C++ Programming

Problems and Solutions

Problem 1

a) Answer: 1

Static typing in C++ means that the types of expressions are checked at compile-time. This means that the compiler checks the type compatibility of expressions during compilation, which helps catch type-related errors early.

b) Answer: 3

The C++ preprocessor is a tool that performs text replacement in C++ source code before it is compiled. It is used to modify the source code, include header files, define constants, and perform other operations that are not part of the C++ language.

c) Answer: 1

A build system like CMake simplifies the compilation of project sources and its dependencies by automating the build process. It generates platform-specific build scripts, which can be used to build the project. CMake can also be used to create reproducible, platform-independent builds.

d) Answer: 2

A typical C++ compiler translates and optimizes valid C++ source code to platform-specific machine code. It also checks the syntax of the source code and performs static analysis to detect bugs and other issues.

e) Answer: 3

The linker is responsible for combining the compiled object files into a single executable or shared library. As it does so, it resolves any references to external symbols, which are functions or variables that are defined in one translation unit and used in another. The linker ensures that the symbol addresses are properly linked so that the program can execute correctly.

f) Answer: 1

Some ODR violations may not be caught by either the compiler or linker. However, both the compiler and linker can catch some ODR violations. Therefore, neither the compiler alone nor the linker alone can catch all ODR violations.

Problem 2

a) Answer: CPP supports so many programming paradigms that developers can use during writing programs,

- Imperative/Procedural programming: This paradigm involves specifying a sequence of commands to be executed by the computer, with an emphasis on how the program does things rather than what it does.

- Object-oriented programming (OOP): This paradigm involves organizing code into objects, which can contain data and functions that operate on that data. OOP emphasizes encapsulation, inheritance, and polymorphism.
- Functional programming: This paradigm involves writing code as a series of functions that take input and produce output without modifying state or causing side effects. It emphasizes immutability and referential transparency.
- Generic programming: This paradigm involves writing code that works with a variety of types, rather than being tied to a specific type. C++ supports generic programming through templates.
- Event-driven programming: This paradigm involves responding to user or system events, such as button clicks or network messages, by triggering functions or other code in response. C++ supports event-driven programming through libraries and frameworks.
- Concurrent programming: This paradigm involves writing code that runs in parallel with other code, often on different threads or processes. C++ supports concurrent programming through multithreading and other mechanisms.
- Meta-programming: This paradigm involves writing code that manipulates or generates other code, often at compile-time rather than run-time. C++ supports meta-programming through template metaprogramming and other techniques.

b) Answer: Here is the one-line example of each,

```
extern int x; // declaration of x
int x; // definition of x
x = 20; // initialization of x
```

The complete program is as follows

```
#include <iostream>

extern int x; // declaration of x

int main() {
    int x; // definition of x
    x = 20; // initialization of x

    std::cout << x << std::endl;
    return 0;
}
```

c) Answer:

```
int x{getValueOfX()};
```

This will yield a compiler error if an implicit conversion potentially results a data loss. Thus, safe initialization saves potential data loss showing an error.

d) Answer:

- struct: default member variables and functions are public.
- class: default member variables and functions are private.

e) Answer:

- const: evaluated at runtime
- constexpr: evaluated at compile-time

Problem 3

a) Answer:

- prvalue: prvalue is a pure rvalue and it should have a persistent memory address. for example '`42`', '`true`' or '`3 + 5`' are prvalues
- xvalue: an xvalue or 'expiring value' is a vlaue that is about to be moved from and therefore, it's resource can be reused. for example

```
int x = 5;
int y = std::move(x); // x is an xvalue here
```

- glvalue stands for generalized lvalue. It is an expression that refers to an object or a function. for example,

```
int x = 42; // x is a glvalue here
int y = x + 3; // x+3 is a prvalue here
```

b) Answer:

```
edagdf
```

Problem 4

a) Answer:

- In the given code, cat is an lvalue because it is a named variable. However, when cat is passed to std::move on the next line, it is cast to an rvalue and the resulting expression is an rvalue expression. When feed is called with std::move(cat), the rvalue expression is passed as an argument to feed, and a becomes an rvalue reference to the argument.
- in short, In line 2, a is an rvalue reference parameter, which can bind to an rvalue argument passed to feed() using std::move().

b) Answer:

A single wrapper function that uses a reference parameter cannot always maintain the value category of the original argument when forwarding it to another function. Because, a reference parameter always has an lvalue type. On the other hand, a wrapper function that uses an rvalue reference parameter can only accept rvalues as arguments, not lvalues. That is why, two wrapper functions should be developed with overloading one another where one will receive lvalue and another will receive rvalue.

c) Answer:

```
template<typename T>
void api_function(T& a); // receives lvalue reference
void api_function(T&& a); // receives rvalue

void wrapper(arg&& a) {
    a.update();
    if constexpr(std::is_lvalue_reference_v<decltype(a)>) {
        api_function(a);
    } else {
        api_function(std::move(a));
    }
}
```

Problem 5

a) Answer:

The idea of RAII is when an object is created, the resource is acquired, and when the object is destroyed, the resource is released. This guarantees that resource is always properly released, even in the face of exception, early returns and other kinds of control flow. Example of RAII implementation,

```
int main() {
    int* raw_ptr = new int{20};

    std::unique_ptr<int> uniq_ptr(raw_ptr);
    std::cout << *uniq_ptr << std::endl;

    return 0;
}
```

b) Answer:

```
class Foo {
public:
    // Default constructor
    Foo();

    // Copy constructor
    Foo(const Foo& other);
```

```
    // Move constructor
    Foo(Foo&& other);

    // Copy assignment operator
    Foo& operator=(const Foo& other);

    // Move assignment operator
    Foo& operator=(Foo&& other);

    // Destructor
    ~Foo();
};
```

c) Answer:

he "Rule of Zero" is a guideline in C++ programming that suggests that when designing a class, you should either explicitly declare or define all of the special member functions (copy constructor, copy assignment, move constructor, move assignment, and destructor), or avoid defining any of them altogether.

d) Answer:

std::shared_ptr is generally considered less performant than std::unique_ptr due to the overhead of reference counting. Each shared_ptr maintains a reference count and a pointer to the resource it manages, and when copied, the reference count is incremented. When a shared_ptr is destroyed, the reference count is decremented, and if the count becomes zero, the resource is deleted. This reference counting process involves synchronization which can add significant overhead in multithreaded scenarios.

In contrast, std::unique_ptr does not maintain a reference count, and simply owns the resource it manages. When a unique_ptr is moved, the ownership of the resource is transferred, without any reference counting or synchronization overhead.

Problem 6

a) Answer

The console output is

```
create Vehicle
create Bike
create Vehicle
create Car
ring ring
honk honk
destroy Vehicle
destroy Vehicle
```

b)

Destructor of the Vehicle class was not declared as virtual.

```
virtual ~Vehicle() { std::cout<<"destroy Vehicle\n"; }
```

By declaring this destructor as virtual, the destructors of `Bike` and `Car` will be called correctly when the `unique_ptr` objects go out of the scope.

Now, the outptu is,

```
create Vehicle
create Bike
create Vehicle
create Car
ring ring
honk honk
destroy Car
destroy Vehicle
destroy Bike
destroy Vehicle
```

Problem 7

a)

A lambda expression in C++ is an anonymous function object that can be defined at the point of use.

b)

Problem: the lambda function captures the reference of the variables from the environment. Thus, it adds input with the address of the base_number.

Solution: use '=' instead of '&' inside the capture block as [=]. code:

```
return [=] (int input) {return base_number + input;};
```

c) Answer:

```
type_r fun(type_a a, type_b& b, type_c c) {
    return a + b + c;
}
```

Problem 8

a) Answer:

parameter packs are template parameters that accept zero or more arguments. templates with at least one parameter packs are called variadic template.

In another words: Variadic templates are templates that accepsts zero or more template arguments.

b) Answer:

```
template<typename R, typename... T>
R reduce_sum(const T&... nums) {
    return (nums + ...);
}
```

c) Answer:

Concepts: Concepts are language features that allow us to specify requirements on template parameters, thereby restricting what types can be used as template arguments.

Requires clauses: Requires clauses are expressions that can be used to specify requirements for template parameters as part of a function signature or a variable declaration.

d) Answer:

In C++20, we can define a concept called EqualityComparable to validate if a type supports equality checks using operator== and operator!=. Here's an example of how to define this concept:

```
template<typename T>
concept EqualityComparable = requires(T a, T b) {
    { a == b } -> std::convertible_to<bool>;
    { a != b } -> std::convertible_to<bool>;
};
```

This concept requires that the type T supports the == and != operators, and that these operators return a value that can be converted to bool. To use this concept, we can simply add it as a constraint to a function or template definition, like this:

```
template<typename T>
requires EqualityComparable<T>
bool are_equal(T a, T b) {
    return a == b;
}
```

Problem 9

```
#include <iostream>
#include <memory>
```

```
template<unsigned N>
constexpr unsigned fibonacci () {
    if constexpr (N >= 2)
        return fibonacci<N-1>() + fibonacci<N-2>();
    else
        return N;
}

int main() {
    std::cout<<fibonacci<5>()<<std::endl;

    return 0;
}
```

Another implementation

```
#include <iostream>

template <int N>
struct Fibonacci {
    static constexpr int value = Fibonacci<N - 1>::value + Fibonacci<N -
2>::value;
};

template <>
struct Fibonacci<0> {
    static constexpr int value = 0;
};

template <>
struct Fibonacci<1> {
    static constexpr int value = 1;
};

int main() {
  constexpr int fib10 = Fibonacci<10>::value;
  std::cout << "Fibonacci(10) = " << fib10 << std::endl;
  return 0;
}
```

Problem 10

a) Answer: 3

A friend in C++ is a non-member function or class that is granted access to the private and protected members of a class.

b) Answer: 2

Polymorphic cloning is used to create a new object of a derived class, but with the same properties and behaviors as the original object, including its complete inheritance hierarchy.

c) Answer: 1

The typical operation of a std::map, including insertion, deletion, and searching, has a guaranteed time complexity of O(log n).

d) Answer: 1

If you want to read-access a large object of type T, you should use a const reference to avoid copying the object.

e) Answer: 3

consteval is a new keyword in C++20 that guarantees a function to be executed at compile-time. A consteval function is required to be evaluated at compile-time by the compiler and it is an error if it is not possible to evaluate the function at compile-time. This can help improve performance by allowing for more computations to be done at compile-time rather than run-time.

f) Answer: 2

Expression templates are a technique in C++ to enable lazy evaluation of arithmetic expressions at compile-time, which can lead to significant performance improvements.

g) Answer: 1

The Curiously Recurring Template Pattern (CRTP) is a technique in C++ to provide static polymorphism, which is a form of compile-time polymorphism that can provide some of the benefits of dynamic polymorphism without the overhead of virtual functions.