Lecture 18 Multi-threading

#CS106L

Multi-threading

Further C++ reading:

Accelerated C++

Effective C++

Effective Modern C++

Exceptional C++

Modern C++ Design

Q1:template specialization?

In C++, template specialization is a feature that allows you to create specific versions of a template for certain types or values. This allows you to customize the behavior of a template for specific cases, while still maintaining the general behavior for other cases.

You can create a specialized version of a template by defining a new template with the same name, but with a different set of template arguments. The specialized version will be used in place of the general version whenever the template arguments match the specialized version's arguments.

Here's an example of a simple template class `MyClass` and a specialized version for type `int`:

```
template <typename T>
class MyClass {
public:
    void printValue(T value) {
        cout << "MyClass: " << value << endl;
```

```
    }
};

template<>
class MyClass<int> {
public:
    void printValue(int value) {
        cout << "MyClass (int): " << value << endl;
    }
};

int main() {
    MyClass<double> d;
    d.printValue(3.14); // Outputs "MyClass: 3.14"

    MyClass<int> i;
    i.printValue(42); // Outputs "MyClass (int): 42"
}
```

Q2:multi-threading?

Multithreading is a method of concurrent execution of two or more parts of a program called threads. Each thread has its own program counter and stack, but they share the same memory space with the main thread and can therefore share information or communicate with each other (through synchronization primitives such as mutexes and semaphores) more easily than if they were separate processes.

Multithreading can be used to improve the performance and responsiveness of a program. For example, a program that performs a long-running computation can use a separate thread to keep the user interface responsive. Similarly, a program that performs a network operation can use a separate thread to keep the user interface responsive while the network operation is in progress.

Additionally, multithreading can also be used to improve the scalability of a program by allowing it to take advantage of multiple processors or cores.

C++11 standard introduced threading support in the form of the `std::thread` class and related classes, and C++17 standard has added more functionality like Parallel Algorithm, atomic operation and shared mutex.

It's worth noting that multithreading can introduce complexity and bugs into a program, such as race conditions and deadlocks, so it's important to use

synchronization primitives and design patterns such as thread-safe queues and thread pools to mitigate these issues.

Q3:meta functions?

In C++, a meta function is a function that is defined as a template, and its behavior is determined by the template arguments passed to it. Meta functions are a type of template metaprogramming, and they are used to perform compile-time computations on types or values.

A common use of meta functions is to perform type trait analysis, which is used to determine the properties of a type at compile-time. For example, you can use a meta function to check if a type is a pointer or if it is a class type.

You can also use meta functions to perform compile-time computations on values. For example, you can use a meta function to compute the factorial of a number at compile time or to check if a value is a prime number.

Here is an example of a meta function to check if a type is a pointer or not:

```
template<typename T>

struct is_pointer {

    static constexpr bool value = false;

};

template<typename T>

struct is_pointer<T*> {

    static constexpr bool value = true;

};

int main() {

    static_assert(is_pointer<int*>::value,"int* is a pointer type");

    static_assert(!is_pointer<int>::value,"int is not a pointer type");

}
```

Q4:template meta programming?

Template Meta Programming (TMP) is a technique in C++ that allows you to perform computations at compile-time using templates. This is achieved by defining templates that are instantiated with different types or values and using the resulting types or values to perform computations or generate code.

One common use of TMP is to perform compile-time computations on types, such as computing the size of an array or the type of a member of a class. For example, you can use TMP to implement a compile-time factorial function as follows:

```
template<int N>

struct Factorial {

    enum { value = N * Factorial<N-1>::value };

};

template<>

struct Factorial<0> {

    enum { value = 1 };

};

constexpr int factorial = Factorial<5>::value; // factorial is 120
```

Another use of TMP is to generate code based on types or values. For example, you can use TMP to implement a type-safe enum as follows:

```
template<typename T, T Value>

struct Enum {

    static constexpr T value = Value;

};

using Color = Enum<int, 0>;

using Shape = Enum<int, 1>;
```

```
Color color = Color::value; // color is 0

Shape shape = Shape::value; // shape is 1
```

TMP can also be used to implement design patterns such as the visitor pattern, where different types of objects can be handled differently.