

Object Oriented Programming C++

Chapter: Templates

Coding Questions and Solutions (Set 6)

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 5.

Part A: Function Template Programs

Q1. Write a function template to sort an array in descending order using selection sort.

Solution:

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

template <class T>
void selectionSortDesc(T a[], int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        int maxIdx = i;
        for (int j = i + 1; j < n; j++)
            if (a[j] > a[maxIdx])
                maxIdx = j;
        T temp = a[i];
        a[i] = a[maxIdx];
        a[maxIdx] = temp;
    }
}

int main()
{
    int a[] = {4, 9, 1, 7};
    selectionSortDesc(a, 4);
    for (int i = 0; i < 4; i++)
        cout << a[i] << " ";
    cout << endl;

    double b[] = {2.5, 0.5, 3.5};
    selectionSortDesc(b, 3);
    for (int i = 0; i < 3; i++)
        cout << b[i] << " ";
    cout << endl;
    return 0;
}
```

Output:

```
9 7 4 1
3.5 2.5 0.5
```

Q2. Write a function template to find the dot product of two arrays.

Solution:

```

#include <iostream>
using namespace std;

template <class T>
T dotProduct(T a[], T b[], int n)
{
    T sum = 0;
    for (int i = 0; i < n; i++)
        sum += a[i] * b[i];
    return sum;
}

int main()
{
    int a[] = {1, 2, 3};
    int b[] = {4, 5, 6};
    cout << dotProduct(a, b, 3) << endl;

    double x[] = {0.5, 1.5};
    double y[] = {2.0, 4.0};
    cout << dotProduct(x, y, 2) << endl;
    return 0;
}

```

Output:

```

32
7

```

For the int arrays: $1 \times 4 + 2 \times 5 + 3 \times 6 = 32$. For the double arrays: $0.5 \times 2.0 + 1.5 \times 4.0 = 7$.

Q3. Write a function template to add two arrays element by element and store the result in a third array.

Solution:

```

#include <iostream>
using namespace std;

template <class T>
void addArrays(T a[], T b[], T c[], int n)
{
    for (int i = 0; i < n; i++)
        c[i] = a[i] + b[i];
}

int main()
{
    int a[] = {1, 2, 3};
    int b[] = {10, 20, 30};
    int c[3];
    addArrays(a, b, c, 3);
    for (int i = 0; i < 3; i++)
        cout << c[i] << " ";
    cout << endl;

    double x[] = {1.5, 2.5};
    double y[] = {0.5, 0.5};
    double z[2];

```

```

    addArrays(x, y, z, 2);
    for (int i = 0; i < 2; i++)
        cout << z[i] << " ";
    cout << endl;
    return 0;
}

```

Output:

```

11 22 33
2 3

```

Q4. Write a function template `countIf()` that counts the elements of an array that satisfy a condition given by a function pointer. Use it with a template function `isPositive()`.

Solution:

```

#include <iostream>
using namespace std;

template <class T>
bool isPositive(T x)
{
    return x > 0;
}

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

int main()
{
    int a[] = {3, -1, 4, -5, 9};
    double b[] = {-1.5, 2.5, -3.5};

    cout << countIf(a, 5, isPositive<int>) << endl;
    cout << countIf(b, 3, isPositive<double>) << endl;
    return 0;
}

```

Output:

```

3
1

```

The condition is passed as a function pointer, so the same `countIf()` can be used with different conditions.

Q5. Write a function template that finds the sum of an array without being given its size. The size must be deduced from the array itself.

Solution:

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

template <class T, int N>
T sumOf(T (&arr)[N])
{
    T sum = 0;
    for (int i = 0; i < N; i++)
        sum += arr[i];
    return sum;
}

int main()
{
    int a[] = {1, 2, 3, 4, 5};
    double b[] = {0.5, 1.5};

    cout << sumOf(a) << endl;
    cout << sumOf(b) << endl;
    return 0;
}
```

Output:

```
15
2
```

The parameter is a reference to an array, so the compiler deduces both the element type `T` and the array size `N` from the argument. No separate size argument is needed.

Q6. Write a function template with a non-type parameter `N`, which multiplies a value by `N`.

Solution:

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

template <class T, int N>
T scale(T x)
{
    return x * N;
}

int main()
{
    cout << scale<int, 5>(4) << endl;
    cout << scale<int, 3>(7) << endl;
    cout << scale<double, 2>(1.5) << endl;
    return 0;
}
```

Output:

```
20
21
3
```

N cannot be deduced from the arguments, so both template arguments are written explicitly in angle brackets.

Q7. Write two overloaded function templates `show()`: one for a normal value and one for a pointer. Show which one is called.

Solution:

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

template <class T>
void show(T x)
{
    cout << "Value: " << x << endl;
}

template <class T>
void show(T *p)
{
    cout << "Pointer to: " << *p << endl;
}

int main()
{
    int n = 9;
    double d = 2.5;
    show(5);
    show(&n);
    show(&d);
    return 0;
}
```

Output:

```
Value: 5
Pointer to: 9
Pointer to: 2.5
```

For `show(&n)` both templates could match (T as `int *` or T as `int`). The compiler chooses the more specialized template, the pointer version.

Q8. Write a function template `findBest()` that finds the best element of an array according to a comparison function passed as a parameter. Use it to find both the maximum and the minimum.

Solution:

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

template <class T>
bool isGreater(T a, T b) { return a > b; }

template <class T>
bool isSmaller(T a, T b) { return a < b; }

template <class T>
```

```

T findBest(T a[], int n, bool (*cmp)(T, T))
{
    T best = a[0];
    for (int i = 1; i < n; i++)
        if (cmp(a[i], best))
            best = a[i];
    return best;
}

int main()
{
    int a[] = {4, 9, 1, 7};
    cout << "Max: " << findBest(a, 4, isGreater<int>) << endl;
    cout << "Min: " << findBest(a, 4, isSmaller<int>) << endl;

    double b[] = {2.5, 0.5, 3.5};
    cout << "Max: " << findBest(b, 3, isGreater<double>) << endl;
    return 0;
}

```

Output:

```

Max: 9
Min: 1
Max: 3.5

```

Part B: Class Template Programs

Q9. Write a Stack class template using a linked list (no fixed size).

Solution:

```

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

template <class T>
class Stack
{
    struct Node
    {
        T data;
        Node *next;
        Node(T d, Node *n) { data = d; next = n; }
    };
    Node *topPtr;
public:
    Stack() { topPtr = NULL; }

    void push(T x) { topPtr = new Node(x, topPtr); }

    T pop()
    {
        if (topPtr == NULL)
        {
            cout << "Stack empty" << endl;
            return T();
        }
        Node *t = topPtr;
    }
}

```

```

        T v = t->data;
        topPtr = topPtr->next;
        delete t;
        return v;
    }

    bool isEmpty() { return topPtr == NULL; }

    ~Stack()
    {
        while (topPtr != NULL)
        {
            Node *t = topPtr;
            topPtr = topPtr->next;
            delete t;
        }
    }
};

int main()
{
    Stack<int> s;
    s.push(10);
    s.push(20);
    s.push(30);
    cout << s.pop() << endl;
    cout << s.pop() << endl;

    Stack<double> d;
    d.push(1.5);
    cout << d.pop() << endl;
    cout << d.pop() << endl;
    return 0;
}

```

Output:

```

30
20
1.5
Stack empty
0

```

Q10. Write a class template `Triple` with three different type parameters.

Solution:

```

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

template <class A, class B, class C>
class Triple
{
    A a;
    B b;
    C c;
public:
    Triple(A x, B y, C z) { a = x; b = y; c = z; }
    void show() { cout << a << ", " << b << ", " << c << endl; }
};

```

```

int main()
{
    Triple<int, char, double> t1(1, 'A', 2.5);
    Triple<string, int, float> t2("Ravi", 21, 8.5f);
    t1.show();
    t2.show();
    return 0;
}

```

Output:

```

1, A, 2.5
Ravi, 21, 8.5

```

Q11. Write a class template `Account` with functions to deposit, withdraw (with a balance check) and get the balance.

Solution:

```

#include <iostream>
using namespace std;

template <class T>
class Account
{
    T balance;
public:
    Account(T b) { balance = b; }

    void deposit(T x) { balance += x; }

    bool withdraw(T x)
    {
        if (x > balance)
        {
            cout << "Insufficient balance" << endl;
            return false;
        }
        balance -= x;
        return true;
    }

    T getBalance() { return balance; }
};

int main()
{
    Account<double> acc(1000.0);
    acc.deposit(500);
    acc.withdraw(2000);
    acc.withdraw(300);
    cout << "Balance: " << acc.getBalance() << endl;

    Account<int> a2(50);
    a2.withdraw(20);
    cout << "Balance: " << a2.getBalance() << endl;
    return 0;
}

```

Output:

```
Insufficient balance
Balance: 1200
Balance: 30
```

Q12. Write a class template `Pair` and overload the `==` operator to compare two pairs.

Solution:

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

template <class T>
class Pair
{
    T a, b;
public:
    Pair(T x, T y) { a = x; b = y; }

    bool operator==(const Pair &p) const
    {
        return a == p.a && b == p.b;
    }
};

int main()
{
    Pair<int> p1(1, 2), p2(1, 2), p3(2, 1);
    cout << (p1 == p2 ? "Equal" : "Not equal") << endl;
    cout << (p1 == p3 ? "Equal" : "Not equal") << endl;
    return 0;
}
```

Output:

```
Equal
Not equal
```

Q13. Write a class template `Shape<T>` with a pure virtual function `area()`. Derive `Circle` from `Shape<double>` and `Square` from `Shape<int>`, and call them through base class pointers.

Solution:

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

template <class T>
class Shape
{
public:
    virtual T area() = 0;
    virtual ~Shape() { }
};

class Circle : public Shape<double>
{
    double r;
```

```

public:
    Circle(double x) { r = x; }
    double area() { return 3.14 * r * r; }
};

class Square : public Shape<int>
{
    int s;
public:
    Square(int x) { s = x; }
    int area() { return s * s; }
};

int main()
{
    Shape<double> *a = new Circle(2);
    Shape<int> *b = new Square(5);
    cout << a->area() << endl;
    cout << b->area() << endl;
    delete a;
    delete b;
    return 0;
}

```

Output:

```

12.56
25

```

This program combines templates (the return type of `area()` is the template parameter) with run-time polymorphism (virtual functions). `Shape<double>` and `Shape<int>` are two different base classes.

Q14. Write a class template `Array` with a copy constructor that makes a deep copy. Show that changing the copy does not change the original.

Solution:

```

#include <iostream>
using namespace std;

template <class T>
class Array
{
    T *p;
    int n;
public:
    Array(int size)
    {
        n = size;
        p = new T[n];
        for (int i = 0; i < n; i++)
            p[i] = T();
    }

    Array(const Array &o)
    {
        n = o.n;
        p = new T[n];
    }
}

```

```

        for (int i = 0; i < n; i++)
            p[i] = o.p[i];
    }

    ~Array() { delete[] p; }

    void set(int i, T v) { p[i] = v; }

    void show()
    {
        for (int i = 0; i < n; i++)
            cout << p[i] << " ";
        cout << endl;
    }
};

int main()
{
    Array<int> a(3);
    a.set(0, 1);
    a.set(1, 2);
    a.set(2, 3);

    Array<int> b = a;
    b.set(1, 99);

    cout << "a: "; a.show();
    cout << "b: "; b.show();
    return 0;
}

```

Output:

```

a: 1 2 3
b: 1 99 3

```

Without the copy constructor, the default copy would copy only the pointer, and both objects would share the same memory. Then changing `b` would also change `a`.

Q15. Write a class template `Pair` with two type parameters and a function template `makePair()` that creates a `Pair` without the user writing the types.

Solution:

```

#include <iostream>
using namespace std;

template <class T1, class T2>
class Pair
{
    T1 first;
    T2 second;
public:
    Pair(T1 a, T2 b) { first = a; second = b; }
    void show() { cout << first << " " << second << endl; }
};

template <class T1, class T2>
Pair<T1, T2> makePair(T1 a, T2 b)

```

```

{
    return Pair<T1, T2>(a, b);
}

int main()
{
    makePair(1, 'A').show();
    makePair(2.5, 10).show();
    return 0;
}

```

Output:

```

1 A
2.5 10

```

A function template can deduce the types from its arguments, so it can be used to create objects of a class template without writing the types in angle brackets.

Q16. Write a class template `Sorter` that stores a copy of an array, sorts it using bubble sort and displays it.

Solution:

```

#include <iostream>
using namespace std;

template <class T>
class Sorter
{
    T *a;
    int n;
public:
    Sorter(T arr[], int size)
    {
        n = size;
        a = new T[n];
        for (int i = 0; i < n; i++)
            a[i] = arr[i];
    }

    ~Sorter() { delete[] a; }

    void sort()
    {
        for (int i = 0; i < n - 1; i++)
            for (int j = 0; j < n - i - 1; j++)
                if (a[j] > a[j + 1])
                {
                    T t = a[j];
                    a[j] = a[j + 1];
                    a[j + 1] = t;
                }
    }

    void display()
    {
        for (int i = 0; i < n; i++)
            cout << a[i] << " ";
        cout << endl;
    }
}

```

```

    }
};

int main()
{
    int x[] = {5, 3, 8, 1};
    Sorter<int> s1(x, 4);
    s1.sort();
    s1.display();

    char y[] = {'d', 'a', 'c'};
    Sorter<char> s2(y, 3);
    s2.sort();
    s2.display();
    return 0;
}

```

Output:

```

1 3 5 8
a c d

```

Part C: Predict the Output

Q17. Predict the output of the following program.

Solution:

```

#include <iostream>
using namespace std;

template <class T>
void counter()
{
    static int c = 0;
    c++;
    cout << c << " ";
}

int main()
{
    counter<int>();
    counter<int>();
    counter<char>();
    counter<int>();
    cout << endl;
    return 0;
}

```

Output:

```

1 2 1 3

```

Explanation: Each instantiation of a function template is a separate function with its own static variable. `counter<int>` is called three times (1, 2, 3) and `counter<char>` once (1), in the order of the calls.

Q18. Predict the output of the following program.

Solution:

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

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

int main()
{
    cout << minimum(5, 3);
    cout << minimum<double>(5, 3.5);
    cout << minimum('a', 'b');
    cout << endl;
    return 0;
}
```

Output:

```
33.5a
```

Explanation: minimum(5, 3) gives 3. In minimum<double>(5, 3.5) the type is forced to double, so 5 is converted to 5.0 and the smaller value 3.5 is returned. minimum('a', 'b') gives the character a. The three outputs are printed together as 33.5a.

Q19. Predict the output of the following program.

Solution:

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

template <class T>
class A
{
    T v;
public:
    A(T x) { v = x; cout << "C" << v << " "; }
    ~A() { cout << "D" << v << " "; }
};

int main()
{
    A<int> a(1);
    {
        A<char> b('x');
    }
    A<double> c(2.5);
    cout << "End ";
    return 0;
}
```

Output:

```
C1 Cx Dx C2.5 End D2.5 D1
```

Explanation: Object `b` is created and destroyed inside the inner block, so "Cx Dx" is printed immediately. Objects `c` and `a` are destroyed when `main()` ends, in the reverse order of their creation: first `c` (D2.5), then `a` (D1).

Part D: Find the Error

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

```
template <class T>
void show(T x) { cout << x << endl; }

void show<int>(int x) { cout << "Integer: " << x << endl; }
```

Error: An explicit specialization of a template must start with `template <>`. Without it, the compiler treats the second definition as an invalid declaration.

Correction:

```
template <>
void show<int>(int x) { cout << "Integer: " << x << endl; }
```

Q21. Find the error in the following code and correct it.

```
template <class T, double N>
class Scaler
{
    T value;
public:
    Scaler(T v) { value = v; }
    T get() { return value * N; }
};

int main()
{
    Scaler<int, 2.5> s(4);
    return 0;
}
```

Error: A non-type template parameter cannot be of floating-point type `double` in standard C++ (before C++20). Non-type parameters must be integral types (such as `int`, `char`, `bool`), enumerations or pointers.

Correction: use an integral non-type parameter, or pass the value through the constructor.

```
template <class T>
class Scaler
{
    T value;
    double factor;
public:
    Scaler(T v, double f) { value = v; factor = f; }
    double get() { return value * factor; }
};
```

```
// Scaler<int> s(4, 2.5);
```