SPECTRAL:
TECHNOLOGIES

WEEK 2

Basic features recap

Templates, iterators, lambdas, move-semantics

Template functions

- The simplest usage is to create type agnostic functions

```
template<class T>
T min(T a, T b) {
        return a < b ? a : b;
}
```

- But we can also parametrise the function's behaviour

```
template<class Iter, class Pred, class Operator>
void TransformIf(Iter first, Iter last, Pred predicate, Operator op);
```

- And even more with metaprogramming (later in this course)

Template functions

- Template "function" isn't a function --- it doesn't exist until there is an instantiation in the code.
- Every unique instantiation creates a new version of this function in the resulting assembly. Thats why an extensive usage of templates can lead to a slower compilation process and bigger executable files.
- Duck typing is a core feature of c++-templates (use only what you need).

Template classes

- Again, can be used as a type parametrization mechanism or provide some logic control

```
template<class T> class Vector { };

template<class K, class V, class RetentionPolicy> class Cache { };
```

- Class doesn't exist until there is an instantiation.
- Even then, only the code for called methods is generated.
- Consider the difference

```
template<class T> T min(T a, T b);
template<class T> class C {
        T min(T a, T b);
}
```

Templates

- Provide static polymorphism, which is basically 0-cost in runtime, but has it’s limitations:

```
vector<Object*> objects{...};
for (auto obj : objects) {
    obj->f();
}
// or
template<class Callback>
struct C1 {
      explicit C1(Callback callback) : callback_(callback) {}
      Callback callback_;
} // vs
struct C2 {
      explicit C2(std::function<void(Error*)> callback) : callback_(callback) {}
      std::function<void(Error*)> callback_;
}
```

Templates

- However a lot can be done at compile time too (CRTP, SFINAE, etc.)
- Quite often the template code boils down to visitor pattern:

```
template<class Visitor>
void f(Visitor *visitor) {
      if (condition1) {
         visitor->OnCondition1(...);
      } else if (condition2) {
         visitor->OnCondition2(...);
      }
      ...
      visitor->OnExit();
}
```

Functors

- Functor is a class with overloaded operator `()`, which allows you to save any state between invocations comparing to a regular function:

```
template<class T>
struct CountingComparator {
	explicit CountingComparator(int *counter) : counter_(counter) {}
	bool operator()(const T& a, const T& b) const {
		++(*counter_);
		return a < b;
	}
	int *counter_;
}
int comparisons_count = 0;
std::sort(data.begin(), data.end(), CountingComparator<int>(&comparisons_count))
```

Lambdas

- Lambda function is a nameless functor:

```
int comparisons_count = 0;
std::sort(data.begin(), data.end(), [&comparisons_count](int a, int b) {
    ++comparisons_count;
    return a < b;
})
```

- Everything in `[]` goes to the constructor (and class members) and the function itself is operator `()` of this class.

Iterators

- A lightweight class, that allows you to look inside a container. Usually a wrapper around a single or a few raw pointers.
- There are multiple categories:
  - random_access (`vector` , `deque`)
  - bidirectional (`list` , `map/set` , `unordered_map/set`)
  - forward (forward_list)
  - other
- If you want your class to support range-based for, then it should provide methods `begin` and `end` , which return an object of class (e.g. `Iterator`) such as:
  - it implements `operator++()` and `operator++(int)` (prefix and postfix)
  - it implements `operator!=` and `operator==`

move-semantics

- Basically the way to say "I don't need this variable anymore, but want to copy it’s content somewhere"
- `std::move` doesn't move anything, since its just a cast to &&. So its not even an instruction in the resulting code.
- Assume you have some non-template function `f(obj)` , which receives some heavy object. Then
  - if you don't need to modify it, pass it as void f(const Object& obj)
  - if you need to modify it and want the caller to get these modifications, pass it as `void f(Object& obj)` (or `void f(Object *obj)`)
  - if you need to modify it and don't want the caller to get these modifications, pass it as `void f(Object obj)` . This allows the caller to move an object inside this function.

move-semantics and templates

- You don't usually want to have something like `void f(Object&& obj)` .
- However this is pretty common:

```
template<class T>
void f(T&& value) {
      // ...
      g(std::forward<T>(value));
}
```

- This is because `value` here isn't exactly an rvalue reference, but something called a universal reference.
- Use `std::forward` for universal references (its like a smart version of move.)
- Widely used by `emplace` -methods of different std-containers.

Literature

- Scott Meyers "Effective C++: 55 Specific Ways to Improve Your Programs and Designs"
- Scott Meyers "Effective Modern C++"