Computing at Scale

Lecture 8: Introduction to Templates

Jacob Merson

February 6, 2025

Logistics

Today's Agenda

- Homework Groups/Questions
- Initialization Reminder
- Noisy Class
- Class Templates

Assignment 1

- Due today at 10PM
- Submit on Github. Make sure to update your README.md in the class repo with a link to your pull request.

Assignment 1 Groups

- **Group 1:** Fuad Hasan, Jay Gaiardelli, Zach Knowlan
- **Group 2:** Bibek Shrestha , Kairvi Lodhiya, Abhiyan Paudel
- **Group 3:** Scott Blender, Ickbum Kim, Mikiel Gica

Object Initialization

Initialization

C++ provide a range of ways to initialize types

```
double d1 = 2.3;
double d2{2.3};
double d3 = {2.3};

std::complex<double> z = 1;
std::complex<double> z2{1, 2};
std::complex<double> z2 = {1, 2};

std::vector v{1, 2, 3, 4}; // initializer list constructor
```

Initialization

- Prefer = `{}` for initialization
- `{}` prevents narrowing conversions
- Be careful if type has a constructor that takes an initializer list

```
int i1 = 6.8; // i1 = 6!?
int i2{6.8}; // error, narrowing conversion
```

Initialization

Use your “Noisy” Class to test the following and note any surprises.

```
int main() {
 Noisy n1{1};
 auto n2 = Noisy{2};
 Noisy n3 = {3};
 n1 = n2;
 n2 = std::move(n3);
 auto n4{std::move(n1)};
}
```

- If all of the functions are named the same, which ones are preferred? Try this with reference types as well.
- When does move/copy occur?

```
value(Noisy n){}
ref(Noisy& n) {}
cref(const Noisy& n) {}
rref(Noisy&& n) {}

int main() {
  Noisy n1{1}, n2{2}, n3{3};
  value(n1);
  value(std::move(n1));
  value(Noisy{4});
  ref(Noisy{5});
  cref(Noisy{6});
  rref(n2); // any moves?
  rref(std::move(n3));
}
```

Template Parameter Passing

- How is the template version different?
- If these functions are overloaded what will happen?

```
template <typename T>
value(T n){}

template <typename T>
ref(T& n) {}

template <typename T>
cref(const T& n) {}

template <typename T>
fref(T&& n) {}
```

```
int main() {
  Noisy n1{1}, n2{2}, n3{3};
  auto& cn1 = n1;
  const auto& cn2 = n2;
  value(n1);
  value(std::move(n1));
  value(Noisy{4});
  ref(Noisy{5});
  cref(Noisy{6});
  rref(n2); // any moves?
  fref(std::move(n3));
}
```

Class Templates

Class Templates

- Class templates are similar to function templates
- Allow DRY code for datastructures
- You have already been using them! `std::vector`

Class Template Example

```cpp
template <typename T>
class Stack {};

int main() {
  Stack<int> s1;
  Stack<double> s2;
}
```

- Use the `template` keyword to define a class template
- Prefer the use of `typename`
- Note, s1 and s2 are different types (as different as int and string)

Stack

```
class Stack{
public:
  Stack(const Stack& );
  Stack& operator=(const Stack&);
  void push(const T& value);
  void pop();
  const T& top() const;
  bool empty() const { return data_.empty(); }

private:
  std::vector<T> data_;
};
```

Constructor

- Any time you use the name Stack in the class it implicitly adds the template parameter

That is,

```
Stack(const Stack& );
Stack& operator=(const Stack&);
```

is equivalent to

```
Stack<T>(const Stack<T>& );
Stack<T>& operator=(const Stack<T>&);
```

Member Function Definition

Use local template parameters to define member functions

```
template <typename T>
void Stack<T>::push(const T& value) {
  data_.push_back(value);
}

template <typename T>
void Stack<T>::pop() {
  data_.pop_back();
}

template <typename T>
const T& Stack<T>::top() const {
  return data_.back();
}
```

Class template Notes

- Remember, only member functions and classes that are called are instantiated. Must be careful about testing.
- Before C++17, template classes had not type deduction. You had to specify the type explicitly when creating the object.
- Often you will see template functions that wrap classes to perform type deduction. E.g., `std::move`

Specialization

- Class templates can be specialized to provide different implementations for different types.
- If you specialize a class template, you must specialize all member functions.
- You can also partially specialize a class template.
- Do not provide a specialization with different member functions or semantics. It will cause you endless pain, confusion, and possibly yelling at your rubber duck. And, no rubber duck deserves that.
- See `std::vector<bool>` for an example of what not to do.

Specialization Example

To specialize a class template, simply append with `template<>`. And, provide specialzed member functions.

```
template<>
class Stack<std::string> {};
void Stack<std::string>push(std::string const& value){
  data_.push_back(value);
}
```

Partial Specialization

Specialized for T*, but still parameterized on T.

```
template <typename T>
class Stack<T*> {};
```

With multiple Parameters

```
template <typename T, typename U> class Pair {};

// specialization for when T and U are the same
template<typename T> class Pair<T, T> {};

// specialization for when T and U are pointers
template<typename T, typename U> class Pair<T*, U*> {};

// specialization for int, int
template<> class Pair<int, int> {};
```

Other Class Template Notes

- Template classes can have default arguments
- After C++17, template classes types can be deduced. See, "Class Template Argument Deduction" (CTAD).
- Template classes can have non-type template parameters. E.g., `std::array<int, 5>`
- You can template alias. E.g., `using IntStack = Stack<int>;`
- You can create templates with variable number of arguments. E.g., `std::tuple`. See variadic templates.

Rvalue References in Class Templates

- Dealing with && references in class templates takes some care.
- If && is applied to the class template type, it will be an rvalue reference. As class template type will already be concrete when member functions are defined. I.e., on class instantiation.
- If you have a *templated* member function in a template class, the && will be an forwarding reference.

```
template <typename T>
class MyClass {
  public:
    void f(T&& t) { /* rvalue reference */ }
    template <typename U>
    void g(U&& u) { /* forwarding reference */ }
    template <typename U>
    MyClass(U&& u) { /* forwarding reference */ }
};
```

Exercise: Update Vector to be a class template.

Exercise: Use a NTTP to create a class that creates a matrix that is either staticly or dynamically sized.