19. Vectors & Templates

vector: create
copy: asgt & init
release mem
access []

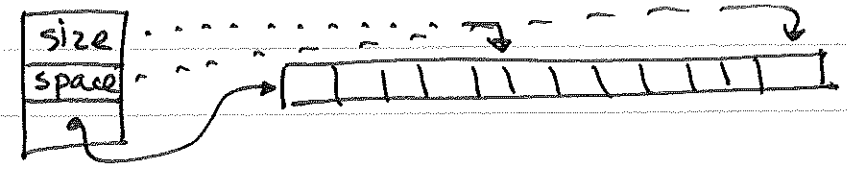
more: change nr elements
report out of range
spec elt type

ctrl
move ctor
~~no op~~
move op=

ex: vector<double> d
d.push_back(x)
d.resize(n)

alloc space – too little → run out
too much → waste memory

representation



size – amt used
space – ~~len~~ of array

push_back – usu O(1)
– sometimes O(n)

but w. are doubling
amortized → O(1) average

ch. 19
2

19.2.2 Reserve & capacity

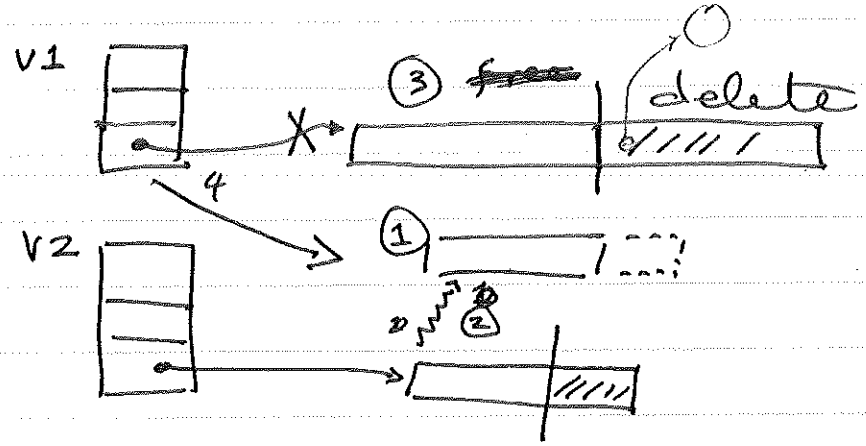
```
size_t vector::capacity() const {
        return space
}

void vector::reserve(size_t cap){
        if(cap <= space) return;
        double *p = new double[cap]
        for // loop copy elements
        delete [] data
        data = p
        space = cap
}
```

space - size = #elmts avail space

19.2.4 push_back

```
void vector::push_back(double d){
        if(space==0) reserve(8)
        else if(size==space) reserve(2*space)
        data[size++] = d
}
```

19.2.5 op=



v1 = v2; if (this ≠ that)

(1) alloc new space = size v2
(2) copy v2 elts to new alloc
(3) delete [] v1 data
(4) v1 data → new alloc

19.3 Templates

```
template <typename T>          // class
class vector {
    ≡
};
```

vector<char>
vector<double>
⋮

- no problems w. primitives

19.3.2 Generic Prog

ch 19
4

- works with a variety of types

Polymorphism — ad-hoc ← conversion
overloading

universal — templates (generics)

inheritance
overriding

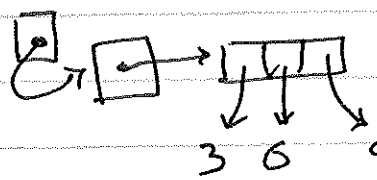
templates — static
- compile time
- recompilation

inheritance — dynamic
- runtime dispatch

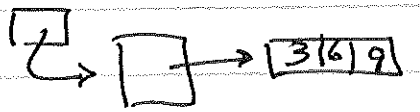
Java class — compiled once as Object
Generic — can't have primitives
- must use boxes like Integer

C++ template = macros
- can lead to code bloat
- faster
- uses less data space

Java



C++



19.3.3 Containers & Inheritance

vector<base*> = ok
vector<base> fails due to slicing

19.3.4 Integer Templates

```
template <typename T, size_t N>
class array {
    T a[N];
};
```

- useful in limited memory systems without free store
- can alloc static memory for it
- vector can't.

```
array<int, 10> x1;
array<double, 20> x2;
```

- not so useful for complicated objects.

19.3.6 Generalizing

Problems:
- vector<X>, but X has no default value
- elements destroyed when finished

- types w/o default:

```
template<class T>
void vector<T>::resize(size_t n, T def=T());
```

— use T() as default.
note int() == 0
double() == 0.0

- uninit data
data struct has both init & uninit data

```
template<class T, class A = allocator<T>>
class vector {
```

class allocator in STL has:

allocate, deallocate	memory
construct, destroy	objects

19.4 Exceptions

exception (hierarchy)

- bad_alloc
- bad_cast
- bad_exception
- bad_typeid
- ios_base::failure
- logic_error
  - domain_error
  - invalid_argument
  - length_error
  - out_of_range
- runtime_error
  - overflow_error
  - range_error
  - underflow_error

(std::)

catch(exception &)

- by reference to avoid slicing
- not by value

op[] vs at

- compatibility
- efficiency
- constraints

what about real time prog?

- hard
- soft

19.5 Resources

- memory
- locks
- file handles
- thread handles
- sockets
- windows

Resource Problems.

~~void~~ foo *p = new foo()
- where delete

```
void f() {
    foo *p = new foo
    ≡
if() p = Q
    ≡
    delete p // what o by ?
}
```

```
if() return
delete p
```
never got here

exception

```
void f() {
    foo *p = new
    ≡
    throw exn
    ≡
    delete p // never got here.
```

```
again() {
    foo *p = new...
    try {
        ≡
    } catch( ) {
        delete p
        throw;
    }
    delete p
}
```

← rethrow caught exn.

19.5.2 RAII

resource acquisition is initialization.

```
beller() {
    foo a;
    foo b
    ⋮
}
```

← free objs @ return or exn whenever a, b exit scope

~~19.5.3 Guarantee~~

precondition
postcondition
invariant

ch 19
10

19.5.3 Guarantees

problem: alloc structs & return
∴ need a pointer

Basic guarantee: succeed or throw
but no leaked resource.

Strong guarantee: all other resources
are same after failure

No throw guarantee:
- avoid throw, new, dynamic cast.
- avoid deref ∅, ÷0, etc.

19.5.4 auto_ptr

- deprecated in C++11.

```
vector *make( ... ) {
    auto_ptr< vector< int >> p (new vector<int>)
    ≡
    return p.release();
}
```

C++11
shared_ptr
unique_ptr.

19.5.5 ~~RAII~~ RAII

memory for vector is resource

~~allocator~~

```
template <class T> class allocator {
        T *allocate (int n)
        void deallocate(T *p, int n)
        Void construct(T *p, const T &v)
        void destroy (T *p)
```