EECS 490 – Lecture 24

Advanced Metaprogramming

Announcements

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

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

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

- 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()

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

This template fails to instantiate, but the previous one succeeds

Both templates are viable

typename enable_if<N >= 0, int>::type

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

factorial<-3>::value

The standard library defines `enable_if` in `<type_traits>`.

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);
}

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

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

This overload is preferred if it is viable

Variadic arguments

Dummy int argument

tuple<int>

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:

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

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

Accepts one type argument

Parameter pack accepts zero or more type arguments

Size of parameter pack

2

Empty parameter pack

Parameter pack contains char and int

First

Rest

tuple<double, char, 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

tuple<char, int>

```
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

make_tuple (3, 4.1)

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...) {}
};
```

r0, r1

Expands to multiple parameters

char r0, int r1

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
```

C++ defines `tuple`, `make_tuple()`, and `get()` in `<tuple>`.

Multidimensional Arrays

- We can use metaprogramming to implement a multidimensional array abstraction in C++
- *Point*: a multidimensional index, represented by a sequence of integers

```
point<3> p = pt(3, -4, 5);
```

- *Domain*: a range of indices, represented by a lower-bound and an upper-bound point

```
rectdomain<3> rd{ pt(3, -4, 5), pt(5, -2, 8) };
```

- *Array*: constructed over a domain, indexed with a point

```
ndarray<double, 3> A(rd);
A[p] = 3.14;
```

p = pt(3, -4, 5);

p[0] p[1] p[8]

Points

- We can implement a point as follows:

```
template <int N> struct point {
  int coords[N];
  int &operator[](int i) {
    return coords[i];
  }
  const int &operator[](int i) const {
    return coords[i];
  }
};

template <class... Is>
point<sizeof...(Is)> pt(Is... is) {
  return point<sizeof...(Is)>{{ is... }};
}
```

Data representation

Function to construct a point

Inner initializer list is for initializing coords array

Point Operations

- Point operations have a common structure:

```
template <int N>
point<N> operator+(const point<N> &a,
                   const point<N> &b) {
  point<N> result;
  for (int i = 0; i < N; i++)
    result[i] = a[i] + b[i];
  return result;
}

template <int N>
bool operator==(const point<N> &a,
                const point<N> &b) {
  bool result = true;
  for (int i = 0; i < N; i++)
    result = result && (a[i] == b[i]);
  return result;
}
```

Generalized Macro

- General structure:

```
#define POINT_OP(op, rettype, header, action) \
  template <int N>                            \
  rettype operator op(const point<N> &a,      \
                      const point<N> &b) {    \
    header;                                   \
    for (int i = 0; i < N; i++)               \
      action;                                 \
    return result;                            \
  }
```

- Arithmetic structure:

```
#define POINT_ARITH_OP(op)                    \
  POINT_OP(op, point<N>, point<N> result,     \
           result[i] = a[i] op b[i])
```

Implementing Operations

- We can implement the operations as follows:

```
POINT_ARITH_OP(+);
POINT_ARITH_OP(-);
POINT_ARITH_OP(*);
POINT_ARITH_OP(/);

#define POINT_COMP_OP(op, start, combiner)       \
  POINT_OP(op, bool, bool result = start,        \
           result = result combiner              \
                    (a[i] op b[i]), result)

POINT_COMP_OP(==, true, &&);
POINT_COMP_OP(!=, false, ||);
POINT_COMP_OP(<, true, &&);
POINT_COMP_OP(<=, true, &&);
POINT_COMP_OP(>, true, &&);
POINT_COMP_OP(>=, true, &&);
```

Rectangular Domains

- Interface:

```
template <int N>
struct rectdomain {
  point<N> lwb;
  point<N> upb;

  int size() const;

  struct iterator;

  iterator begin() const;

  iterator end() const;
};
```

Exclusive upper bound

```
rectdomain<3> rd{ pt(1, 2, 3),
                  pt(3, 4, 5) }
for (auto p : rd)
  cout << p << endl;
```

```
(1,2,3)
(1,2,4)
(1,3,3)
(1,3,4)
(2,2,3)
(2,2,4)
(2,3,3)
(2,3,4)
```

Array Interface

```
template <class T, int N>
struct ndarray {
private:
  rectdomain<N> domain;
  int sizes[N];
  T *data;

  int indexof(const point<N> &index) const;

public:
  ndarray(const rectdomain<N> &dom);
  ndarray(const ndarray &rhs);
  ndarray &operator=(const ndarray &rhs);
  ~ndarray();

  T &operator[](const point<N> &index);
  const T &operator[](const point<N> &index) const;
};
```

Dimensionality

Element type

Translates multidimensional to linear index

Linear data representation

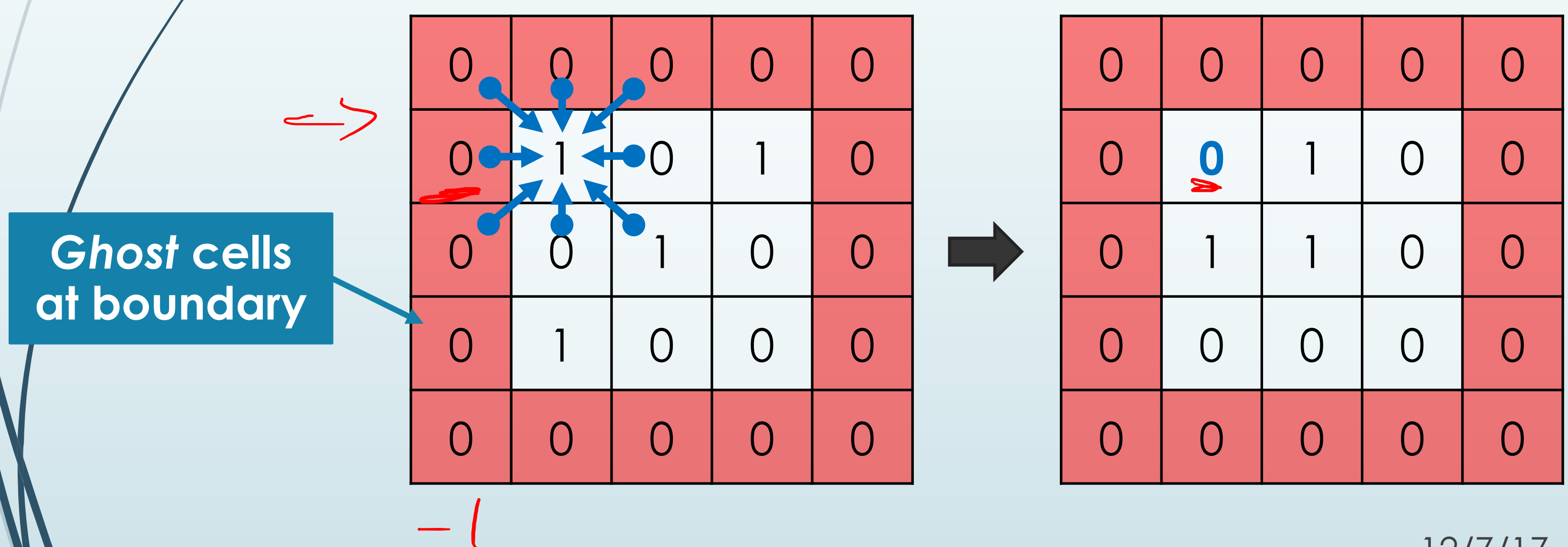
The Big 3

Break Time!

- We'll start again in five minutes.

Stencil

- A *stencil* is an iterative computation that updates grid points according to the previous value of neighboring points
- In the *Jacobi* method, the updates are *out of place*, so that new values are recorded in a different grid than old values



Stencil Data Structures

- Domains and arrays for 3D heat equation:

```
point<3> start = pt(0, 0, 0);
point<3> end = pt(xdim, ydim, zdim);

rectdomain<3> domain{ start + pt(-1, -1, -1),
                      end + pt(1, 1, 1) };
rectdomain<3> interior{ start, end };

ndarray<double, 3> gridA(domain);
ndarray<double, 3> gridB(domain);
```

Domain with ghost cells

Include ghost cells in array

Stencil Loop

- A single timestep:

```
for (auto p : interior) {
    gridB[p] =
      gridA[p + pt( 0, 0, 1)] +
      gridA[p + pt( 0, 0, -1)] +
      gridA[p + pt( 0, 1, 0)] +
      gridA[p + pt( 0, -1, 0)] +
      gridA[p + pt( 1, 0, 0)] +
      gridA[p + pt(-1, 0, 0)] +
      WEIGHT * gridA[p];
}
```

- Problem: this is very slow on some compilers, including GCC

Nested Iteration

- Some compilers can do powerful analysis on nested loops, optimizing the iteration order to take advantage of the memory hierarchy
- This loop is 5x faster in GCC:

```
for (p[0] = interior.lwb[0];
     p[0] < interior.upb[0]; p[0]++) {
  for (p[1] = interior.lwb[1];
       p[1] < interior.upb[1]; p[1]++) {
    for (p[2] = interior.lwb[2];
         p[2] < interior.upb[2]; p[2]++) {
      gridB[p] = ...;
    }
  }
}
```

Implementing Nested Iteration

- Template metaprogramming to generate nested loops:

```
template <int N> struct rdloop {
  template <class F, class... Is>
  static void loop(const F &func, const int *lwb,
                          const int *upb, Is... is) {
    for (int i = *lwb; i < *upb; i++)
      rdloop<N-1>::loop(func, lwb+1, upb+1, is..., i);
  }
};

template <> struct rdloop<1> {
  template <class F, class... Is>
  static void loop(const F &func, const int *lwb,
                          const int *upb, Is... is) {
    for (int i = *lwb; i < *upb; i++)
      func(pt(is..., i));
  }
};
```

Dimensions remaining

Functor object

Index bounds

Indices computed so far

Call functor with a point using computed indices

Custom Loop

- Loop abstraction:

Point variable **Domain**

```
foreach (p, interior) {
  gridB[p] =
    gridA[p + pt( 0, 0, 1)] +
    gridA[p + pt( 0, 0, -1)] +
    gridA[p + pt( 0, 1, 0)] +
    gridA[p + pt( 0, -1, 0)] +
    gridA[p + pt( 1, 0, 0)] +
    gridA[p + pt(-1, 0, 0)] +
    WEIGHT * gridA[p];
};
```

Looks like body of lambda function

Loop Macro

- Macro for `foreach`:

```
#define foreach(p, dom)                                   \
  for (auto _iter = (dom).iter();                         \
       !_iter.done;                                       \
       _iter.done = true)                                 \
    _iter = [&](const decltype((dom).lwb) &p)
```

Dummy loop header to introduce iterator object

Iterator object overloads assignment operator to take functor

Capture locals by reference

Deduce point type from domain

Point variable is lambda parameter

Fast Iterator

- Implementation:

```
struct fast_iter {
  const rectdomain &domain;
  bool done;

  fast_iter(const rectdomain &dom)
    : domain(dom), done(false) {}

  template <class F>
  fast_iter &operator=(const F &func) {
    rdloop<N>::loop(func, domain.lwb.coords,
                    domain.upb.coords);
    return *this;
  }
};
```

For dummy loop header

Assignment operator takes functor

Use nested loop generator

This matches the performance of nested loops on GCC 6.