



EECS 490 – Lecture 23

Template Metaprogramming

1

Announcements

- ▶ HW5 due Tue 12/5 at 8pm
- ▶ Project 5 due Tue 12/12 at 8pm

Code Generation

- Macros allow us to use the facilities of a language to generate code
- However, a macro system may be unavailable or otherwise unsuited for the task at hand
- We can write a *code generator* in a separate program, in the same language or a different one, in order to generate the required code
- This is also called *automatic programming*

Scheme c*r Combinations

- Scheme implementations are required to provide combinations of car and cdr up to 4 levels deep

(cadar x) -> (car (cdr (car x)))

- We can write a Python program to generate definitions for these combinations, which we can then include as a library file in an interpreter

```
import itertools
```

```
for i in range(2, 5):
```

```
    for seq in itertools.product(('a', 'd'),  
                                repeat=i):
```

```
        print(defun(seq))
```

Define a
combination for
the sequence

Levels 2
up to 4

Create sequences
of 'a' and 'd'
with length i

Defining a Combination

- Function to construct body:

```
def cadrify(seq):  
    if len(seq):  
        body = "(c{0}r {1})"  
        return body.format(seq[0],  
                             cadrify(seq[1:]))  
    return "x"
```

Base case

Call for first item in sequence

Recursive call

- Function to construct a definition:

```
def defun(seq):  
    func = "(define (c{0}r x) {1})"  
    return func.format(''.join(seq),  
                        cadrify(seq))
```

Combinations

► Result of script:

```
(define (caar x) (car (car x)))  
(define (cadr x) (car (cdr x)))  
(define (cdar x) (cdr (car x)))  
(define (cddr x) (cdr (cdr x)))  
(define (caaar x) (car (car (car x))))  
(define (caadr x) (car (car (cdr x))))  
...  
(define (cdddar x) (cdr (cdr (cdr (car x)))))  
(define (cddddr x) (cdr (cdr (cdr (cdr x)))))
```

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

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;
};
```

Specialization
for int
argument

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

This specialization
has no template
parameters of its own

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

Compile-time
assertion

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

Dependent on
template parameter
so that assertion is
after instantiation

```
report<int, 5> foo;
```

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;
};
```

Type aliases
act as
"variables"

Compile-time
constant can be
used as argument
to report

```
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

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

Reversed
so far

Base
case

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

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

Seed initial
values

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;  
};
```

Compile-time
constant

Base
case

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

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

```
factorial.cpp: In instantiation of 'struct report<12011>':  
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<243290200817664000011>':  
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;
};
```

Helper template
does computation

```
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;
};
```

Prevent
instantiation of
helper with
negative value

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
once¹

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

- ▶ We'll start again in five minutes.

Templates and Function Overloading

- Function templates can be specialized, but functions can also be overloaded, so overloading a function template with a non-template function is more common
- C++ prefers a non-template over a template instantiation if the parameter types are equally compatible with the arguments

```
template <class T>
string to_string(const T &item) {
    std::ostringstream oss;
    return (oss << item).str();
}

string to_string(bool item) {
    return item ? "true" : "false";
}
```

```
to_string(3.14)
-> "3.14"
to_string(true)
-> "true"
```

SFINAE

- A key to function templates is that *substitution failure is not an error (SFINAE)*
- This means that it is not an error if a function template fails to instantiate due to the types and expressions in the header being incompatible with the argument
- Instead, the template is removed from consideration

```
template <class T>  
auto to_string(const T &item) ->  
    decltype(std::to_string(item)) {  
    return std::to_string(item);  
}
```

Requires compatible
std::to_string()

This template fails to
instantiate, but the
previous one succeeds

```
to_string(Complex{ 3, 3.14 })  
-> "(3,3.14i)"  
to_string(3.14)  
-> error: call is ambiguous
```

Both templates
are viable

Causing a Substitution Failure

- Sometimes we need to cause a substitution failure
- Common tool:

```
template <bool B, class T> struct enable_if {  
    typedef T type;  
};
```

```
template <class T> struct enable_if<false, T> {  
};
```

- Example:

```
template <int N> struct factorial {  
    static const typename  
        enable_if<N >= 0, long long>::type value =  
        N * factorial<N - 1>::value;  
};
```

This doesn't
exist if $N < 0$,
resulting in an
error

Overloading and Variadic Arguments

- We can use the fact that variadic arguments have lowest priority in overload resolution to prefer one overload over another:

```
template <class T>
auto to_string_helper(const T &item, int ignored)
    -> decltype(std::to_string(item)) {
    return std::to_string(item);
}
```

This overload
is preferred if
it is viable

```
template <class T>
string to_string_helper(const T &item, ...) {
    std::ostringstream oss;
    oss << item;
    return oss.str();
}
```

Variadic
arguments

```
template <class T>
string to_string(const T &item) {
    return to_string_helper(item, 0);
}
```

Dummy int
argument

Variadic Templates

- C++11 introduced support for templates that take a variable number of arguments
- Allows definition of variadic classes and functions that are type safe
- Example:

Accepts one
type argument

Parameter pack
accepts zero or more
type arguments

```
template <class First, class... Rest>
struct tuple {
    static const int size = 1 + sizeof...(Rest);
    // more code here
};
```

Size of
parameter pack

Empty
parameter
pack

```
tuple<int> t1;
tuple<double, char, int> t2;
```

Parameter
pack contains
char and int

Pattern Expansion

- An ellipsis to the right of a pattern that contains the name of a parameter pack is expanded into a comma-separated list

```
using first_type = First;  
using rest_type = tuple<Rest...>;
```

```
first_type first;  
rest_type rest;
```

If Rest contains char
and int, expanded
to tuple<char, int>

Recursive data
representation

Tuple Definition

```
template <class First, class... Rest>
struct tuple {
    static const int size = 1 + sizeof...(Rest);
    using first_type = First;
    using rest_type = tuple<Rest...>;
    first_type first;
    rest_type rest;
    tuple(First f, Rest... r) :
        first(f), rest(r...) {}
};
```

Expands to multiple parameters

Base
case


```
template <class First>
struct tuple<First> {
    static const int size = 1;
    using first_type = First;
    first_type first;
    tuple(First f) : first(f) {}
};
```

Constructing a Tuple

- We can write a function template to construct a tuple and then use it with argument deduction

```
template <class... Types>
tuple<Types...> make_tuple(Types... items) {
    return tuple<Types...>(items...);
}
```

```
tuple<int> t1 = make_tuple(3);
tuple<double, char, int> t2 =
    make_tuple(4.9, 'c', 3);
```



Argument
types
deduced

Representing a Tuple Element

- We define a struct to represent a tuple element:

```
template <int Index, class Tuple>
struct tuple_element {
    using rest_type =
        tuple_element<Index - 1,
                      typename Tuple::rest_type>;
    using type = typename rest_type::type;
    type &item;
    tuple_element(Tuple &t) :
        item(rest_type(t.rest).item) {}
};
```

```
template <class Tuple>
struct tuple_element<0, Tuple> {
    using type = typename Tuple::first_type;
    type &item;
    tuple_element(Tuple &t) : item(t.first) {}
};
```

Obtaining a Tuple Element

- We can then define a function template to obtain an item from a tuple:

**Alias
template**

```
template <int Index, class... Types>
tuple_element_t<Index, tuple<Types...>> &
get(tuple<Types...> &t) {
    return tuple_element<Index,
                        tuple<Types...>>(t).item;
}
```

```
tuple<double, char, int> t2 = make_tuple(4.9, 'c', 3);
cout << get<0>(t2) << endl;
cout << get<1>(t2) << endl;
cout << get<2>(t2) << endl;
get<0>(t2)++;
get<1>(t2)++;
get<2>(t2)++;
cout << get<0>(t2) << endl;
cout << get<1>(t2) << endl;
cout << get<2>(t2) << endl;
```

```
4.9
c
3
5.9
d
4
```