Currying and other Functional Constructs in C++

Achuthan Krishna

InterContinental Exchange

Similarities to std::function / std::bind

- Handles all callable entities
- Early binding of arguments
- Placeholder support
- Make member variables callable

Differences from std::function

- Almost zero abstraction penalty
- No type erasure
- No memory allocation
- No virtual functions
- Cannot be stored in containers
- Types are unwieldy
- Once callables are 'curry-enabled' they natively support currying

Interesting features

- Anonymous placeholders
- Omit trailing placeholders
- Enable currying on the callable entity. Makes the usage nicer
- Composition of functions
- Additional FP features
    - Monads
        - With Each (Acts like inverted for_each)
        - Maybe
        - Easy to add Error Monad, Exception Monad etc
    - Higher Order Functions
        - fmap, filter, foldr, foldl etc.,
    - Easy to enable currying on make existing STL functions

Code Available at https://github.com/KrishnaAchuthan/curry

```
struct Point {
   double x;
   double y;
};

auto operator + (Point p1, Point p2) {
   return Point{ p1.x + p2.x, p1.y + p2.y };
};

auto operator / (Point p, int scalar) {
   return Point{ p.x / scalar, p.y / scalar };
};

auto get_input_file_names(string path) {
   vector<string> file_names;
   for (auto it = directory_iterator(path); it != directory_iterator(); ++it) {
      const auto& file = it->path();
      if (file.extension() == ".points") {
         file_names.push_back(file.string());
      }
   }
   return file_names;
};

auto read_file(string full_path) {
   ifstream file(full_path);
   vector<string> lines;
   if (file) {
      string line;
      while (getline(file, line)) {
         lines.push_back(line);
      }
   }
   return lines;
};

auto parse_point = [](string str) {
   regex pattern("(\\d+),(\\d+)");
   std::smatch match;
   if (std::regex_match(str, match, pattern)) {
      if (match.size() == 3) {
         return Point{double(std::stoi(match[1].str())),
                      double(std::stoi(match[2].str()))};
      }
   }
   return Point{};
};
```

Code Available at https://github.com/KrishnaAchuthan/curry

```
int main() {
   auto points_box =
        get_input_file_names("C:\\Point_Files")
      | with_each
      | read_file
      | parse_point;

   auto points = unbox(points_box);

   auto centroid_of_cloud = foldl(_ + _, Point{}, points) / points.size();

   return 0;
}
```

Code Available at https://github.com/KrishnaAchuthan/curry

Function Composition

```
//foo takes four arguments

auto f1 = fn(foo);

//bar is a function that takes two arguments. It is composed in.
auto f2 = f1(arg1, _, arg2, arg3) * bar;

//Calls foo(arg1, bar(arg4, arg5), arg2, arg3)
auto f3 = f2(arg4, arg5);
```

With Each

```
int add_three_numbers(int a, int b, int c) {
   return a + b + c;
}

std::vector<int> v1 = { 19,17,21 };
std::vector<int> v2 = { 23,49 };
std::vector<int> v3 = { 7, 13 };

auto adder  = fn(add_three_numbers);
auto result = adder(with_each(v1), with_each(v2), with_each(v3));
//result will be a vector with 12 values {49, 55, 75, 81, 47, 53, 73, 79, 51, 57, 77, 83}
```

Maybe

```
int add_three_numbers(int a, int b, int c) {
   return a + b + c;
}

auto value1 = maybe(10);
auto value2 = maybe(20);
auto value3 = maybe(30);
auto value4 = maybe_t<int>(); //Empty

auto adder  = fn(add_three_numbers);

auto result1 = adder(value1, value2, value3);
//result1 will be maybe_t<int>(60)

auto result2 = adder(value1, value2, value4);
//result2 will be maybe_t<int>()
```

Higher Order Functions

```
auto square = fmap(_1*_1);
auto fmap_result = square(lst3); //result will be [1,2,9,16,25,36,49,64,81]

auto sum = foldl(_+_, 0);        //curried foldl application
auto product = foldr(_*_, 1);    //curried foldr application

auto sum_result = sum(lst3);     //result will be 45
std::cout << sum_result << std::endl;

auto product_result = lst3 | product; //result will be 362880
std::cout << product_result << std::endl;
```

Code Available at https://github.com/KrishnaAchuthan/curry