

# Defining Abstract Data Types

Laurent Mathy

Object-Oriented Programming Projects

April 20, 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.*

---

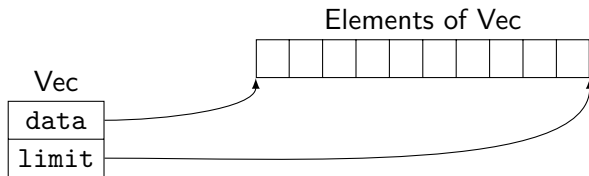
```
1 // Construction
2 vector<Student_info> vs;
3 vector<double> v(100);
4
5 // Type names
6 vector<Student_info>::const_iterator b, e;
7 vector<Student_info>::size_type i = 0;
8
9 // Size() and indexing
10 for (i = 0; i != vs.size(); ++i)
11     cout << vs[i].name();
12
13 // Iterator positions
14 b = vs.begin();
15 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:



---

```
1  template<class T> class Vec {  
2  public:  
3      // Interface  
4  private:  
5      T* data;  
6      T* limit;  
7  };
```

---

# Constructors

---

```
6  template<class T> class Vec {
7  public:
8      Vec() { /* TODO: allocate and initialise empty array */ }
9      explicit Vec(std::size_t n, const T& t = T()) {
10         // TODO: Allocate array and initialise it
11         //         with n copies of t
12     }
13 private:
14     T* data; // First element in the `Vec`
15     T* limit; // One past the allocated memory
16 };
```

---

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!

---

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

---

## Type definitions

```
6  template<class T> class Vec {
7  public:
8      typedef T* iterator;
9      typedef const T* const_iterator;
10     typedef std::size_t size_type;
11     typedef T value_type;
12     typedef T& reference;
13     typedef const T& const_reference;
14
15     Vec() { /* ... */ }
16     explicit Vec(size_type n, const T& t = T()) {
17         /* ... */
18     }
19 private:
20     iterator data; // First element in the `Vec`
21     iterator limit; // One past the allocated memory
22 };
```

## Size and Index

size function must be a member function.

---

```
23 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.

---

```
25 T& operator[](size_type i) { return data[i]; }  
26 const T& operator[](size_type i) const {  
27     return data[i];  
28 }
```

---

## Returning iterators

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

---

```
30 iterator begin() { return data; }
31 const_iterator begin() const { return data; }
32
33 iterator end() { return limit; }
34 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!

20

---

```
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

---

```
36 Vec& operator=(const Vec&);  
  
50 template<class T> Vec<T>& Vec<T>::operator=(const Vec& rhs) {  
51     if (&rhs != this) {  
52         uncreate();  
53         create(rhs.begin(), rhs.end());  
54     }  
55     return *this;  
56 }
```

---

- 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.

---

```
1 string url_ch = "~;/?:@=&$-_.+!*'(),"; // Initialisation
2 string spaces(url_ch.size(), ' ');      // Initialisation
3 string y; // Initialisation
4 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.

---

21 `~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

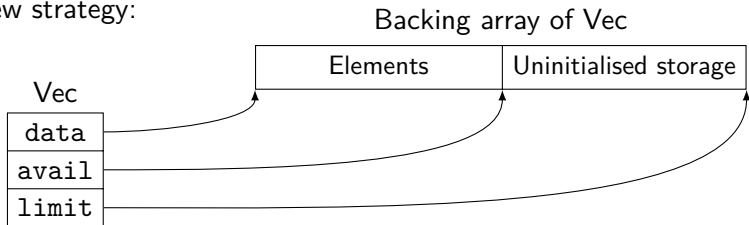
If your class needs a **destructor**, it likely also needs a **copy constructor** and an **assignment operator**.

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

## Dynamic Vecs

We need to support a `push_back` operation.

New strategy:



---

```
1 public:
2     size_type size() const { return avail - data; } // Changed
3     iterator end() { return avail; } // Changed
4     const_iterator end() const { return avail; } // Changed
5     void push_back(const T& t) { // New
6         if (avail == limit)
7             grow();
8         unchecked_append(t);
9     }
10 private:
11     iterator data;
12     iterator avail; // New
13     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.

---

```
1  template<class T> class Allocator {
2  public:
3      T* allocate(size_t n); // Allocates enough space for n objects of type T
4      void deallocate(T*, size_t n); // Deallocates space equal to n objects of type T
5      void construct(T*, const T&); // Initialises a T into allocated space
6      void destroy(T*); // Destroys object of type T, storage become uninitialised
7      // ...
8  };
9
10 template<class In, class For>
11 For uninitialized_copy(In, In, For); // Copies range into destination
12
13 template<class For, class T>
14 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

```
67  template<class T>
68  void Vec<T>::create() {
69      data = limit = avail = nullptr;
70  }
71
72  template<class T>
73  void Vec<T>::create(size_type n, const T& val) {
74      data = alloc.allocate(n);
75      limit = avail = std::uninitialized_fill_n(data, n, val);
76  }
77
78  template<class T>
79  void Vec<T>::create(const_iterator b, const_iterator e) {
80      data = alloc.allocate(e - b);
81      limit = avail = std::uninitialized_copy(b, e, data);
82  }
```



## Memory management functions (2)

```
84  template <class T>
85  void Vec<T>::uncreate() {
86      if (data) {
87          // Destroy initialised elements in reverse order
88          iterator it = avail;
89          while (it != data)
90              alloc.destroy(--it);
91
92          // Return all the space that was allocated
93          alloc.deallocate(data, limit - data);
94      }
95      // Reset pointers to indicate this `Vec` is empty
96      data = limit = avail = nullptr;
97  }
```

## Memory management functions (3)

```
99  template<class T> void Vec<T>::grow() {
100     // When growing, allocate twice as much space as currently in use
101     size_type new_size = max(2 * (limit - data), std::ptrdiff_t(1));
102
103     // Allocate new space and copy existing elements to the new space
104     iterator new_data = alloc.allocate(new_size);
105     iterator new_avail = std::uninitialized_copy(data, avail, new_data);
106
107     // Return the old space
108     uncreate();
109
110     // Reset pointers to point to the newly allocated space
111     data = new_data;
112     avail = new_avail;
113     limit = data + new_size;
114 }
115
116 // Assumes `avail` points at allocated, but uninitialized space
117 template <class T> void Vec<T>::unchecked_append(const T& val) {
118     alloc.construct(avail++, val);
119 }
```