Lab 19: Auto, Ranged-For, Smart Pointers

1. **Fixing bad code with auto, ranged-for, and vector [20 minutes]:** Code along with the GSI as we fix and improve some code. The template code for this example is here. This code was taken from a popular online tutorial on detecting cycles in graphs, but it is ugly and leaks memory. After applying `auto`, ranged-for, and removing the usage of raw pointers, this code will be improved in several ways:
   a. no memory leaks!
   b. easier to debug, because the debugger can actually show us the values
   c. more readable, since ugly iterator and pointer syntax will be removed
   d. almost as fast*

2. **Rewriting code using auto [10 minutes]:** Copy the following code into a file called `lab19_auto.cpp`, then go through and try to replace every explicit type with the `auto` keyword. Keep track of how many things you were able to replace with auto, we will ask about this on the quiz. You don't need to worry about exactly what the code does, but just know that it should print `55374`.

```
#include <iostream>
#include <map>

std::array<unsigned long, 500> pentagonal;

unsigned long partition(unsigned long n) {
    static std::map<unsigned long, unsigned long> memoization = {{0, 1},
                                                                 {1, 1}};
    std::map<unsigned long, unsigned long>::iterator const it = memoization.find(n);
    if (it != memoization.cend()) {
        return it->second;
    }

    long partitions{0l};
    for (int k{1};; ++k) {
        int const sign = ((k + 1) / 2) % 2 == 0 ? -1 : 1;
        unsigned long const pent = pentagonal[k];
        if (n < pent) {
            break;
        }
        partitions += sign * partition(n - pent);
    }
    partitions = partitions % 1'000'000;
    if (partitions < 0) {
        partitions += 1'000'000;
    }
    memoization.emplace(n, partitions);
    return static_cast<unsigned long>(partitions);
}

unsigned long problem() {
    // pre-compute pentagonal numbers
    int k{1};
    for (unsigned long j{1ul}; j < pentagonal.size(); ++j) {
        pentagonal[j] = static_cast<unsigned long>(k * (3 * k - 1) / 2);
        if (k > 0) {
            k = -k;
        } else {
```

```
            k = -k + 1;
        }
    }

    unsigned long n{0ul};
    while (true) {
        unsigned long const p = partition(n);
        if (p == 0) {
            return n;
        }
        ++n;
    }

}

int main() {
    // Solves Project Euler 78: https://projecteuler.net/problem=78
    unsigned long solution = problem();
    std::cout << solution << '\n';
    return 0;
}
```

3. **Practice using ranged-based for [10 minutes]:** Copy the following code into a file called `lab19_ranged_for.cpp`, then go through and fill in the for-loops using ranged-for syntax and `auto`.

```
#include <iostream>
#include <vector>
#include <cmath>
#include <map>
#include <utility>
#include <string>

struct JointLimit
{
    double lower_;
    double upper_;

    JointLimit(double lower, double upper) : lower_(lower), upper_(upper) {}
};

double clip(double joint_position, JointLimit const &joint_limit)
{
    if (joint_position < joint_limit.lower_)
    {
        return joint_limit.lower_;
    }
    else if (joint_position > joint_limit.upper_)
    {
        return joint_limit.upper_;
    }
    else
    {
        return joint_position;
    }
}

void enforce_joint_limits(std::map<std::string, JointLimit> const &joint_limits, std::map<std::string, double> &q)
{
    for () // create a for-loop over "q" with the loop variable "kv" auto const&
    {
        std::string const joint_name = kv.first;
        double joint_position = kv.second;
        JointLimit const joint_limit = joint_limits.at(joint_name);
        q[joint_name] = clip(joint_position, joint_limit);
    }
```

```
}

int main()
{
    std::vector<double> joint_values{0, 0.2, 0.5, -0.4, 0.5};
    // create a ranged-for loop over joint_values with loop variable named "joint_i".
    // IMPORTANT!!!! we want to modify the joint values!
    for ()
    {
        joint_i *= 2;
    }

    std::map<std::string, JointLimit> joint_limits{
        {"joint1", JointLimit{0, M_PI}},
        {"joint2", JointLimit{-M_PI, M_PI}},
        {"joint3", JointLimit{-M_PI, 0}},
        {"joint4", JointLimit{-2 * M_PI, 2 * M_PI}},
        {"joint5", JointLimit{-2 * M_PI, 2 * M_PI}},
    };

    std::map<std::string, double> q{
        {"joint1", -2.0},
        {"joint2", 2.0},
        {"joint3", 2.0},
        {"joint4", 3.0},
    };

    enforce_joint_limits(joint_limits, q);

    double final_answer = 0;
    for () // create a ranged-for loop over "q" with loop variable named "kv"
    {
        final_answer += kv.second;
    }
    for () // create a ranged-for loop over "joint_values" with loop variable named joint_i
    {
        final_answer += joint_i;
    }

    std::cout << final_answer << "\n";
    return 0;
}
```

4. **Understanding the lifecycle of smart pointers [15 minutes]**: Copy the following code into a file called `lab19_lifecycle.cpp`. Then, run it in the debugger and figure out exactly where and when the constructor and destructor are called. We will ask you about this on the quiz. I suggest annotating each line of code in `main` with a comment saying whether a constructor/destructor is called on that line.

```
#include <iostream>
#include <memory>

class BigOldClass
{
public:
  BigOldClass(int size) : _size(size)
  {
    std::cout << "constructing BigOldClass: " << this << std::endl;
  }

  BigOldClass(BigOldClass const& copy)
  {
    std::cout << "constructing BigOldClass by copying: " << this << std::endl;
    _size = copy._size;
  }
```

```
  ~BigOldClass()
  {
    std::cout << "destructing BigOldClass: " << this << std::endl;
  }

  int _size;
};

void print_copy(BigOldClass c)
{
  std::cout << "pass by value: address: " << &c << std::endl;
}

void print_reference(BigOldClass const &c)
{
  std::cout << "pass by const ref: address: " << &c << std::endl;
}

void print_pointer(BigOldClass const *c)
{
  std::cout << "pass by raw ptr: address: " << c << std::endl;
}

void print_unique_pointer(std::unique_ptr<BigOldClass> const &c)
{
  std::cout << "pass by const ref to unique_ptr: address: " << c.get() << std::endl;
}

int main()
{
  BigOldClass c1(100);
  auto *c1_ptr = &c1;
  auto const c1_unique_ptr_copy = std::make_unique<BigOldClass>(c1);
  auto const c1_unique_ptr_emplace = std::make_unique<BigOldClass>(100);

  print_copy(c1);
  print_reference(c1);
  print_pointer(&c1);
  print_unique_pointer(c1_unique_ptr_copy);
  print_unique_pointer(c1_unique_ptr_emplace);

  return 0;
}
```

5. **Practice using shared_ptr [15 minutes]:** Copy the following code into a filed called `lab19_shared_ptr.cpp`. Then fill in the code according to the comments. Practice using `auto` and `std::make_shared`. The code should run and print “Data of size: 10” (the value of size is up to you).

```
#include <memory>
#include <iostream>

class BigOldClass
{
public:
    BigOldClass(int size) : _size(size) {}

    int _size;
};

class BigDataManager
{
public:
```

```
    // Create a constructor that takes in a shared pointer to a BigOldClass and initializes the data_ member
    // --- Your code here
    //

    std::shared_ptr<BigOldClass> data_;
};

std::shared_ptr<BigOldClass> make_big_thing() {
    // create and return a shared pointer to a BigOldClass, you can set whatever "size" you want.
}

void print_manager(BigDataManager const &m) {
    std::cout << "Data of size: " << m.data_->_size << std::endl;
}

int main()
{
    // call make_big_thing -- use auto!

    // construct a BigDataManager using the BigOldClass you just got from calling make_big_thing

    // call print_manager using the BigDataManager you just created

    return 0;
}
```