EECS 390 – Lecture 23

Template Metaprogramming II

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 <u>the header</u> being incompatible with the argument
- Instead, the template is removed from consideration

Requires compatible std::to_string()

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

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 {
  using type = T;
};

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

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

Overloading and Variadic Arguments

- We can use the fact that C-style 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

Variadic Templates

- C++ has 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) {}
};
```

Tuple Access

```
template <class First, class... Rest>
struct tuple {
  first_type first;
  rest_type rest;
};
template <class First>
struct tuple<First> {
  first_type first;
};

template <int Index, class Tuple>
auto get(Tuple &&tuple) {
  if constexpr (Index == 0) {
    return std::forward<Tuple>(tuple).first;
  } else {
    return
      get<Index-1>(std::forward<Tuple>(tuple).rest);
  }
}
```

Move-semantics stuff

Compile-time conditional

Tuple Access and SFINAE

- We can induce a substitution failure if a different type is passed to our `get()` function template:

Has value member that is false

```
template<class T>
struct is_tuple : std::false_type {};
template<class... T>
struct is_tuple<tuple<T...>> : std::true_type {};

template <int Index, class Tuple,
          class = std::enable_if_t<
            is_tuple<
              std::remove_reference_t<Tuple>
            >::value,
            void
          >>
auto get(Tuple &&tuple) { /* ... */ }
```

Template parameter becomes void if tuple passed in, fails to substitute otherwise

Structured Bindings

- C++ has "structured bindings" that allow tuple-like types to be unpacked into separate variables:

```
tuple<int, double, char> t{3, 4.1, 'c'};
auto [i, d, c] = t; // i = 3, d = 4.1, c = 'c'
```

- To make this work, we need the following:

```
namespace std {
  template<class... T>
  struct tuple_size<::tuple<T...>> {
    static const int value = ::tuple<T...>::size;
  };
  template<std::size_t I, class T, class... U>
  struct tuple_element<I, ::tuple<T, U...>> :
    tuple_element<I-1, ::tuple<U...>> {};
  template<class T, class... U>
  struct tuple_element<0, ::tuple<T, U...>> {
    using type = T;
  };
}
```

Specialization of `std::tuple_size` for our tuples

Specializations of `std::tuple_element` to compute the element types for our tuples

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;
```

Points

- We can implement a point as follows:

Data representation

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

Inner initializer list is for initializing coords array

Function to construct a point

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

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

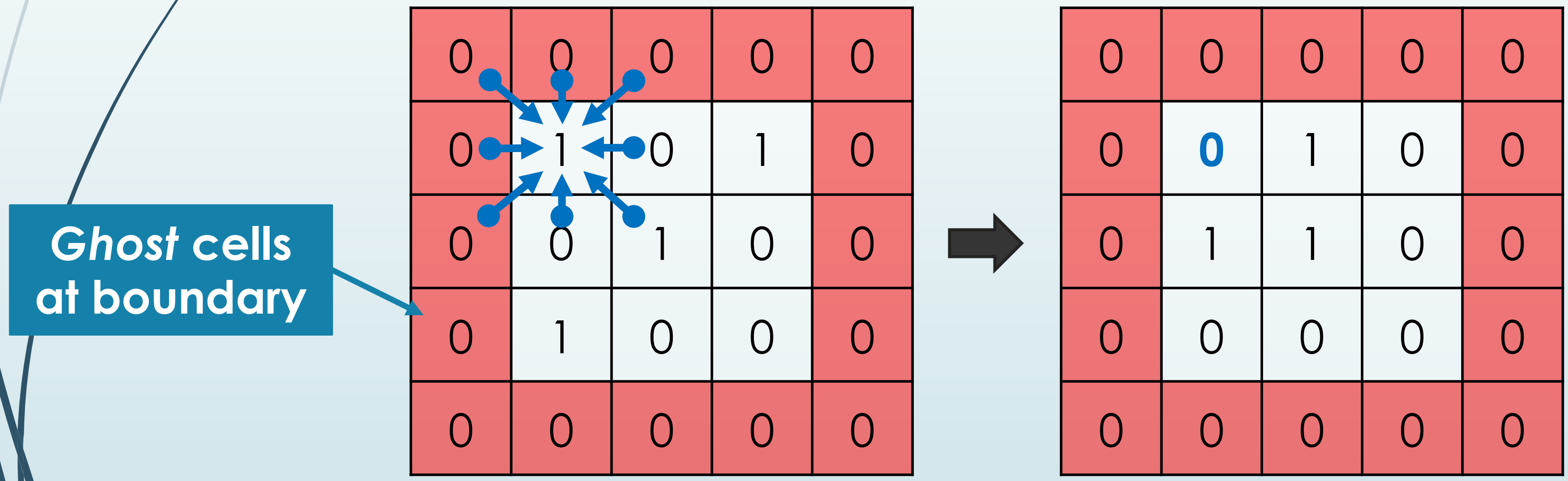
Translates multidimensional to linear index

Linear data representation

The Big 3

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];
 }
```

- Can be made significantly faster with custom loop construct, using macros, lambdas, and template metaprogramming (see the notes for details)