Type Traits

Rupesh Nasre.

OOAIA
January 2020

Introduction

- Templates provide generic functionality.
  - Allow arbitrary types
- Situations exist when we need more control over the types.
  - for correctness
  - for efficiency
- This control is provided via Type Traits.

Example

- Consider a requirement where you are writing a function to swap bytes in values.
- This function should work whether the data type is 16-bit, 32, 64, ...
- Thus, *0x*123456789 becomes *0x*341278569.

Example: Issue

- Does not work for **T** = char (fixable).
- What happens with **T** = double?
- We do not have control over the type passed.

```
template <typename T>
T byteSwap(T value) {
    unsigned char *bytes = reinterpret_cast<unsigned char * >(&value);

    for (size_t ii = 0; ii < sizeof(T); ii += 2) {
        unsigned char tt = bytes[ii];
        bytes[ii] = bytes[ii + 1];
        bytes[ii + 1] = tt;
    }
    return value;
}
```

Example: Fix

- Specialize for special types
- But then
  - float? pointer?
  - Quickly gets to several specializations
  - This is what templates are supposed to avoid!

```
template <>
double byteSwap(double value) {
    assert(false && “Illegal to swap doubles”);
    return value;
}

template <>
char byteSwap(char value) {
    assert(false && “Illegal to swap characters”);
    return value;
}
```

Solution: Type Traits

- Provide information of types passed in templates
- Allow making intelligent / specialized decisions
- All at compile time

- In particular:
  - Allows querying if the template argument type is integer or pointer or void ...
  - Allows custom structure creation with special flags
  - Allows writing to and reading from the flags

```
template <typename T>
struct is_swapable {
    static const bool value = false;     // by default, nothing is swappable
};

// except these types
template <>
struct is_swapable<unsigned short> { static const bool value = true; };

template <>
struct is_swapable<short> { static const bool value = true; };

template <>
struct is_swapable<unsigned long> { static const bool value = true; };

template <>
struct is_swapable<long> { static const bool value = true; };

template <>
struct is_swapable<unsigned long long> { static const bool value = true; };

template <>
struct is_swapable<long long> { static const bool value = true; };
```

Type Traits Functioning

- **byteSwap** asks if the type **T** is swappable.
- Compiler finds the right struct based on **T** and finds the value.
- But the assertion failure happens at runtime.

```
template <typename T>
T byteSwap(T value) {
    assert(is_swapable<T>::value && “Cannot swap this type”);
    unsigned char *bytes = reinterpret_cast<unsigned char * >(&value);

    for (size_t ii = 0; ii < sizeof(T); ii += 2) {
        unsigned char tt = bytes[ii];
        bytes[ii] = bytes[ii + 1];
        bytes[ii + 1] = tt;
    }
    return value;
}
```

Existing Type Traits

- You may say this is too much coding!
- Yes, but this saves you.
- In addition, C++ 11 onward provides existing type traits.
- The following assertion failure occurs at compile-time.

```
static_assert(std::is_integral<T>::value && sizeof(T) >= 2,
                    “Cannot swap this type”);
```

List of Type Traits

- A very long list
  - has_virtual_destructor, is_arithmetic, is_array, is_assignable, is_class, is_base_of, is_const, is_function, is_literal_type, is_null_pointer, is_pointer, is_polymorphic, is_reference, is_scalar, is_volatile, ...

Array Rank Example

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

int main() {
  std::cout << "int: "          << std::rank<int>::value          << std::endl;
  std::cout << "int[]: "        << std::rank<int[]>::value        << std::endl;
  std::cout << "int[][10]: "    << std::rank<int[][10]>::value    << std::endl;
  std::cout << "int[10][10]: "  << std::rank<int[10][10]>::value << std::endl;
  return 0;
}
```

$ a.out

```
int: 0
int[]: 1
int[][10]: 2
int[10][10]: 2
```

Same Types Example

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

typedef int integer_type;
struct A { int x, y; };
struct B { int x, y; };
typedef A C;

int main() {
  std::cout << std::boolalpha;
  std::cout << "int, const int: " << std::is_same<int, const int>::value << std::endl;
  std::cout << "int, integer_type: "
            << std::is_same<int, integer_type>::value << std::endl;
  std::cout << "A, B: " << std::is_same<A,B>::value << std::endl;
  std::cout << "A, C: " << std::is_same<A,C>::value << std::endl;
  std::cout << "signed char, std::int8_t: "
            << std::is_same<signed char,std::int8_t>::value << std::endl;
  return 0;
}
```

$ a.out

```
is_same:
int, const int: false
int, integer_type: true
A, B: false
A, C: true
signed char, std::int8_t: true
```

Classwork

- Use a simpler algorithm if array is 1D or 2D. Otherwise, use another complex algorithm.
- If the type is vector<int>, use binary search. Otherwise, use linear search.

Acknowledgments

- http://blog.aaronballman.com/2011/11/a-simple-introduction-to-type-traits/
- http://www.cplusplus.com/reference/type_traits/