Logs

Matt Warner

1 8/10/24

1.1 Operator overload returns

Say we had the following code:

```
vec1 = vec2;
```

Inside the body of the overloaded `operator=` function, we are copying over the internal state of the right operand to the object on the left side of `operator=` (`this`). Internally, it might look like this:

```
if (this != &rhs)  {

    delete[] data;
    m_size = rhs.m_size;
    m_capacity = rhs.m_capacity

    std::copy(rhs.begin(), rhs.end(), data);
    return *this;
}
```

But why do we need to return `*this`? If we are just copying data to our current object, why not just use void?

The reason why we return `*this` is simple, returning `*this` allows for **chaining operations**. Like this:

```
vec1 = vec2 = vec3;
```

Here's how it works:

- vec2 = vec3 calls the assignment operator, which modifies vec2 and returns *this (which is now vec2).
- The result of vec2 = vec3 is then assigned to vec1, calling the assignment operator again, this time for vec1.

Without returning *this, you cannot chain assignments in this manner. If operator= returned void, the assignment vec1 = vec2 = vec3 would not work as expected because the intermediate result (the updated vec2) would not be usable for further assignments.

Therefore, you should return *this whenever writing `operator overloads` that can be chained together. These are:

- `operator=`
- `operator+=`
- `operator-=`
- `operator-`
- `operator+`
- `operator++`
- `operator--`

Furthermore, consider these:

```
it1 = it2 - it3;
it2 = ++it3;

// perhaps even this one
it2 = ++(++it3);
```

These examples would not be possible without returning the modified object.

1.2 pointer arithmetic

Say we had a pointer to a dynamically allocated array:

```
int* x = new int[5];

// Filling the array
int* ptr = x;
for (; ptr < x+5; ++ptr) {
    *ptr = 1;
}
```

We can use pointer arithmetic to jump to positions in the array like so:

```
int* ptr2 = x + 2;  // Moves the pointer to the third element of the array
ptr = (x + 2) - 2;  // Moves the pointer back to the start of the array
ptr += 2;           // Moves the pointer forward by 2 positions (to the third element)
ptr -= 2;           // Moves the pointer back by 2 positions (to the start of the array)
```

However, there are some invalid operations to consider:

```
ptr += ptr2;   // Not allowed: adding two pointers
ptr = ptr + ptr2; // Not allowed: adding a pointer and another pointer
ptr-=ptr2; // also not allowed
```

You cannot add two pointers together because it does not make sense in terms of pointer arithmetic. Pointers represent positions in memory, and adding them does not yield a meaningful result.

Furthermore, subtracting two pointers yields a distance of type `ptrdiff_t`

```
auto distance = ptr - ptr2;  // yields the distance between two pointers
```

1.3 adding typename to typedefs

In C++, whenever you are creating typedefs for types that require template parameters, you need to add the keyword `typename`. For example, if you had an iterator for your vector class:

```
template <class vector>
class vector_iterator  {
public:
    typedef typename vector::value_type value_type; // typename required
    typedef value_type* pointer; // no typename needed
};
```

1.4 `This()` memeber function

When designing a class, we can throw in the following function:

```
class x {
public:
    x* This() { return this; }
};
```

This gives us a way to get a pointer to an object:

```
x obj;
x* = obj.this();
```

This is usually not neccassary to do but it might be worth knowing.

2 8/17/24

2.1 Variadic Templates

In C, we could can use ellipses (...) to allow functions to accept an arbitrary number of parameters.

```
void foo(int x,char...);
```

Modern C++ adds a feature known as **variadic templates**, which can be seen when a template has a parameter pack in its parameter list. The syntax looks like this:

```
template <typename... Args>
void foo(Args... args);
```

A template with at least one parameter pack is called a **variadic template**.

A variadic class template can be instantiated with any number of template arguments:

```
template<typename.. Types>
struct Tuple {};

Tuple<> t0;             // Types contains no arguments
Tuple<int> t1;          // Types contains one argument: int
Tuple<int,float> t2;    // Types contains two arguments: int and float
Tuple<0> t3;            // Error: 0 is not a type
```

A variadic function template can be called with any number of function arguments (the template arguments are deduced through template argument deduction).

```
template <typename... Types>
void f(Types... args);

f();        // OK: args contains no arguments.
f(1);      // OK: args contains one argument: int
f(2,1.0); // OK: args contains two arguments: int and double
```

In a primary class template, the **template parameter pack** must be the final parameter in the template parameter list. In a function template, the template parameter pack may appear earlier in the list provided that all following parameters can be deduced from the function arguments, or have default arguments.

```
template <typename U, typename... Ts>
struct valid; // OK: can deduce U

template <typename... Ts, typename U>
struct invalid; // Error: Ts... Not at the end

template <typename... Ts, typename U, typename=void>
void valid(U, Ts...);       // OK: can deduce U
// Void valid(Ts..., U);    // Can't be used: Ts... is a non-deduced context in this
↪  position

valid(1.0, 1, 2, 3);        // OK: deduces U as double, Ts as {int, int, int}
```

If every valid specialization of a variadic template requires an empty template parameter pack, the program is ill-formed. no diagnostic required.

2.1.1 Pack expansion

A pattern followed by an ellipsis, in which the name of at least one parameter pack appears at least once, is *expanded* into zero or more instantiations of the pattern, where the name of the parameter pack is replaced by each of the elements from the pack, in order. Instantiations of **allignment specifiers** are space-separated, other instantiations are comma-seperated.

```
template <typename... Us>
void f(Us... pargs) {}
template<typename... Ts>
void g(Ts... args) {
    f(&args...);        // "&args..." is a pack expansion
                        // "&args" is its pattern
}
```

If the names of two parameter packs appear in the same pattern, they are expanded simultaneously, and they must have the same length:

```
template<typename...>
struct Tuple {};

template<typename T1, typename T2>
struct Pair {};

template<typename... Args1>
struct zip {
    template<typename... Args2>
    struct with {
        typedef Tuple<Pair<Args1, Args2>...> type;

    };
};
```

Printing w/ template packs

If you are using C++17 or later, there is a feature called **fold expressions** that make it very simple to print your parameter packs.

```
// Variadic template function to print multiple arguments
template<typename... Args>
void print(Args... args) {
(std::cout << ... << (args, " ")) << std::endl;
}
```

Without C++17, recursion must be used. This means that we need a base case with zero arguments, and a recursive case with 1 explcit argument and a tail consisting of a variadic list of arguments.

```
// Base case with 0 arguments
void print(){
std::cout << std::endl;
}
// Recursive case
template<typename Tp, typename... Args>
void print(Tp const &head, const &Args... tail) {
    std::cout << head;
    if (sizeof...(tail)) {
        std::cout << ", ";
    }
    print(tail...);
}
```

Note:-

the `sizeof...` `operator` Queries the number of elements in a `parameter pack`

Example for std::vector's emplace

When implementing `emplace()` for your vector class, You should be using **variadic templates** in order to give the constructor all the members of the type. Therefore, your emplace function might look like this:

```
template <typename it, typename... Args>
iterator vector::emplace(it&& position, Args&&... args) {
    // Check for full container
    if (size == capacity) {
        (capacity) ? reserve(capacity * 2) : reserve(1);
    }

    ptrdiff_t offset = position - begin();

    // Move all the elements to the right
    for (ptrdiff_t i = static_cast<ptrdiff_t>(m_size); i > offset; --i) {
        m_data[i] = m_data[i - 1];
    }
    // Using placement new
    new (m_data + offset) value_type(std::forward<Args>(args)...);
    ++m_size;
    position = m_data + offset;

    return position;

}
// Calling emplace.
struct items{
    int x;
    double y;
    char k;

    items(int x=0, double y=0, char k='F') : x(x), y(y), k(k);
};
vector<items> vec{1,2,3,4};
vec.emplace(1,2.0, 'a');
```

Now, we can call emplace with the precise amount of data to match our types data members.

3 9-7-24

3.1 What happens when you write a lambda expression

The standard spells this out quite nicely. It states that the closure type for a *lambda-expression* has a *public inline function call operator* (for a non-generic lambda) or *function call operator template* (for a generic lambda) whose parameters and return type are described by the lambda-expression's *parameter-declaration clause.* and *trailing-return-type* respectively, and whose *template paramter-list* consists of the specified *template-parameter-list*, if any.

For example, this:

```
[](const Person &lhs, const Person rhs) {
  return lhs.name < rhs.name;
}
```

Would look something like this:

```
struct __lambda_1 {
  inline bool operator()(const Person &lhs, const Person &rhs) const {
```

```
    return lhs.name < rhs.name;
  }
};
__lambda_1();
```

Note that this is *some approximation* of what the compiler generates. Additionally, the *call operator* is `const` by default, so if you want to modify members, you need to write `mutable`.

Lambda expressions with an empty **capture clause** are implicitly converted to *function pointers*. So something like this is valid code:

```
void legacy_call(int(*f)(int)) {
  std::cout << f(7) << '\n';
}
int main() {
  legacy_call([](int i) {
    return i * i;
  }) // Prints 49
}
```

This is also spelled out by the standard. It states that The closure type for a non-generic *lambda-expression* with no *lambda-capture* whose constraints (if any) are satisfied has a conversion function to pointer to function. Here is what that might look like:

```
  struct __lambda_1 {
    inline bool operator()(const Person &lhs, const Person &rhs) {
      return lhs.name < rhs.name;
    }
    __lambda_1() = delete; // not default-constructible.
    __lambda_1& operator=(const __lambda_1&) = delete // not copyable or assignable.

    using __func_type = bool(*)(const Person&, const Person&);
    inline operator __func_type() const noexcept {
    return &__invoke;
  }
  private:
  static inline bool __invoke(const Person &lhs, const Person &rhs) {
    return lhs.name < rhs.name;
  }
};
```

3.2 captures

For each entity captured by copy, an unnamed non-static data member is declared in the closure type. The order of these members is unspecified.

```
int i = 0;
int j = 0;
auto f = [=] {
  return i == j;
};
```

Would look like this:

```
struct __lambda {
  __lambda_2(int i, int j) : __i(i), __j(j) { }

  inline bool operator()() const {
    return __i == __j;
```

```
  }
  private:
    int __i;
    int __j;
}
__lambda_2(i, j);
```

For each entity captured by reference, the type of the data member will be a referenced type if the entity is a reference to an object, an lvalue reference to the referenced function if the entity is a reference to a function, or the type of the corresponding captured entity otherwise.

Note:-

A member of an anonymous union shall not be captured by copy.

Here is an example of capture by reference:

```
auto f = [&] {
  return i == j;
}
struct __lambda {
  __lambda_2(int &i, int &j) : __i(i), __j(j) { }

  inline bool operator()() const {
    return __i == __j;
  }
  private:
  int &__i;
  int &__j;
}
__lambda_2(i, j);
```

3.3 Immediatly invoked lambdas

With lambda expressions, if you're not creating them in a function that takes a unary function object. You need to pair it with `auto` followed by some arbirary name that can be used to invoke the lambda expression. Writing something like this makes very little sense.

```
[]() { std::cout << "Hello, World!"; }; // Unused.
```

However, it is possible to have create lambda expressions that are immediatly called. They look like this:

```
[] { std::cout << "Hello, World!"; }(); // IIFE
```

As you can see, placing the double parentheses after the body of the lambda will immediatly call the lamda. Therere it does not need a callable name.

Now, lets say you default construct some object of type `Foo` and need to assign it based on the results of some external factor.

```
Foo foo;
if (has_database) {
  foo = get_foo_from_database;
}
else {
  foo = get_foo_from_elsewhere;
}
```

Instead of doing it like this, we can do this:

```
  Foo foo = [&] {
  if (has_database) {
    return get_foo_from_database();
  }
  else {
    return get_foo_from_elsewhere();
  }();
}
```

Now, we are declaring and defining at the same time. Thus, there is no need to worry about if we need foo to be const. Or if foo is even default constructable in the first place.

If we wanted the code to be more readable, we could wrap the lambda in `std::invoke` instead of double parentheses at the rear.

3.4 Generic lambdas

Lambda support forward references & perfect forwarding

As it turns out, we can write something like this:

```
std::vector<std::string> v;

auto f = [&v](auto&& item) {
  v.push_back(std::forward<decltype(item)>(item));
};
```

This is a lambda that takes a universal reference (`auto&& item`) and uses perfect forwarding to insert the item in the back of the container. Note that if push_back took an lvalue reference, this would not work. Since we are forward is converting `item` back to a rvalue.

Variadic lambdas

lambda also support parameter packs. So we can do something like this:

```
auto f = [](auto&&... args) {
  (std::cout << ... << args)  // Fold expression from c++17
}
f(1, "Hello", 1.5);
```

Note that this is using fold expression. Pre c++17 would require some recursion to get this to work. Thankfully, c++14 made recursive lambdas possible, the trick for recursive lambdas is to pass the lambda to itself.

```
auto print_stuff = [](auto&&... args) {
  auto print = [] (auto &ff, auto&& head, auto&&... tail) {
      std::cout << head;
      if constexpr (sizeof...(tail)) {
        std::cout << ", ";
        ff(ff,std::forward<decltype(tail)>(tail)...);
      }
  }; print(print, std::forward<decltype(args)>(args)...); };
```

4 9/12/24

4.1 Aggregate types

4.2 Structured bindings

Binds the specified names to subobjects or elements of the initializer. Like a reference, a structured binding is an alias to an existing object. Unlike a reference, a structured binding does not have to be a reference type.

attr(optional) *cv-auto ref-qualifier*(optional)[identifier-list] = *expression*; (1)

attr(optional) *cv-auto ref-qualifier*(optional)[identifier-list]{*expression*} (2)

attr(optional) *cv-auto ref-qualifier*(optional)[identifier-list](*expression*) (3)

attr - sequence of any number of **attributes**

cv-auto - possibly cv-qualified type specifier `auto`, may also include **storage class specifier** `static` or `thread_local`; including `volatile` in cv-qualifiers is deprecated (since C++ 20)

ref-qualifier - either `&` or `&&`

identifier-list - list of comma-seperated identifiers.

expression - an expression that does not have the comma operator at the top level (gramatically, an *assignment-expression*), and has either array or non-union class type. If ***expression*** refers to any of the names from ***identifier-list***, the declaration is ill-formed.