EECS 390 – Lecture 22

Template Metaprogramming

Template Metaprogramming

- Uses templates to produce source code at compile time, which is then compiled with the rest of the program's code
- A form of compile-time specialization that takes advantage of the language's rules for template instantiation
- Most common in C++, though it is available in D and a handful of other languages
- Template metaprogramming is Turing complete, with computations expressed recursively

Motivation

- Can be used to compute values at compile time
  - The values can then be used where compile-time constants are required, such as the size of a non-dynamic array
  - However, newer versions of C++ enable compile-time value computation with `constexpr` rather than template metprogramming
- Must be used to manipulate types
  - Types are compile-time entities, so they can only be manipulated at compile time
  - Example: `std::tuple`

```
std::tuple<int, double> items = { 3, 4.1 };
int first = std::get<0>(items);
double second = std::get<1>(items);
```

`std::get()` uses template metaprogramming to determine types of tuple elements

Template Specialization

- Key to template metaprogramming
- Allows a specialized definition for instantiating a template with specific arguments
- Example:

Generic definition

```
template <class T>
struct is_int {
  static const bool value = false;
};
```

This specialization has no template parameters of its own

Specialization for int argument

```
template <>
struct is_int<int> {
    static const bool value = true;
};
```

Full argument list for specialization

`is_int<double>::value` is false
`is_int<int>::value` is true

Reporting a Value

- We can report a value at compile time by arranging for it to be contained in an error message

```
template <class A, int I>
struct report {
  static_assert(I < 0, "report");
};

report<int, 5> foo;
```

Compile-time assertion

Dependent on template parameter so that assertion is after instantiation

Values

```
pair.cpp: In instantiation of 'struct report<int, 5>':
pair.cpp:70:16:   required from here
pair.cpp:67:3: error: static assertion failed: report
   static_assert(I < 0, "report");
   ^
```

Pairs

- We can represent a pair, whose items are arbitrary types, as:

```
template <class First, class Second>
struct pair {
    using car = First;
    using cdr = Second;
};
```

Type aliases

- We can represent an empty list as:

```
struct nil {
};
```

Alias Templates

- We can introduce alias templates to extract the first and second from a pair:

```
template <class Pair>
using car_t = typename Pair::car;

template <class Pair>
using cdr_t = typename Pair::cdr;
```

- The `typename` keyword is required when we have a nested type whose enclosing type depends on a template parameter
  - Otherwise, the compiler assumes we are referring to a value rather than a type

Empty Predicate

- Template specialization to determine if a list is empty:

```
template <class List>
struct is_empty {
  static const bool value = false;
};

template <>
struct is_empty<nil> {
  static const bool value = true;
};
```

Compile-time constant can be used as argument to report

Type aliases act as "variables"

```
using x =
  pair<char, pair<int, pair<double,
                           nil>>>;
using z = nil;
report<x, is_empty<x>::value> a;
report<z, is_empty<z>::value> c;
```

```
pair.cpp: In instantiation of 'struct
report<pair<char, pair<int, pair<double,
nil> > >, 0>':
pair.cpp:76:33:   required from here
pair.cpp:67:3: error: static assertion
failed: report
pair.cpp: In instantiation of 'struct
report<nil, 1>':
pair.cpp:78:33:   required from here
pair.cpp:67:3: error: static assertion
failed: report
```

Variable Templates

- C++14 introduced variable templates, which are parameterized variables that hold a value:

```
template <class List>
const bool is_empty_v = is_empty<List>::value;
```

- Then `is_empty_v<nil>` is true, but `is_empty_v<pair<int, nil>>` is false

```
using x =
  pair<char, pair<int, pair<double,
                            nil>>>;
using z = nil;
report<x, is_empty_v<x>> a;
report<z, is_empty_v<z>> c;
```

```
pair.cpp: In instantiation of 'struct
report<pair<char, pair<int, pair<double,
nil> > >, 0>':
pair.cpp:76:33:   required from here
pair.cpp:67:3: error: static assertion
failed: report
pair.cpp: In instantiation of 'struct
report<nil, 1>':
pair.cpp:78:33:   required from here
pair.cpp:67:3: error: static assertion
failed: report
```

Pair Length

- We can use a recursive template to compute the length of a list:

```
template <class List>
struct length {
  static const int value =
    length<cdr_t<List>>::value + 1;
};
template <>
struct length<nil> {
  static const int value = 0;
};
template <class List>
const int length_v = length<List>::value;
```

Base case

Variable template

```
report<x, length_v<x> d;
```

```
pair.cpp: In instantiation of 'struct report<pair<char,
pair<int, pair<double, nil> > >, 3>':
pair.cpp:79:31:   required from here
pair.cpp:67:3: error: static assertion failed: report
```

Reverse

- Reverse defined "tail recursively" as follows:

Remaining list

Reversed so far

```
template <class List, class SoFar>
struct reverse_helper {
  using type =
    typename reverse_helper<cdr_t<List>,
      pair<car_t<List>, SoFar>>::type;
};
```

Base case

```
template <class SoFar>
struct reverse_helper<nil, SoFar> {
  using type = SoFar;
};
```

Seed initial values

```
template <class List>
using reverse_t =
  typename reverse_helper<List, nil>::type;
```

Partial Class Template Specialization

- A class template may be partially specialized, accepting a subset of the template parameters

```
template <class SoFar>
struct reverse_helper<nil, SoFar> {
  using type = SoFar;
};
```

Any instantiation where the first argument is `nil` will use this

```
using x = pair<char, pair<int, pair<double, nil>>>;
report<reverse_t<x>, 0> e;
```

```
pair.cpp: In instantiation of 'struct report<pair<double,
pair<int, pair<char, nil> > >, 0>':
pair.cpp:80:32:   required from here
pair.cpp:67:3: error: static assertion failed: report
```

Numerical Computations

- We can use C++'s support for integer template arguments to perform numerical computations
- New version of `report` template:

```
template <long long N> struct report {
  static_assert(N > 0 && N < 0, "report");
};
```

Ensure that assertion will fail after instantiation, not before

Factorial

- Recursive computation of factorial:

```
template <int N> struct factorial {
  static const long long value =
    N * factorial<N - 1>::value;
};

template <>
struct factorial<0> {
  static const long long value = 1;
};
```

Compile-time constant

Base case

```
report<factorial<5>::value> a;
```

```
factorial.cpp: In instantiation of 'struct report<120ll>':
factorial.cpp:51:34:   required from here
factorial.cpp:47:3: error: static assertion failed: report
   static_assert(N > 0 && N < 0, "report");
   ^
```

Command-Line Macros

- We can use a macro to make our computation generic, and then specify the value at the command line

```
#ifndef NUM
#define NUM 5
#endif

report<factorial<NUM>::value> a;
```

Define a macro from command line

```
$ g++-mp-5 --std=c++11 factorial.cpp -DNUM=20
factorial.cpp: In instantiation of 'struct report<2432902008176640000ll>':
factorial.cpp:51:34:   required from here
factorial.cpp:47:3: error: static assertion failed: report
   static_assert(N > 0 && N < 0, "report");
   ^
```

Preventing Negative Input

- Negative input causes unbounded recursion
- We can prevent it as follows:

```
template <int N>
struct factorial_helper {
  static const long long value =
    N * factorial_helper<N - 1>::value;
};

template <>
struct factorial_helper<0> {
  static const long long value = 1;
};

template <int N> struct factorial {
  static_assert(N >= 0,
                "argument must be non-negative");
  static const long long value =
    factorial_helper<N >= 0 ? N : 0>::value;
};
```

Helper template does computation

Prevent instantiation of helper with negative value

Alternative: Default Argument

- Alternatively, we can use a second default argument that prevents unbounded recursion when the first argument is negative:

```
template <int N, bool /*Positive*/ = (N > 0)>
struct factorial {
  static const long long value =
    N * factorial<N - 1>::value;
};

template <int N>
struct factorial<N, false> {
  static const long long value = 1;
};
```

`factorial<5>` translates to `factorial<5, true>`, which uses the generic version

No name necessary here, since we don't use the parameter for anything else

`factorial<0>` translates to `factorial<0, false>`, which matches the specialization

Fibonacci Numbers

- We can compute Fibonacci numbers as follows:

```
template <int N> struct fib {
  static const long long value =
    fib<N - 1>::value + fib<N - 2>::value;
};

template <>
struct fib<1> {
  static const long long value = 1;
};

template <>
struct fib<0> {
  static const long long value = 0;
};
```

Two base cases

Computation is efficient, since compiler only instantiates a set of arguments once[1]

[1]This is akin to ***memoization*** in functional programming.