```cpp
// C++ ADVANCED EXERCISES SOLUTIONS
// ==============================

#include <iostream>
#include <vector>
#include <functional>
#include <memory>
#include <type_traits>
#include <utility>

using namespace std;

// ============================================
// FIRST CODE FROM IMAGE 1 - LAMBDA EXERCISE
// ============================================

/*
template<class Functor>
vector<int> find_template(const vector<int>& v, Functor t) {
    vector<int> r;
    for(auto it = v.begin(); it != v.end(); ++it) if (t(*it))
r.push_back(*it);
    return r;
}

unsigned int find_1(const vector<int>& v,int k) {
    vector<int> w = find_template(v, [v,k](int n) { return n>k; } );
    return w.size();
}

vector<int> find_2(const vector<int>& v) {
    return find_template(v, [v](int n) { return n<v.size(); } );
}

vector<int> v1 = {3,6,4,6,2,5,-2,4,2}; vector<int> v2 =
{-2,-6,4,4,2,5,0,4,2,3,2,0};

Answer to the first image exercise:
cout << find_1(v1,2); // Output: 4 (elements in v1 that are > 2 are:
3,6,4,6,5,4)
cout << find_1(v2,2); // Output: 5 (elements in v2 that are > 2 are:
4,4,5,4,3)
cout << find_2(v1).size(); // Output: 8 (elements in v1 that are < v1.size()
[which is 9] are all except 9)
cout << find_2(v2).size(); // Output: 12 (all elements in v2 are < v2.size()
[which is 12])
*/
```

```
// ==========================================
// SECOND CODE FROM IMAGE 2 - RUNTIME ERROR
// ==========================================

/*
class A { public: virtual ~A(){} };
class B: public A {};
class C: public A {};
int main() {/* ... */ dynamic_cast<C&>(*pb);}

Answer to the second image exercise:
The solution creates a situation where a dynamic_cast will fail and throw a
std::bad_cast exception:

class A { public: virtual ~A(){} };
class B: public A {};
class C: public A {};
int main() {
    A* pa = new A; B* pb = new B; C* pc = new C;
    dynamic_cast<C&>(*pb); // Fails at runtime - can't cast B to C
}
*/

// ==========================================
// LAMBDA EXERCISE 1: CAPTURE AND MODIFICATION
// ==========================================

void lambda_exercise1() {
    cout << "\n=== Lambda Exercise 1 - Capture and Modification ===\n";

    int x = 10;
    int y = 20;

    auto l1 = [=]() mutable { x = 30; y = 40; return x + y; };
    auto l2 = [&]() { x = 50; y = 60; return x + y; };

    cout << l1() << endl;  // Output: 70 (30 + 40, but only in lambda's
copy)
    cout << "x: " << x << ", y: " << y << endl;  // Output: x: 10, y: 20
(unchanged)

    cout << l2() << endl;  // Output: 110 (50 + 60, modifies original
variables)
    cout << "x: " << x << ", y: " << y << endl;  // Output: x: 50, y: 60
(changed)

    // Explanation:
    // l1 captures by value (=) and uses 'mutable' to allow modifying its
copies
```

```
    // However, the original x and y remain unchanged
    //
    // l2 captures by reference (&) and directly modifies the original
variables
}

// ============================================
// LAMBDA EXERCISE 2: RECURSION WITH LAMBDA
// ============================================

void lambda_exercise2() {
    cout << "\n=== Lambda Exercise 2 - Recursion with Lambda ===\n";

    std::function<int(int)> factorial = [&factorial](int n) {
        return n <= 1 ? 1 : n * factorial(n - 1);
    };

    cout << "Factorial of 5: " << factorial(5) << endl;  // Output: 120

    auto fibonacci = [](int n) {
        std::function<int(int)> fib = [&fib](int n) {
            return n <= 1 ? n : fib(n-1) + fib(n-2);
        };
        return fib(n);
    };

    cout << "Fibonacci of 7: " << fibonacci(7) << endl;  // Output: 13

    // Explanation:
    // The factorial lambda captures a reference to itself to allow
recursion
    // It calculates 5! = 5 * 4 * 3 * 2 * 1 = 120
    //
    // The fibonacci lambda creates a nested lambda that captures itself
    // Fib(7) = Fib(6) + Fib(5) = 8 + 5 = 13
}

// ============================================
// TRUE OR FALSE EXERCISE 3: INHERITANCE AND POLYMORPHISM
// ============================================

void true_or_false_inheritance() {
    cout << "\n=== True or False Exercise 3 - Inheritance and Polymorphism
===\n";

    cout << "1. dynamic_cast<D*>(nullptr) returns nullptr even if D is not
polymorphic: TRUE\n";
    // dynamic_cast on nullptr returns nullptr regardless of the target type

    cout << "2. A pointer to derived class can be assigned to base class
```

```
pointer without cast: TRUE\n";
    // This is standard upcasting, always allowed

    cout << "3. If base class has virtual destructor, all derived classes
must redefine it: FALSE\n";
    // Derived classes inherit the base class's virtual destructor
automatically

    cout << "4. typeid on non-polymorphic types checks only the static type:
TRUE\n";
    // typeid requires RTTI for dynamic type checking, otherwise uses static
type

    cout << "5. In diamond inheritance with virtual base, base constructor
called once: TRUE\n";
    // Virtual base classes are constructed exactly once by the most derived
class

    cout << "6. sizeof a class returns sum of member sizes plus padding:
TRUE\n";
    // sizeof includes members, padding, and vtable pointers if applicable

    cout << "7. A method can be both final and pure virtual at the same
time: FALSE\n";
    // A pure virtual method is meant to be overridden, while final prevents
overriding

    cout << "8. dynamic_cast<A*>(new C()) is always valid if C inherits from
A through B: TRUE\n";
    // Upcasting to any base class is always valid

    cout << "9. A const method can call non-const methods of the same class:
FALSE\n";
    // const methods can only call other const methods on the same object

    cout << "10. static_cast can convert a void pointer to any type:
TRUE\n";
    // static_cast can convert void* to any pointer type (but the programmer
is responsible for correctness)
}

// =============================================
// TRUE OR FALSE EXERCISE 4: TEMPLATES AND STL
// =============================================

void true_or_false_templates() {
    cout << "\n=== True or False Exercise 4 - Templates and STL ===\n";

    cout << "1. Function templates support automatic type deduction, class
templates don't: TRUE\n";
```

```cpp
    // Before C++17's class template argument deduction, this was true

    cout << "2. std::vector<int>::iterator is guaranteed to be a pointer to int: FALSE\n";
    // The iterator type is implementation-defined, not guaranteed to be a raw pointer

    cout << "3. std::move physically moves data from one object to another: FALSE\n";
    // std::move is just a cast to rvalue reference, doesn't actually move anything

    cout << "4. An STL container can contain references: FALSE\n";
    // C++ containers require elements to be assignable and copyable

    cout << "5. Iterators of std::map point to std::pair<const Key, Value>: TRUE\n";
    // map iterators point to key-value pairs with const keys

    cout << "6. std::shared_ptr supports arrays of objects: TRUE (since C++17)\n";
    // C++17 added support for arrays in shared_ptr

    cout << "7. A lambda with no captures can be converted to a function pointer: TRUE\n";
    // Capture-less lambdas can be converted to function pointers

    cout << "8. emplace_back is always more efficient than push_back: FALSE\n";
    // emplace_back can be more efficient but isn't guaranteed to be

    cout << "9. std::function can store any callable with the correct signature: TRUE\n";
    // That's exactly what std::function is designed for

    cout << "10. std::unique_ptr can be used as a key in std::map: FALSE\n";
    // unique_ptr is movable but not copyable, and map keys need to be copyable
}

// ============================================
// PROBLEMATIC HIERARCHY EXERCISE 5: AMBIGUITY
// ============================================

// Original problematic code with ambiguity
class Base {
public:
    virtual void foo() { cout << "Base::foo" << endl; }
    int x = 10;
};
```

```cpp
class Derived1 : public Base {
public:
    void foo() override { cout << "Derived1::foo" << endl; }
    int x = 20;
};

class Derived2 : public Base {
public:
    void foo() override { cout << "Derived2::foo" << endl; }
    int x = 30;
};

// Original problematic Diamond class
/*
class Diamond : public Derived1, public Derived2 {
public:
    void foo() override { cout << "Diamond::foo" << endl; }
};
*/

// Fixed version with virtual inheritance
class FixedDerived1 : virtual public Base {
public:
    void foo() override { cout << "FixedDerived1::foo" << endl; }
    int x = 20;
};

class FixedDerived2 : virtual public Base {
public:
    void foo() override { cout << "FixedDerived2::foo" << endl; }
    int x = 30;
};

class FixedDiamond : public FixedDerived1, public FixedDerived2 {
public:
    void foo() override { cout << "FixedDiamond::foo" << endl; }

    // Resolve x ambiguity
    using FixedDerived1::x; // Choose one of the derived class x values
    // Alternatively, define a new x:
    // int x = 40;
};

void problematic_hierarchy() {
    cout << "\n=== Problematic Hierarchy Exercise 5 - Ambiguity ===\n";

    cout << "Problems in the original hierarchy:\n";
    cout << "1. Diamond inherits Base twice, creating two Base
subobjects\n";
```

```
    cout << "2. Ambiguous reference to x since two versions exist\n";
    cout << "3. Cannot convert Diamond* to Base* because of ambiguity\n\n";

    FixedDiamond d;
    d.foo();  // Calls FixedDiamond::foo
    cout << "Fixed Diamond's x: " << d.x << endl;  // Now works, prints 20

    // This now works because of virtual inheritance
    Base* b = &d;
    b->foo();  // Calls FixedDiamond::foo through virtual dispatch

    cout << "Solution: Use virtual inheritance to ensure only one Base
instance\n";
}

// =============================================
// PROBLEMATIC HIERARCHY EXERCISE 6: PRIVATE INHERITANCE
// =============================================

class BaseEx6 {
public:
    virtual void foo() { cout << "BaseEx6::foo" << endl; }
    void bar() { cout << "BaseEx6::bar" << endl; }
};

class DerivedEx6 : private BaseEx6 {
public:
    void foo() override { cout << "DerivedEx6::foo" << endl; }
    void bar() { cout << "DerivedEx6::bar" << endl; }
    using BaseEx6::foo;  // Makes BaseEx6::foo accessible through DerivedEx6
    // Note: This doesn't change the access of the overridden foo()
};

void private_inheritance() {
    cout << "\n=== Problematic Hierarchy Exercise 6 - Private Inheritance
===\n";

    DerivedEx6 d;
    d.foo();  // Calls DerivedEx6::foo
    d.bar();  // Calls DerivedEx6::bar

    cout << "Explanations:\n";
    cout << "1. DerivedEx6::foo compiles and calls the overridden method\n";
    cout << "2. DerivedEx6::bar compiles and calls the derived method\n";
    cout << "3. BaseEx6* b = &d; DOESN'T COMPILE because BaseEx6 is private
base\n";
    cout << "4. BaseEx6& r = d; DOESN'T COMPILE for the same reason\n";
    cout << "5. 'using BaseEx6::foo;' makes the base method accessible but
doesn't\n";
    cout << "   change the access level for inheritance purposes\n";
```

```cpp
}

// ===========================================
// ADVANCED CONCEPT EXERCISE 7: CRTP
// ===========================================

template <typename Derived>
class BaseEx7 {
public:
    void interface() {
        static_cast<Derived*>(this)->implementation();
    }

    void implementation() {
        cout << "BaseEx7 implementation" << endl;
    }
};

class DerivedEx7 : public BaseEx7<DerivedEx7> {
public:
    void implementation() {
        cout << "DerivedEx7 implementation" << endl;
    }
};

void crtp_pattern() {
    cout << "\n=== Advanced Concept Exercise 7 - CRTP ===\n";

    DerivedEx7 d;
    d.interface();  // Calls DerivedEx7::implementation

    BaseEx7<DerivedEx7>* b = &d;
    b->interface();  // Calls DerivedEx7::implementation
    b->implementation();  // Calls BaseEx7::implementation

    cout << "\nExplanation of CRTP pattern:\n";
    cout << "1. CRTP (Curiously Recurring Template Pattern) enables static polymorphism\n";
    cout << "2. It allows a base class to use functionality from a derived class at compile-time\n";
    cout << "3. Benefits include no virtual function overhead and compile-time binding\n";
    cout << "4. Common uses: static polymorphism, mixins, and method chaining in builder patterns\n";
}

// ===========================================
// ADVANCED CONCEPT EXERCISE 8: SFINAE
// ===========================================
```

```cpp
class NonCopyable {
public:
    NonCopyable() = default;
    NonCopyable(const NonCopyable&) = delete;
    NonCopyable& operator=(const NonCopyable&) = delete;

    void use() { std::cout << "NonCopyable used" << std::endl; }
};

template <typename T, typename... Args>
std::enable_if_t<std::is_constructible<T, Args...>::value, T*>
create(Args&&... args) {
    return new T(std::forward<Args>(args)...);
}

template <typename T, typename... Args>
std::enable_if_t<!std::is_constructible<T, Args...>::value, T*>
create(Args&&... args) {
    std::cout << "Cannot construct T with given arguments" << std::endl;
    return nullptr;
}

void sfinae_example() {
    cout << "\n=== Advanced Concept Exercise 8 - SFINAE ===\n";

    auto* nc1 = create<NonCopyable>();
    if (nc1) nc1->use();

    NonCopyable src;
    auto* nc2 = create<NonCopyable>(src);  // Attempting copy
    if (nc2) nc2->use();

    delete nc1;
    delete nc2;  // Safe even if nullptr

    cout << "\nExplanation of SFINAE:\n";
    cout << "1. SFINAE (Substitution Failure Is Not An Error) enables 
template specialization based on properties\n";
    cout << "2. enable_if selects between two implementations based on a 
compile-time condition\n";
    cout << "3. First call succeeds because NonCopyable is default-
constructible\n";
    cout << "4. Second call fails because NonCopyable is not copy-
constructible\n";
    cout << "5. This technique allows for compile-time introspection and API 
customization\n";
}

// ============================================
// MAIN FUNCTION TO RUN ALL EXERCISES
```

```cpp
// ===========================================

int main() {
    cout << "C++ ADVANCED EXERCISES SOLUTIONS\n";
    cout << "================================\n";

    lambda_exercise1();
    lambda_exercise2();
    true_or_false_inheritance();
    true_or_false_templates();
    problematic_hierarchy();
    private_inheritance();
    crtp_pattern();
    sfinae_example();

    return 0;
}
```