Defining Abstract Data Types

Laurent Mathy

Object-Oriented Programming Projects

April 17, 2020

Class design

We look at a simplified `Vec` class to investigate language support for type design.

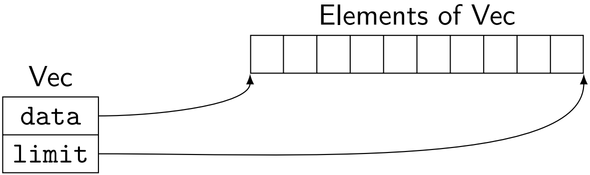
What kind of interface do we need? *E.g.*

```
// Construction
vector<Student_info> vs;
vector<double> v(100);

// Type names
vector<Student_info>::const_iterator b, e;
vector<Student_info>::size_type i = 0;

// Size() and indexing
for (i = 0; i != vs.size(); ++i)
    cout << vs[i].name();

// Iterator positions
b = vs.begin();
e = vs.end();
```

Vec class principles

Template facility also works for classes: compiler will instantiate a class with appropriate type when we declare instances.

Internals:




```
template<class T> class Vec {
public:
    // Interface
private:
    T* data;
    T* limit;
};
```

Constructors

```cpp
template<class T> class Vec {
public:
    Vec() { /* TODO: allocate and initialise empty array */ }
    explicit Vec(std::size_t n, const T& t = T()) {
        // TODO: Allocate array and initialise it
        //       with n copies of t
    }
private:
    T* data;  // First element in the `Vec`
    T* limit; // One past the allocated memory
};
```

Second constructor is `explicit`:

- For constructors that can receive a *single* argument.
- Compiler will use it only in context where user explicitly requested it – never used for implicit type conversion!

```cpp
Vec<int> vi(100);  // Explicitly constructs a Vec from an int
Vec<int> vi = 100; // Compile error: implicit conversion
```

Type definitions

```
template<class T> class Vec {
public:
    typedef T* iterator;
    typedef const T* const_iterator;
    typedef std::size_t size_type;
    typedef T value_type;
    typedef T& reference;
    typedef const T& const_reference;

    Vec() { /* ... */ }
    explicit Vec(size_type n, const T& t = T()) {
        /* ... */
    }
private:
    iterator data;  // First element in the `Vec`
    iterator limit; // One past the allocated memory
};
```

Size and Index

`size` function must be a member function.

```
size_type size() const { return limit - data; }
```

Operator overloading:

- Overloaded operator defined like a function.
- Special name: `operator` then append operator symbol.
- If operator is not a member function: function takes as many arguments as operator has operands:
  - First argument bound to left operand;
    second argument bound to right operand.
- If operator defined as member function: left operand implicitly bound to the object.
- Index operator **must** be a member function.

```
T& operator[](size_type i) { return data[i]; }
const T& operator[](size_type i) const {
    return data[i];
}
```

Returning iterators

Again, we need `const` and non-`const` versions:

```
iterator begin() { return data; }
const_iterator begin() const { return data; }

iterator end() { return limit; }
const_iterator end() const { return limit; }
```

Copy control

Passing object by value to function, or returning object by value from function, implicitly copies the object.

You control both explicit and implicit copies through **copy constructor**: normal constructor that takes a single argument of same type as class.

In fact, since we are defining what it means to make copies, including for function arguments, this parameter for the copy constructor **must** be a reference type!

```
Vec(const Vec& v) { create(v.begin(), v.end()); }
```

- What we want is a **deep** copy of the original vector: we want a copy of the array.
- A **shallow** copy (sharing the array) would result in disaster.
- We will define the `create` function later.

Assignment operator

The overloaded `operator=` that takes a `const` reference to the class itself is the **assignment operator**.

It has one big difference compared with the copy constructor: it always obliterates the content of its left-hand operand.

It **must** be a class member

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

```
template<class T> Vec<T>& Vec<T>::operator=(const Vec& rhs) {
    if (&rhs != this) {
        uncreate();
        create(rhs.begin(), rhs.end());
    }
    return *this;
}
```

- Within scope of the template, we can omit the <T>.
- `this` is pointer to the object member function is operating on.
- `*this` is the object itself; we return a reference to this object.

Assignment vs Initialisation

Assignment **always** obliterates a previous value; initialisation **never** does so.

Initialisation:

- Variable declaration.
- Function parameters on function entry.
- Return value on function exit.
- Constructor initialisers.

```
string url_ch = "~;/?:@=&$-_.+!*'(),"; // Initialisation
string spaces(url_ch.size(), ' ');     // Initialisation
string y;   // Initialisation
y = url_ch; // Assignment
```

Note: assigning class type return value from function is done in two steps:

- Copy constructor is run to copy return value into a temp.
- Assignment operator is run to copy temp into left-hand side.

Destructor

Object created in local scope is destroyed when leaving this scope.

Dynamically allocated objects are destroyed when `delete` is called.

We should say what happens on object destruction, through **destructor** who does the clean up.

```
~Vec() { uncreate(); }
```

Default operations

If we do not define **any** constructor, the compiler synthesises a default constructor for us:

- All data members are default/value initialised, recursively.

Default versions synthesised for copy constructor, assignment operator and destructor if not explicitly defined:

- Designed to work recursively.
- Built-in types are simply copied/assigned.
- No work to be done for built-in type destruction.
- Corresponding operations called on other data members.

Beware of *shallow* operations: copying a pointer shares the content.

Beware of memory leaks: destroying a pointer does not destroy the content.

It is generally a good idea to make sure there is a default constructor – otherwise, type may not be member of other type for which default constructor is synthesised.

The rule of Three

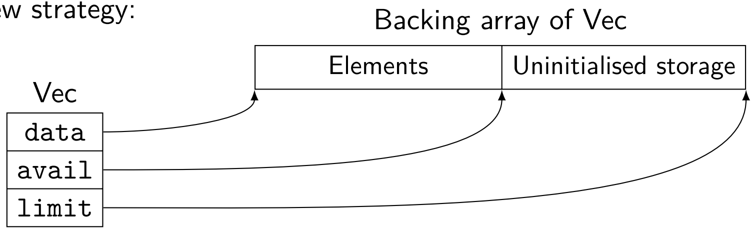
If your class needs a **destructor**, it likely also needs a **copy constructor** and an **assignment operator**.

Failure to respect this requirement will result in strange behaviour or crash!

Dynamic Vecs

We need to support a `push_back` operation.

New strategy:




```cpp
public:
    size_type size() const { return avail - data; } // Changed
    iterator end() { return avail; }                // Changed
    const_iterator end() const { return avail; }    // Changed
    void push_back(const T& t) {                    // New
        if (avail == limit)
            grow();
        unchecked_append(t);
    }
private:
    iterator data;
    iterator avail;                                 // New
    iterator limit;
```

Memory management

new allocates **and initialises** memory, so type `T` would need a default initialiser.

This is bad both from flexibility and performance point-of-view!

Use `allocator<T>` in `<memory>` instead to allocate without initialisation.

```
template<class T> class Allocator {
public:
    T* allocate(size_t n); // Allocates enough space for n objects of type T
    void deallocate(T*, size_t n); // Deallocates space equal to n objects of type T
    void construct(T*, const T&); // Initialises a T into allocated space
    void destroy(T*); // Destroys object of type T, storage become uninitialised
    // ...
};

template<class In, class For>
For uninitialized_copy(In, In, For); // Copies range into destination

template<class For, class T>
void uninitialized_fill(For, For, const T& t); // Construct copies of t to fill range
```

We will use a new data member: `allocator<T> alloc;` to access these facilities.

Memory management functions

```
template<class T>
void Vec<T>::create() {
    data = limit = avail = nullptr;
}

template<class T>
void Vec<T>::create(size_type n, const T& val) {
    data = alloc.allocate(n);
    limit = avail = std::uninitialized_fill_n(data, n, val);
}

template<class T>
void Vec<T>::create(const_iterator b, const_iterator e) {
    data = alloc.allocate(e - b);
    limit = avail = std::uninitialized_copy(b, e, data);
}
```

Memory management functions (2)

```
template <class T>
void Vec<T>::uncreate() {
    if (data) {
        // Destroy initalialised elements in reverse order
        iterator it = avail;
        while (it != data)
            alloc.destroy(--it);

        // Return all the space that was allocated
        alloc.deallocate(data, limit - data);
    }
    // Reset pointers to indicate this `Vec` is empty
    data = limit = avail = nullptr;
}
```

Memory management functions (3)

```
template<class T> void Vec<T>::grow() {
    // When growing, allocate twice as much space as currently in use
    size_type new_size = max(2 * (limit - data), std::ptrdiff_t(1));

    // Allocate new space and copy existing elements to the new space
    iterator new_data = alloc.allocate(new_size);
    iterator new_avail = std::uninitialized_copy(data, avail, new_data);

    // Return the old space
    uncreate();

    // Reset pointers to point to the newly allocated space
    data = new_data;
    avail = new_avail;
    limit = data + new_size;
}

// Assumes `avail` points at allocated, but uninitialized space
template <class T> void Vec<T>::unchecked_append(const T& val) {
    alloc.construct(avail++, val);
}
```