isSorted(c, 3) << endl;
        return 0;
    }

```

Output:

```

true
false
true

```

Q6. Write a function template that finds both the minimum and the maximum of an array and returns them through reference parameters.

Solution:

```

#include <iostream>
using namespace std;

template <class T>
void findMinMax(T a[], int n, T &mn, T &mx)
{
    mn = mx = a[0];
    for (int i = 1; i < n; i++)
    {
        if (a[i] < mn) mn = a[i];
        if (a[i] > mx) mx = a[i];
    }
}

int main()
{
    int a[] = {7, 3, 9, 1, 5};
    int lo, hi;
    findMinMax(a, 5, lo, hi);
    cout << "Min=" << lo << " Max=" << hi << endl;

    double b[] = {2.5, 0.5, 4.5};
    double l, h;
    findMinMax(b, 3, l, h);
    cout << "Min=" << l << " Max=" << h << endl;
    return 0;
}

```

Output:

```

Min=1 Max=9
Min=0.5 Max=4.5

```

A function can return only one value, so the second value is returned through a reference parameter. Here both results are returned through references.

Q7. Write a function template `larger()` that returns the larger of two values. Give a specialization for C-style strings so that it returns the longer string.

Solution:

```
#include <iostream>
#include <cstring>
using namespace std;

template <class T>
T larger(T a, T b)
{
    return (a > b) ? a : b;
}

template <>
const char *larger<const char *>(const char *a, const char *b)
{
    return (strlen(a) > strlen(b)) ? a : b;
}

int main()
{
    cout << larger(3, 9) << endl;
    cout << larger("apple", "kiwi") << endl;
    cout << larger(2.5, 1.5) << endl;
    return 0;
}
```

Output:

```
9
apple
2.5
```

For `const char *`, the general version would compare the addresses of the two strings. The specialization compares their lengths instead.

Q8. Write a function template to rotate the elements of an array one position to the left.

Solution:

```
#include <iostream>
using namespace std;

template <class T>
void rotateLeft(T a[], int n)
{
    T first = a[0];
    for (int i = 0; i < n - 1; i++)
        a[i] = a[i + 1];
    a[n - 1] = first;
}

int main()
{

```

```

int a[] = {1, 2, 3, 4};
rotateLeft(a, 4);
for (int i = 0; i < 4; i++)
    cout << a[i] << " ";
cout << endl;

char c[] = {'a', 'b', 'c'};
rotateLeft(c, 3);
for (int i = 0; i < 3; i++)
    cout << c[i] << " ";
cout << endl;
return 0;
}

```

Output:

```

2 3 4 1
b c a

```

Part B: Class Template Programs

Q9. Write a class template `MyStack` that uses the STL vector internally and has functions `push()`, `pop()`, `top()` and `size()`.

Solution:

```

#include <iostream>
#include <vector>
#include <string>
using namespace std;

template <class T>
class MyStack
{
    vector<T> v;
public:
    void push(T x) { v.push_back(x); }
    void pop() { v.pop_back(); }
    T top() { return v.back(); }
    bool empty() { return v.empty(); }
    int size() { return v.size(); }
};

int main()
{
    MyStack<int> s;
    s.push(10);
    s.push(20);
    s.push(30);
    cout << "Top: " << s.top() << endl;
    s.pop();
    cout << "Top: " << s.top() << endl;
    cout << "Size: " << s.size() << endl;

    MyStack<string> t;
    t.push("one");
    t.push("two");
    cout << t.top() << endl;
    return 0;
}

```

```
}
```

Output:

```
Top: 30  
Top: 20  
Size: 2  
two
```

Since `vector` grows automatically, this stack has no fixed size limit.

Q10. Write a class template `Complex` and overload the `+` and `*` operators. Use it with `int` and `double`.

Solution:

```
#include <iostream>
using namespace std;

template <class T>
class Complex
{
    T re, im;
public:
    Complex(T r = 0, T i = 0) { re = r; im = i; }

    Complex operator+(Complex c)
    {
        return Complex(re + c.re, im + c.im);
    }

    Complex operator*(Complex c)
    {
        return Complex(re * c.re - im * c.im, re * c.im + im * c.re);
    }

    void show()
    {
        cout << re << " + " << im << "i" << endl;
    }
};

int main()
{
    Complex<int> a(1, 2), b(3, 4);
    (a + b).show();
    (a * b).show();

    Complex<double> c(1.5, 0.5), d(2.0, 1.0);
    (c + d).show();
    return 0;
}
```

Output:

```
4 + 6i  
-5 + 10i  
3.5 + 1.5i
```

For $(1 + 2i)(3 + 4i)$ the real part is $1 \times 3 - 2 \times 4 = -5$ and the imaginary part is $1 \times 4 + 2 \times 3 = 10$.

Q11. Write a class template `KeyValue` with two different type parameters, one for the key and one for the value.

Solution:

```
#include <iostream>
#include <string>
using namespace std;

template <class K, class V>
class KeyValue
{
    K key;
    V value;
public:
    KeyValue(K k, V v) { key = k; value = v; }
    void show() { cout << key << " = " << value << endl; }
};

int main()
{
    KeyValue<string, int> a("Age", 21);
    KeyValue<int, double> b(1, 99.5);
    KeyValue<char, string> c('N', "Asha");
    a.show();
    b.show();
    c.show();
    return 0;
}
```

Output:

```
Age = 21
1 = 99.5
N = Asha
```

Q12. Write a simple smart pointer class template `SmartPtr` that deletes the memory automatically in its destructor. Overload the `*` and `->` operators.

Solution:

```
#include <iostream>
using namespace std;

template <class T>
class SmartPtr
{
    T *ptr;
public:
    SmartPtr(T *p) { ptr = p; }
    ~SmartPtr()
    {
        delete ptr;
        cout << "Memory freed" << endl;
    }
    T &operator*() { return *ptr; }
    T *operator->() { return ptr; }
}
```

```
};

struct Point
{
    int x, y;
};

int main()
{
    SmartPtr<int> a(new int(25));
    cout << *a << endl;

    Point *raw = new Point;
    raw->x = 3;
    raw->y = 4;
    SmartPtr<Point> p(raw);
    cout << p->x << " " << p->y << endl;
    return 0;
}
```

Output:

```
25
3 4
Memory freed
Memory freed
```

The two "Memory freed" lines appear when `main()` ends and the objects `p` and `a` go out of scope, so the memory is released without writing `delete` in the program.

Q13. Write a class template `Utility` that contains static member functions `square()` and `larger()`.

Solution:

```
#include <iostream>
using namespace std;

template <class T>
class Utility
{
public:
    static T square(T x) { return x * x; }
    static T larger(T a, T b) { return (a > b) ? a : b; }
};

int main()
{
    cout << Utility<int>::square(6) << endl;
    cout << Utility<double>::square(1.5) << endl;
    cout << Utility<char>::larger('a', 'z') << endl;
    return 0;
}
```

Output:

```
36
2.25
```

Static functions are called with the class name and scope resolution operator, without creating an object. The type must still be given in angle brackets.

Q14. Write a class template `Box`, and then a normal (non-template) class `IntBox` derived from `Box<int>` with an extra function.

Solution:

```
#include <iostream>
using namespace std;

template <class T>
class Box
{
protected:
    T v;
public:
    Box(T x) { v = x; }
    T get() { return v; }
};

class IntBox : public Box<int>
{
public:
    IntBox(int x) : Box<int>(x) { }
    int doubled() { return v * 2; }
};

int main()
{
    IntBox b(21);
    cout << b.get() << endl;
    cout << b.doubled() << endl;
    return 0;
}
```

Output:

```
21
42
```

A normal class can be derived from one particular instance of a class template. Since `IntBox` is not a template, it can use `v` directly, without `this->`.

Q15. Show that a class template can be instantiated with another template class as its type (nested templates).

Solution:

```
#include <iostream>
using namespace std;

template <class T>
class Box
{
    T v;
```

```

public:
    Box(T x) { v = x; }
    T get() { return v; }
};

int main()
{
    Box<int> inner(7);
    Box< Box<int> > outer(inner);
    cout << outer.get().get() << endl;
    return 0;
}

```

Output:

```
7
```

Here T of the outer box is Box<int>. So outer.get() returns a Box<int>, and its get() returns the integer 7. A space is kept between the closing angle brackets in older C++ standards.

Q16. Write a class template Counter and overload both the prefix and postfix ++ operators.

Solution:

```

#include <iostream>
using namespace std;

template <class T>
class Counter
{
    T val;
public:
    Counter(T v) { val = v; }

    Counter &operator++()          // prefix
    {
        ++val;
        return *this;
    }

    Counter operator++(int)        // postfix
    {
        Counter temp = *this;
        ++val;
        return temp;
    }

    T get() { return val; }
};

int main()
{
    Counter<int> a(5);
    ++a;
    a++;
    cout << a.get() << endl;

    Counter<double> b(1.5);
    ++b;
}

```

```
    cout << b.get() << endl;

    Counter<char> c('A');
    c++;
    cout << c.get() << endl;
    return 0;
}
```

Output:

```
7
2.5
B
```

The dummy `int` parameter is used only to tell the compiler that this is the postfix form.

Part C: Predict the Output

Q17. Predict the output of the following program.

Solution:

```
#include <iostream>
using namespace std;

template <class T>
class Show
{
public:
    void print() { cout << "General "; }
};

template <>
class Show<int>
{
public:
    void print() { cout << "Int "; }
};

int main()
{
    Show<int> a;
    Show<char> b;
    Show<double> c;
    a.print();
    b.print();
    c.print();
    cout << endl;
    return 0;
}
```

Output:

```
Int General General
```

Explanation: Only `Show<int>` has a special (specialized) definition. All other types use the general template.

Q18. Predict the output of the following program.

Solution:

```
#include <iostream>
#include <string>
using namespace std;

template <class T>
T twice(T x)
{
    return x + x;
}

int main()
{
    cout << twice(4) << " ";
    cout << twice(2.5) << " ";
    cout << twice(string("ab")) << endl;
    return 0;
}
```

Output:

```
8 5 abab
```

Explanation: For `int`, $4 + 4 = 8$. For `double`, $2.5 + 2.5 = 5.0$, printed as 5. For `string`, the `+` operator joins the strings, giving `abab`. The same template works for any type that supports `+`.

Q19. Predict the output of the following program.

Solution:

```
#include <iostream>
using namespace std;

template <class T, class U>
void f(T a, U b)
{
    cout << a << b << endl;
}

int main()
{
    f<int>(2.9, 7);
    f<double, int>(1, 2.5);
    return 0;
}
```

Output:

```
27
12
```

Explanation: In `f<int>(2.9, 7)`, `T` is forced to `int`, so 2.9 becomes 2, and `U` is deduced as `int`, giving 27. In `f<double, int>(1, 2.5)`, `T` is `double` so 1 is printed as 1, and `U` is `int` so 2.5 becomes 2, giving 12.

Part D: Find the Error

Q20. Find the error in the following code and correct it.

```
template <T>
T square(T x)
{
    return x * x;
}
```

Error: In the template parameter list, the keyword `class` or `typename` is missing, so the compiler does not know that `T` is a type parameter.

Correction:

```
template <class T>
T square(T x)
{
    return x * x;
}
```

Q21. The following program gives a linker error. Find the reason and correct it.

```
#include <iostream>
using namespace std;

template <class T>
class Counter
{
public:
    static int count;
};

int main()
{
    Counter<int>::count = 5;
    cout << Counter<int>::count << endl;
    return 0;
}
```

Error: The static data member `count` is declared inside the class template but is never defined outside it, so no storage exists for it and the linker reports an undefined reference.

Correction: define the static member outside the class, using the template header.

```
template <class T>
int Counter<T>::count = 0;
```