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.