CS2810 OOAIA

Type Traits

Example Code Snippets

1. Checking Integral Types

```
#include <iostream>
#include <type_traits>

template<typename T>
void check_integral() {
    if (std::is_integral<T>::value) {
        std::cout << "Type is integral\n";
    } else {
        std::cout << "Type is NOT integral\n";
    }
}

int main() {
    check_integral<int>();      // Output: Type is integral
    check_integral<double>();   // Output: Type is NOT integral
    check_integral<char>();     // Output: Type is integral
    return 0;
}
```

2. Compile-time Assertion for Pointer Types

```
#include <type_traits>

template<typename T>
void process_pointer(T* ptr) {
    static_assert(std::is_pointer<T*>::value, "Must pass a pointer
type");
    // Process the pointer
}

int main() {
    int x = 10;
    process_pointer(&x);  // OK
    // process_pointer(x); // Compile error
    return 0;
}
```

3. Conditional Functionality Based on Type

```
#include <iostream>
#include <type_traits>

template<typename T>
void process(T value) {
    if constexpr (std::is_floating_point<T>::value) {
        std::cout << "Processing float: " << value * 1.5 << "\n";
    } else if constexpr (std::is_integral<T>::value) {
        std::cout << "Processing integer: " << value * 2 << "\n";
    } else {
        std::cout << "Unsupported type\n";
    }
}

int main() {
    process(5);      // Output: Processing integer: 10
    process(3.14);   // Output: Processing float: 4.71
    process("test"); // Output: Unsupported type
    return 0;
```

```
}
```

4. Checking Array Dimensions

```
#include <iostream>
#include <type_traits>

int main() {
    std::cout << "int rank: " << std::rank<int>::value << "\n";
    std::cout << "int[10] rank: " << std::rank<int[10]>::value <<
"\n";
    std::cout << "int[10][20] rank: " <<
std::rank<int[10][20]>::value << "\n";

    // Output:
    // int rank: 0
    // int[10] rank: 1
    // int[10][20] rank: 2

    return 0;
}
```

5. Type Comparison with is_same

```
#include <iostream>
#include <type_traits>

int main() {
    std::cout << std::boolalpha;
    std::cout << "int vs float: " << std::is_same<int, float>::value
<< "\n";
    std::cout << "int vs int: " << std::is_same<int, int>::value <<
"\n";
    std::cout << "int vs const int: " << std::is_same<int, const
int>::value << "\n";
```

```
    // Output:
    // int vs float: false
    // int vs int: true
    // int vs const int: false

    return 0;
}
```

6. Type-Aware Serialization Engine (practical, real-world example)

```
/*
Works for primitive types (int, float, etc.) → direct byte copy.
Handles STL containers (vector, list) → serialize each element.
Rejects unsafe types (pointers, non-POD classes) → compile-time
error.
Supports custom classes (if they define a serialize() method).
*/

#include <iostream>
#include <vector>
#include <type_traits>
#include <list>

// ------ Type Traits ------
template <typename T>
struct is_contiguous_container : std::false_type {};

template <typename T>
struct is_contiguous_container<std::vector<T>> : std::true_type {};

// Custom class check (SFINAE)
template <typename T>
class has_serialize_method {
    template <typename U>
    static auto test(U* u) -> decltype(u->serialize(),
std::true_type{});
    static auto test(...) -> std::false_type;
```

```
public:
    static constexpr bool value = decltype(test((T*)nullptr))::value;
};

// ------ Serializer ------
template <typename T>
void serialize(const T& data) {
    if constexpr (std::is_arithmetic_v<T>) {
        std::cout << "Serializing primitive: " << data << " ("
                  << sizeof(T) << " bytes)\n";
    }
    else if constexpr (is_contiguous_container<T>::value) {
        std::cout << "Serializing vector (size=" << data.size() <<
"): [ ";
        for (const auto& item : data) std::cout << item << " ";
        std::cout << "]\n";
    }
    else if constexpr (has_serialize_method<T>::value) {
        std::cout << "Custom class serialization: ";
        data.serialize();
    }
    else {
        static_assert(sizeof(T) == 0, "Unsupported type for
serialization!");
    }
}

// ------ Usage ------
class MyClass {
public:
    void serialize() const { std::cout << "MyClass data\n"; }
};

int main() {
    serialize(42);                          // Primitive
    serialize(std::vector{1, 2, 3});       // Container
    serialize(MyClass{});                   // Custom class
    // serialize(std::list{1, 2, 3});      // Error: Unsupported type
}
```

```
/*
Output:
Serializing primitive: 42 (4 bytes)
Serializing vector (size=3): [ 1 2 3 ]
Custom class serialization: MyClass data
*/
```

Type Trait	Description	Example
std::is_integral<T>	Checks if T is an integral type	is_integral<int>::value → true
std::is_floating_point<T>	Checks if T is a floating-point type	is_floating_point<double>::value → true
std::is_pointer<T>	Checks if T is a pointer type	is_pointer<int*>::value → true
std::is_array<T>	Checks if T is an array type	is_array<int[10]>::value → true
std::rank<T>	Gets the dimension of an array	rank<int[10][20]>::value → 2
std::is_same<T, U>	Checks if T and U are the same type	is_same<int, float>::value → false
std::is_const<T>	Checks if T is const-qualified	is_const<const int>::value → true
std::is_class<T>	Checks if T is a class/struct type	is_class<std::string>::value → true

Practice Problems

1. Problem: Write a function `print_type_info(T value)` that:

- Prints "Integral type" if `T` is an integral type.
- Prints "Floating-point type" if `T` is a floating-point type.
- Fails to compile for all other types (using `std::enable_if` or `static_assert`).

Example:

```
print_type_info(42);       // Output: "Integral type"
print_type_info(3.14);     // Output: "Floating-point type"
print_type_info("hello");  // Fails to compile
```

2. Problem: Create a function `safe_add(T a, T b)` that:

- Performs addition if `T` is an arithmetic type (integral or floating-point).
- Fails at compile-time if `T` is not arithmetic.
- Uses `static_assert` with a custom error message.

Example:

```
safe_add(5, 3);       // OK, returns 8
safe_add(2.5, 1.5);   // OK, returns 4.0
safe_add("a", "b");   // Compile error: "Non-arithmetic type"
```

3. Problem: Implement a custom type trait `is_swappable<T>` that:

- Returns `true` for types where `sizeof(T) == 2, 4, or 8` (like `short`, `int`, `long`).
- Returns `false` otherwise (e.g., `char`, `double`).

Use this trait in a `byte_swap(T value)` function that swaps bytes only for valid types.

Example:

```
byte_swap(0x1234);      // OK, swaps bytes
byte_swap(3.14);        // Fails at compile-time
```

4. Problem: Write a function `print_array_info(T& arr)` that:

- Prints "Array size: N" if `T` is a fixed-size array.
- Prints "Dynamic array" if `T` is a pointer (decayed array).
- Fails for non-array types.

Example:

```
int arr[5];
int* dyn_arr = new int[10];
print_array_info(arr);      // Output: "Array size: 5"
print_array_info(dyn_arr);  // Output: "Dynamic array"
print_array_info(42);       // Fails to compile
```

Hint: Use `std::extent` and `std::is_array`.

5. Problem: Create two overloads of a function `process`:

- For iterable containers (checks for `begin()/end()` members using SFINAE).
- For non-iterable types (default overload).

Example:

```
std::vector<int> v = {1, 2, 3};
process(v);  // Calls iterable overload (prints "Container")
process(42); // Calls default overload (prints "Non-container")
```

6. Problem: Build a `TypeDispatcher` class template that:

- Provides a `dispatch()` method calling different functions based on whether `T` is:
  - Integral, floating-point, pointer, or other.
- Uses `if constexpr` or template specialization.

Example:

```
TypeDispatcher<int>::dispatch();    // Calls handle_integral()
TypeDispatcher<float>::dispatch();  // Calls handle_float()
TypeDispatcher<std::string>::dispatch();  // Calls
handle_other()
```