Demystifying vector<T>

So what *is* a vector?

It seems like a normal finite size type

...but it can contain any number of values.

We now have ***almost*** enough to build our own vector

My vector : a pointer and a count

```
struct my_int_vec{
    int *elements;
    int count;
};
```

My vector : getting the size

```
struct my_int_vec{
    int *elements;
    int count;
};

int size(my_int_vec *v)
{
    return v->count;
}
```

My vector : modifying an element

```
struct my_int_vec{
    int *elements;
    int count;
};

void write(my_int_vec *v, int index, int value )
{
    v->elements[index] = value;
}
```

My vector : reading an element

```
struct my_int_vec{
    int *elements;
    int count;
};

int read(my_int_vec *v, int index)
{
    return v->elements[index];
}
```

My vector : a type plus functions

```
struct my_int_vec;

int  size( my_int_vec *v);
int  read( my_int_vec *v, int index);
void write(my_int_vec *v, int index, int value);
```

The type + functions provides an ***API***

API : Application Programming Interface

Can use the functionality without knowing the details

My vector : a type plus functions

```
struct my_int_vec;

int  size( my_int_vec *v);
int  read( my_int_vec *v, int index);
void write(my_int_vec *v, int index, int value);
```

Why bother with:

```
write(&v, 3, 15);
```

rather than doing it directly?:

```
v->element[3] = 15;
```

My vector : adding checks

```
void write(my_int_vec *v, int index, int value )
{
  v->elements[index] = value;
}
```

```
void write(my_int_vec *v, int index, int value )
{
  if( index < 0 || v->count < index ){
    cerr << "Error : index out of range." << endl;
    exit(1); // Immediately quits program with error code
  }
  v->elements[index] = value;
}
```

My vector : a type plus functions

```c
struct my_int_vec;

int  size( my_int_vec *v);
int  read( my_int_vec *v, int index);
void write(my_int_vec *v, int index, int value);
```

Why pass a pointer to the vector:

```c
write(&v, 3, 15);
```

rather than passing a copy of the vector?:

```c
write(v, 3, 15);
```

My vector : adding features

```
struct my_int_vec{
    int *elements;
    int count;
    int total_reads; // Track the number of read ops
};

int read(my_int_vec *v, int index)
{
    v->total_reads += 1;
    return v->elements[index];
}
```

We can modify the vector during operations.

All existing code that uses `my_int_vec`

immediately gets access to new features

Open questions

- How do we parameterize on the type : `<T>` ?
  - *Templates*: after Christmas
- How do we do `v.size()` rather than `size(&v)` ?
  - *Objects*: after Christmas
- How does it pretend to do array indexing with `[]`?
  - *Overloading* : after Christmas
- How do we create the array behind the vector?
  - *Dynamic allocation*: now

Dynamic Allocation

Scopes and lifetimes : *recap*

- Every instance has a scope and a lifetime
  - ***Scope*** : is the name of the instance valid?
  - ***Lifetime*** : when is the instance created and destroyed?
- So far lifetimes have been managed automatically
  - Lifetime begins when the variable is declared
  - Lifetime ends when the enclosing block finishes
- Scopes and lifetimes are not quite the same
  - An instance can be alive, but not in scope
  - An instance cannot be in scope and not alive

Dangling pointers and lifetimes

Dangling pointers are bugs in lifetime management

```
string *repeat_string(string x, int n)
{
    string s;   // Lifetime and scope of s begins
    for(int i=0; i<n; i++){
        s += x;
    }
    return &s;  // Lifetime and scope of s ends
} // Return value points at ?
```

Dangling pointers and lifetimes

Sometimes we can modify function prototype to fix it

```
// Target of dst is alive before function
string *repeat_string(string *dst, string x, int n)
{
    *dst="";
    for(int i=0; i<n; i++){
        *dst += x;
    }
    return dst;
}
// Target of dst is still alive after function
```

May lead to quite awkward and complicated APIs

Automatic lifetimes come from scope

- So far every instance has had a name
  - The introduction of the name defines the start of scope
  - The exit of the name defines the end of the scope
  - Lifetime automatically follows the scope

- Automatic lifetime variables cannot be “nameless”
  - The same name can appear many times
  - Variables can be in scope but shadowed
    - But... shadowed variables are still alive

Back to my_int_vec

```
struct my_int_vec{
    int *elements;
    int count;
};
```

We have really limited choices for creating a vector

```
int main()
{
    int backing[16];    // Fixed-size storage

    my_int_vec vec={ &backing[0], 16 };

    write(&vec, 3, 42); // Write 42 into index 3
}
```

- The vector's lifetime can't be longer than the backing array
- We can't create vectors with sizes determined at run-time

How do we return a fresh vector?

```
vector<int> iota(int n)
{
  vector<int> res;
  for(int i=0; i<n; i++){
    res.push_back(i);
  }
  return res;
}
```

```
my_int_vec iota(int n)
{
  my_int_vec res={ ????? , n };
  for(int i=0; i<n; i++){
    write(&res, i, i);
  }
  return res;
}
```

Internally vector<int> must point to an array, just like my_int_vec

but

we have no way of creating an array with a lifetime longer than iota

This is quite a deep and fundamental problem

Dynamic allocation to the rescue

- What we have: named instances
  - The “name” and scope the instance controls lifetime
- What we want: ***un-named*** instances
  - A lifetime that is not fixed to any specific name and scope

The solution is *dynamic allocation* of an array:

```
int *p = new int[10];
```

The `new` array operator

p only points at the instance

*It is **not** the name of the new array*

Type of array element

Can be any normal type

```
type *p = new type[size];
```

New unnamed instance

Instance created has no scope

Size of the array

*A **run-time** value – can change for each allocation*

Syntax only: this won’t compile without concrete values for type and size.

A dynamically allocated vector

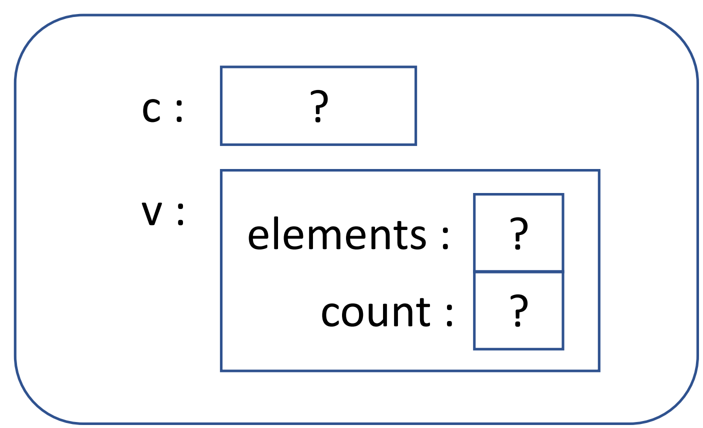
```
my_int_vec iota(int n)
{
  my_int_vec res={ new int[n] , n };
  for(int i=0; i<n; i++){
    write(&res, i, i);
  }
  return res;
}
```

A dynamically allocated vector

```
my_int_vec iota(int n)
{
  my_int_vec res={ new int[n] , n };
  for(int i=0; i<n; i++){
    write(&res, i, i);
  }
  return res;
}

int main()
{
  int c;
  my_int_vec v;
  cin >> c;
  v=iota(c);
  cout << read(&v, c-1);
}
```

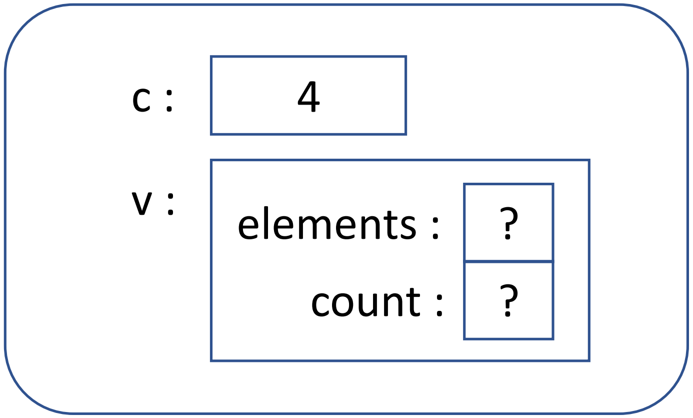
A dynamically allocated vector

```
my_int_vec iota(int n)
{
  my_int_vec res={ new int[n] , n };
  for(int i=0; i<n; i++){
    write(&res, i, i);
  }
  return res;
}

int main()
{
  int c;
  my_int_vec v;
  cin >> c;
  v=iota(c);
  cout << read(&v, c-1);
}
```



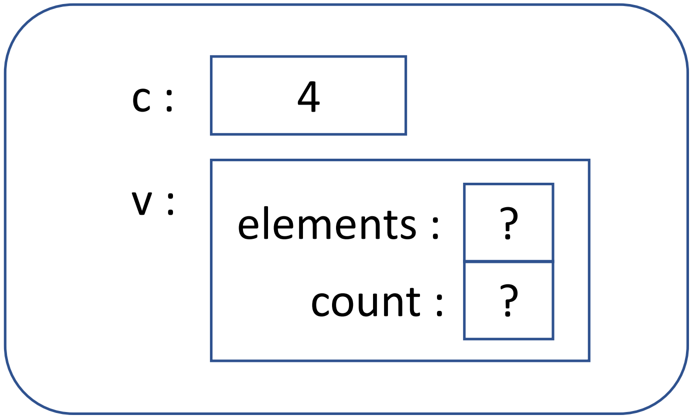
A dynamically allocated vector

```
my_int_vec iota(int n)
{
  my_int_vec res={ new int[n] , n };
  for(int i=0; i<n; i++){
    write(&res, i, i);
  }
  return res;
}

int main()
{
  int c;
  my_int_vec v;
  cin >> c;
  v=iota(c);
  cout << read(&v, c-1);
}
```



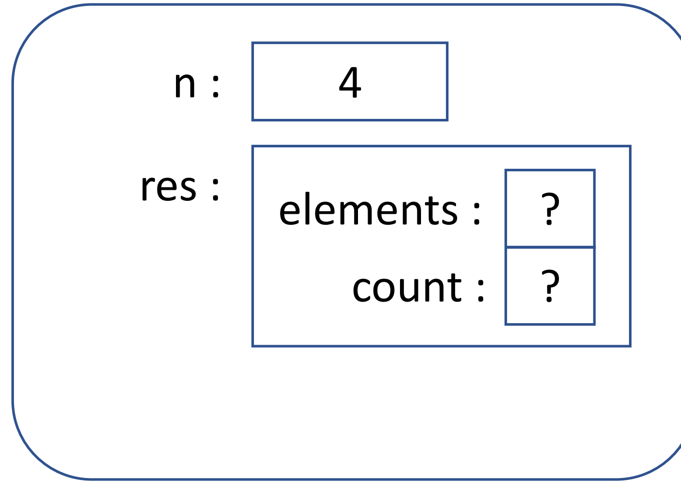
A dynamically allocated vector

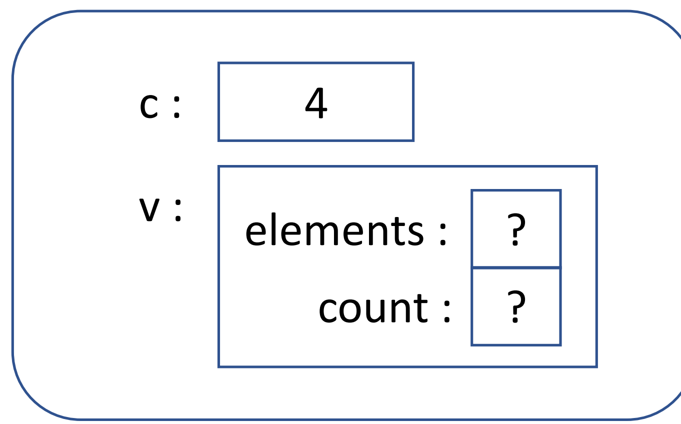
```
my_int_vec iota(int n)
{
  my_int_vec res={ new int[n] , n };
  for(int i=0; i<n; i++){
    write(&res, i, i);
  }
  return res;
}

int main()
{
  int c;
  my_int_vec v;
  cin >> c;
  v=iota(c);
  cout << read(&v, c-1);
}
```



A dynamically allocated vector

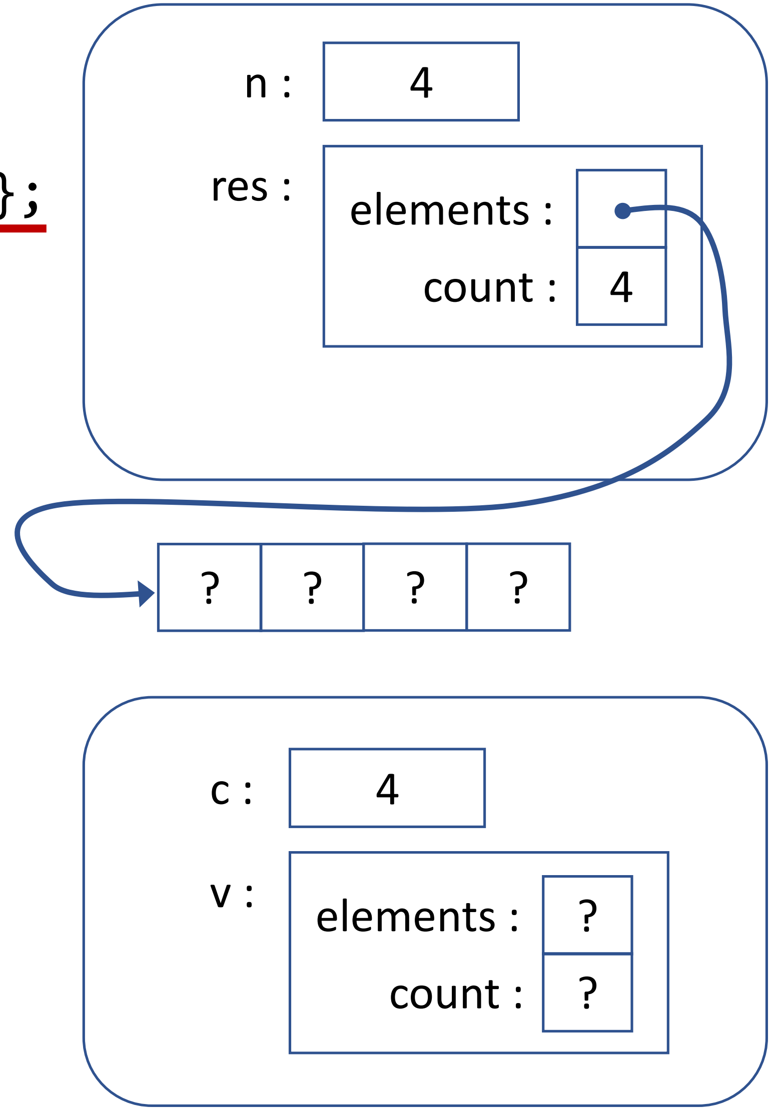
```
my_int_vec iota(int n)
{
  my_int_vec res={ new int[n] , n };
  for(int i=0; i<n; i++){
    write(&res, i, i);
  }
  return res;
}

int main()
{
  int c;
  my_int_vec v;
  cin >> c;
  v=iota(c);
  cout << read(&v, c-1);
}
```





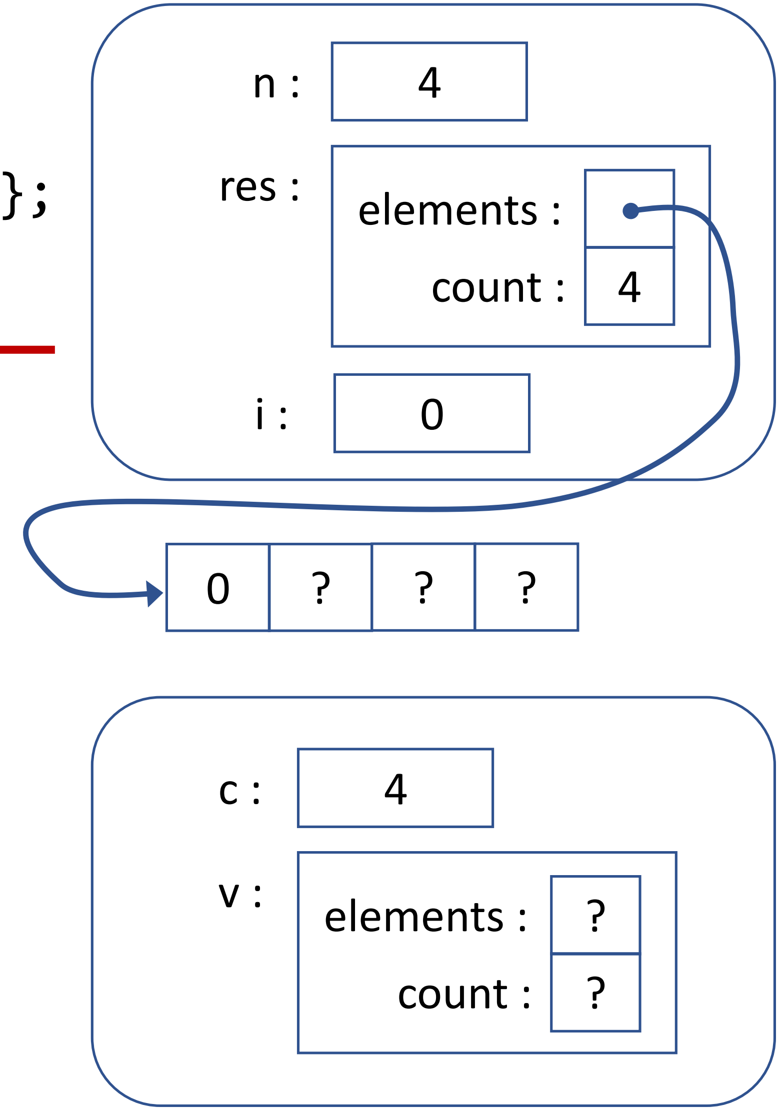
A dynamically allocated vector

```
my_int_vec iota(int n)
{
  my_int_vec res={ new int[n] , n };
  for(int i=0; i<n; i++){
    write(&res, i, i);
  }
  return res;
}

int main()
{
  int c;
  my_int_vec v;
  cin >> c;
  v=iota(c);
  cout << read(&v, c-1);
}
```



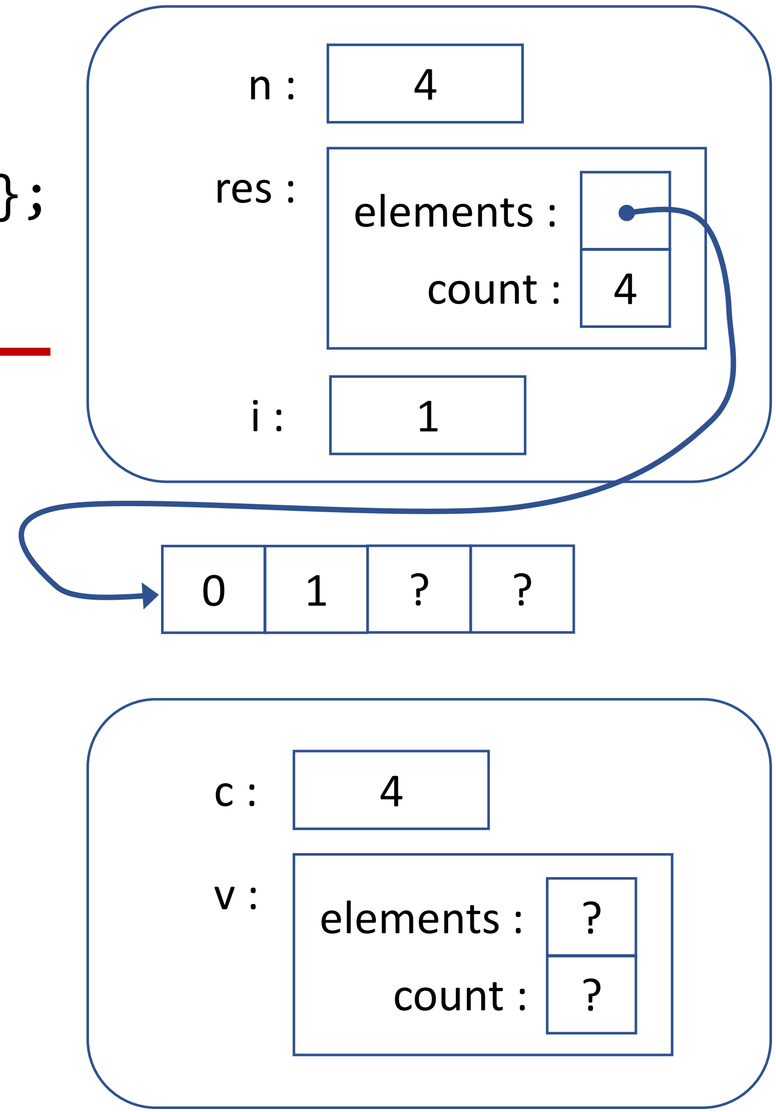
A dynamically allocated vector

```
my_int_vec iota(int n)
{
  my_int_vec res={ new int[n] , n };
  for(int i=0; i<n; i++){
    write(&res, i, i);
  }
  return res;
}

int main()
{
  int c;
  my_int_vec v;
  cin >> c;
  v=iota(c);
  cout << read(&v, c-1);
}
```



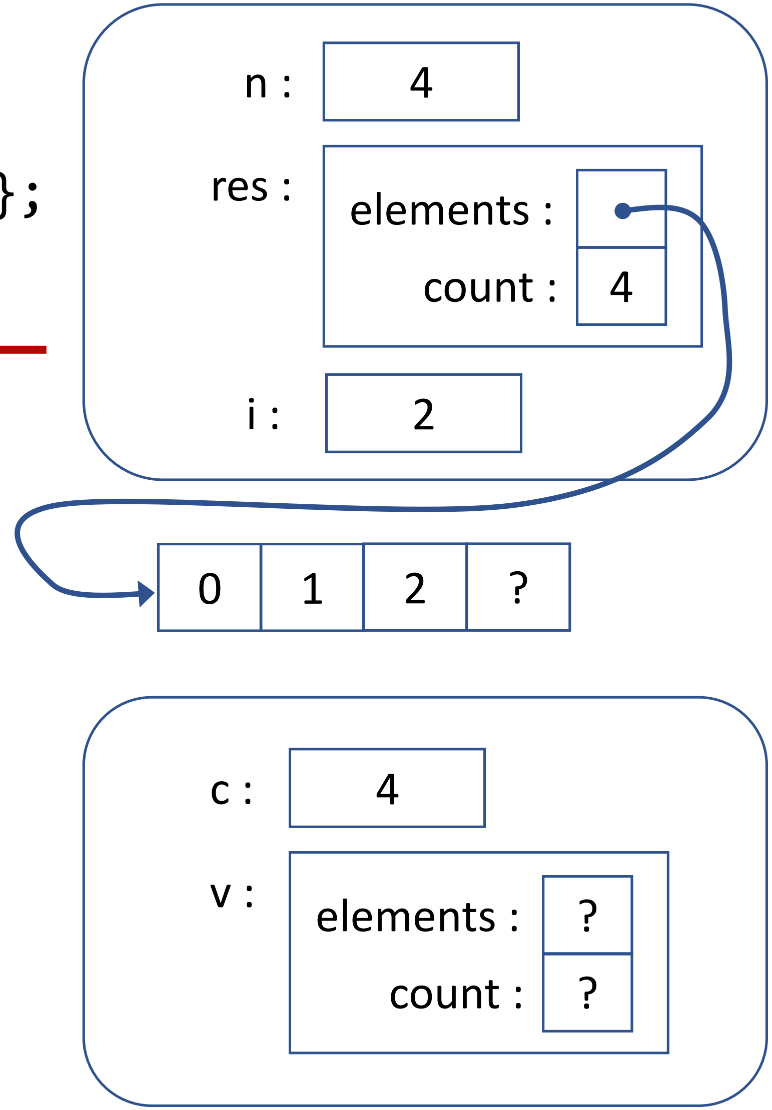
A dynamically allocated vector

```
my_int_vec iota(int n)
{
  my_int_vec res={ new int[n] , n };
  for(int i=0; i<n; i++){
    write(&res, i, i);
  }
  return res;
}

int main()
{
  int c;
  my_int_vec v;
  cin >> c;
  v=iota(c);
  cout << read(&v, c-1);
}
```



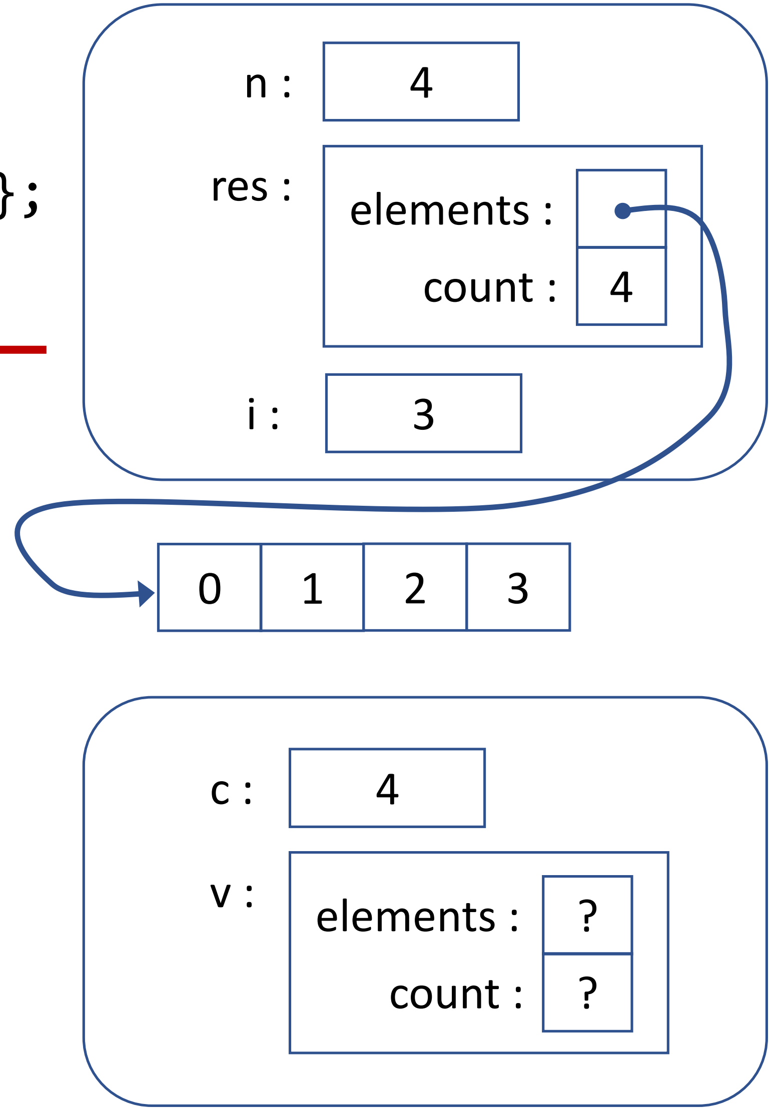
A dynamically allocated vector

```
my_int_vec iota(int n)
{
  my_int_vec res={ new int[n] , n };
  for(int i=0; i<n; i++){
    write(&res, i, i);
  }
  return res;
}

int main()
{
  int c;
  my_int_vec v;
  cin >> c;
  v=iota(c);
  cout << read(&v, c-1);
}
```



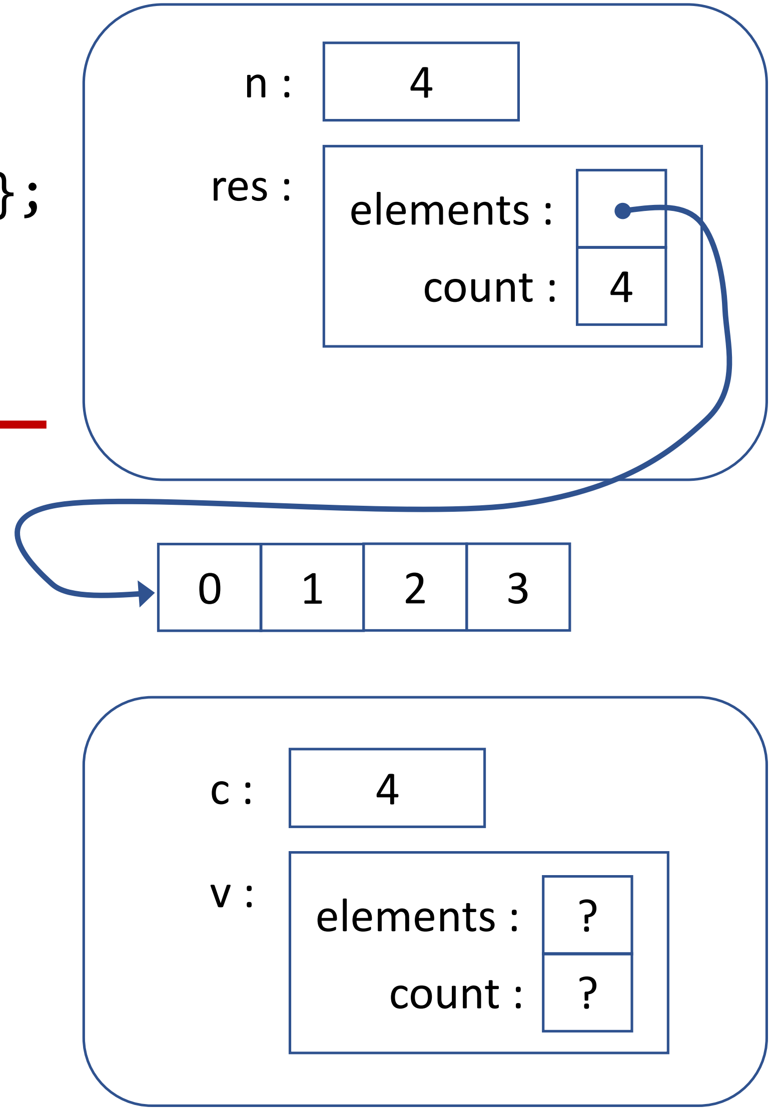
A dynamically allocated vector

```
my_int_vec iota(int n)
{
  my_int_vec res={ new int[n] , n };
  for(int i=0; i<n; i++){
    write(&res, i, i);
  }
  return res;
}

int main()
{
  int c;
  my_int_vec v;
  cin >> c;
  v=iota(c);
  cout << read(&v, c-1);
}
```



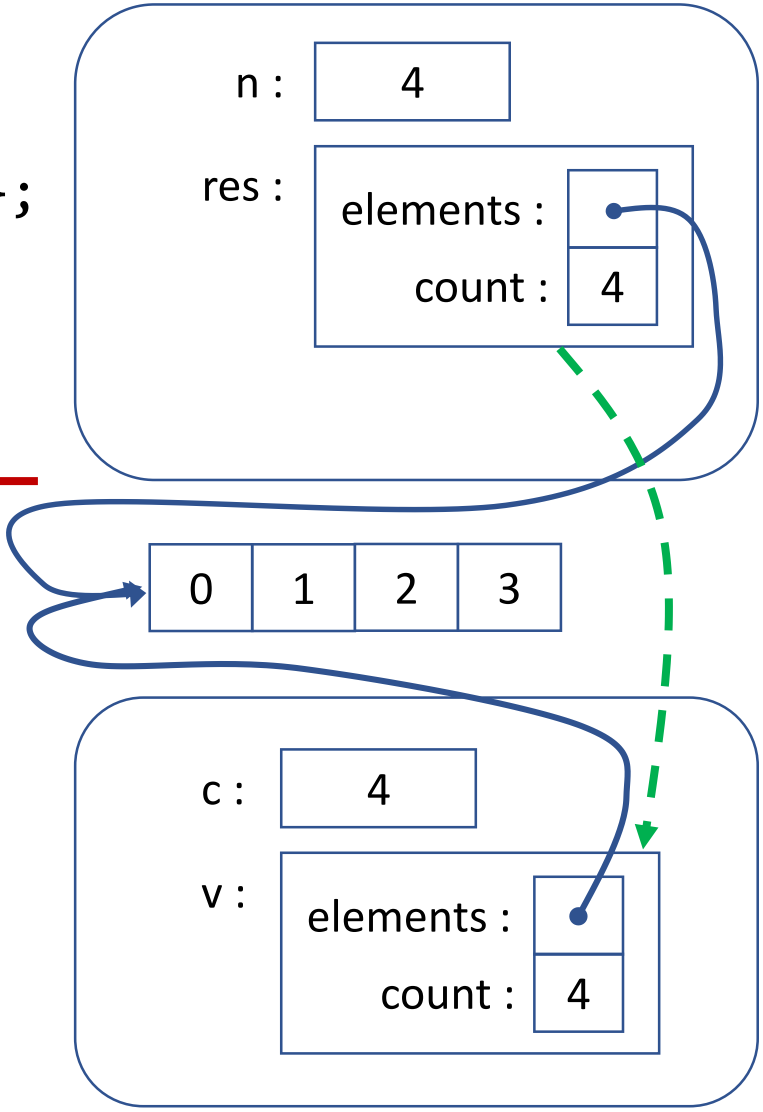
A dynamically allocated vector

```
my_int_vec iota(int n)
{
  my_int_vec res={ new int[n] , n };
  for(int i=0; i<n; i++){
    write(&res, i, i);
  }
  return res;
}

int main()
{
  int c;
  my_int_vec v;
  cin >> c;
  v=iota(c);
  cout << read(&v, c-1);
}
```



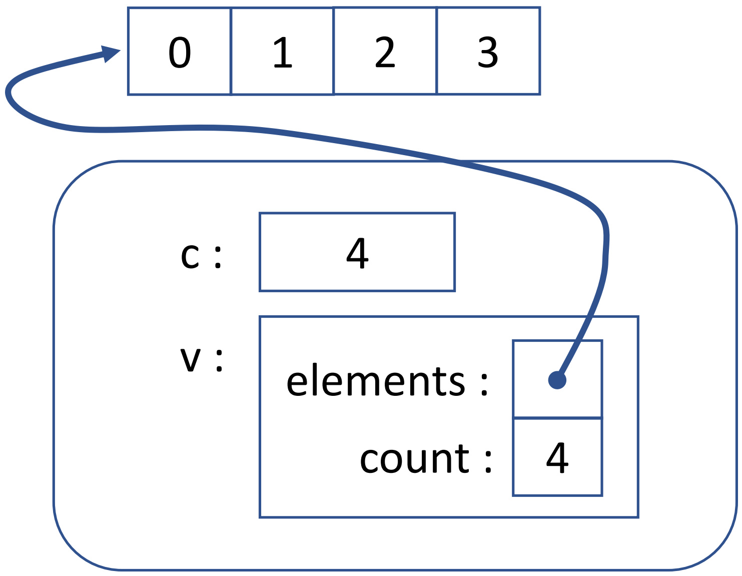
A dynamically allocated vector

```
my_int_vec iota(int n)
{
  my_int_vec res={ new int[n] , n };
  for(int i=0; i<n; i++){
    write(&res, i, i);
  }
  return res;
}

int main()
{
  int c;
  my_int_vec v;
  cin >> c;
  v=iota(c);
  cout << read(&v, c-1);
}
```



A dynamically allocated vector

```
my_int_vec iota(int n)
{
  my_int_vec res={ new int[n] , n };
  for(int i=0; i<n; i++){
    write(&res, i, i);
  }
  return res;
}

int main()
{
  int c;
  my_int_vec v;
  cin >> c;
  v=iota(c);
  cout << read(&v, c-1);
}
```



Automatic vs dynamic allocation

We now have two types of instance allocation

- Automatic : local variables and parameters
- Dynamic : created via new

Each allocation is in one of two conceptual spaces

- Automatic -> ***stack*** : instances reside in the call stack
- Dynamic -> ***heap*** : instances exist outside the call stack

The “call stack” : think of recursive functions and variables

Memory management

Heap lifetimes are manual

```cpp
int main()
{
  int n = 100000;
  for(int i=1; i<=n; i++){
    int *p=new int[i];

  }
}
```

```cpp
int main()
{
  int n = 100000;
  for(int i=1; i<=n; i++){
    vector<int> v;
    v.resize(i);
  }
}
```

p is automatically de-allocated
but not the thing p points too

Memory needed:
~$n^2/2$ integers
~ 20GB

v is automatically de-allocated

Memory needed:
~n integers
~ 400KB

Explicit de-allocation : `delete`

- Dynamic/heap memory management is manual
  - ***You*** asked for it to be allocated
  - ***You*** must explicitly state when it should be de-allocated

Any pointer value that was created with `new[]`

```
delete[] p;
```

De-allocate unnamed instance

De-allocating memory

```
int main()
{
  int n = 100000;
  for(int i=1; i<=n; i++){
    int *p=new int[i];
    delete[] p;
  }
}
```

p is automatically de-allocated

and also what p points to

Memory needed:

~n integers

~ 400KB

```
int main()
{
  int n = 100000;
  for(int i=1; i<=n; i++){
    vector<int> v;
    v.resize(i);
  }
}
```

v is automatically de-allocated

Memory needed:

~n integers

~ 400KB

Deallocating my_int_vec

```
struct my_int_vec{
    int *elements;
    int count;
};

my_int_vec iota(int n)
{
  my_int_vec res={ new[n] , n };
  // ...
  return res;
}

void release(my_int_vec *v)
{
   delete[] v->elements;
}
```

Deallocating my_int_vec

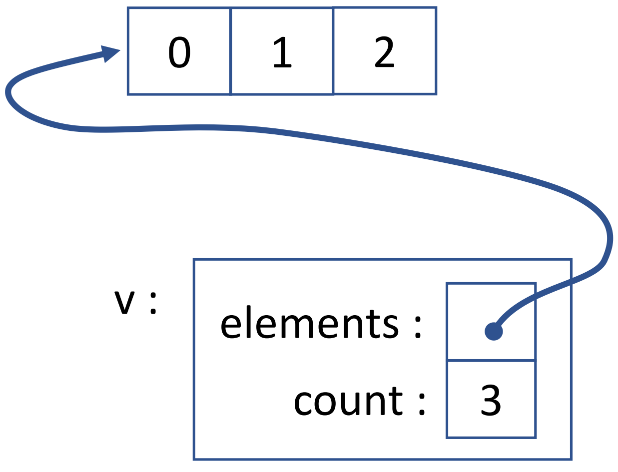
```
struct my_int_vec{
    int *elements;
    int count;
};

my_int_vec iota(int n)
{
  my_int_vec res={ new[n] , n };
  // ...
  return res;
}

void release(my_int_vec *v)
{
   delete[] v->elements;
   v->elements=nullptr;
   v->count=0;
}
```

Potential issues : leaks

```
int main()
{
    my_int_vec v = iota(3);
    // ...
    v = iota(5);
    // ...
    release(&v);
}
```

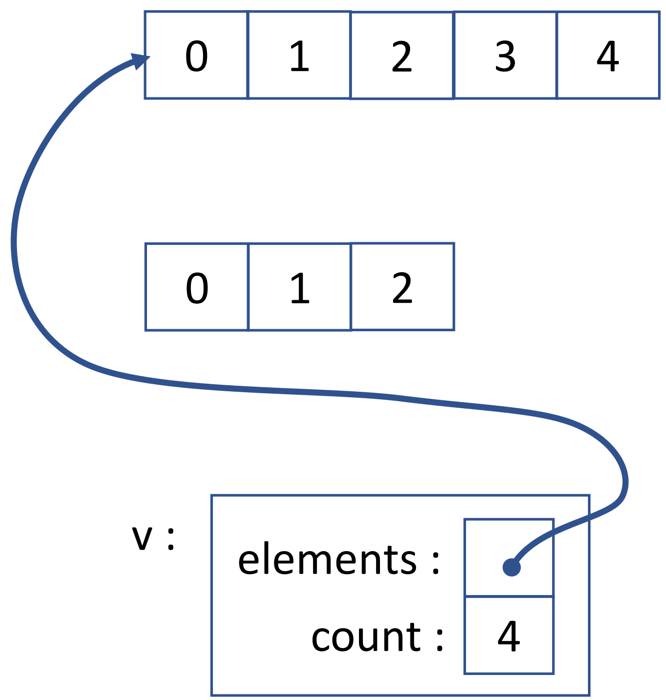
Potential issues : leaks

```c
int main()
{
    my_int_vec v = iota(3);
    // ...
    v = iota(5);
    // ...
    release(&v);
}
```



Potential issues : leaks

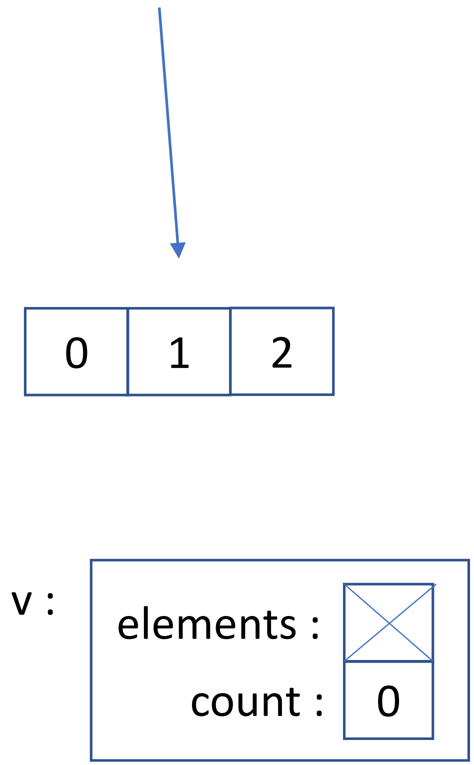
```c
int main()
{
    my_int_vec v = iota(3);
    // ...
    v = iota(5);
    // ...
    release(&v);
}
```



Potential issues : leaks

```
int main()
{
    my_int_vec v = iota(3);
    // ...
    v = iota(5);
    // ...
    release(&v);
}
```

No remaining pointers to first array
Memory has been lost or “leaked”

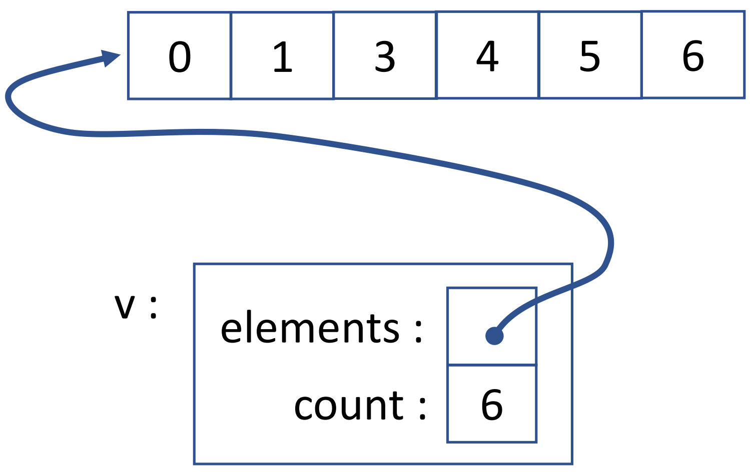


Potential issues : dangling pointers

```
int main()
{
    my_int_vec v = iota(6);
    // ...
    my_int_vec w = v;
    // ...
    release(&v);
    // ...
    write(&w, 0, 10);
}
```

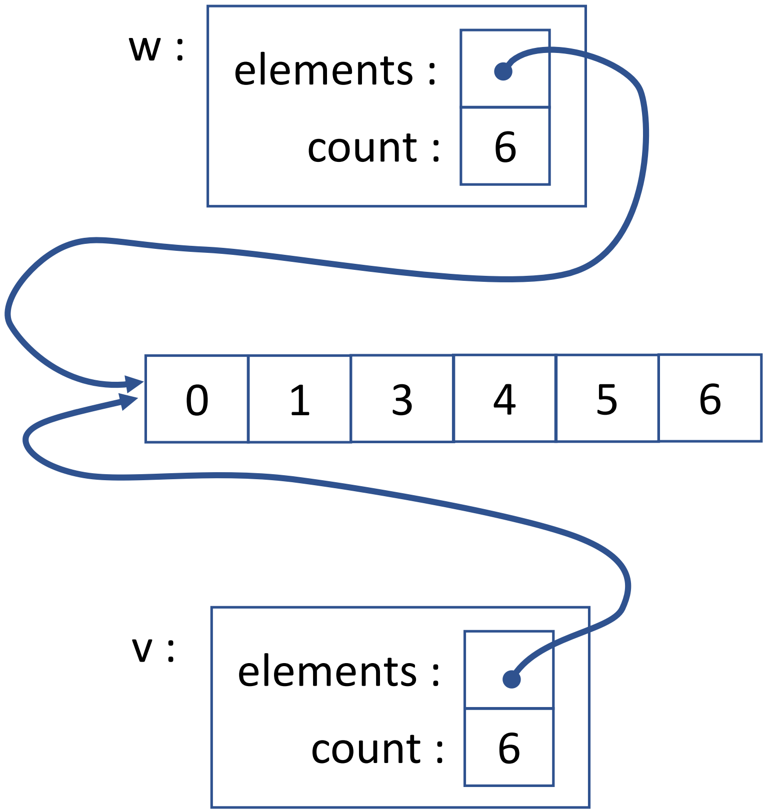
Potential issues : dangling pointers

```
int main()
{
    my_int_vec v = iota(6);
    // ...
    my_int_vec w = v;
    // ...
    release(&v);
    // ...
    write(&w, 0, 10);
}
```



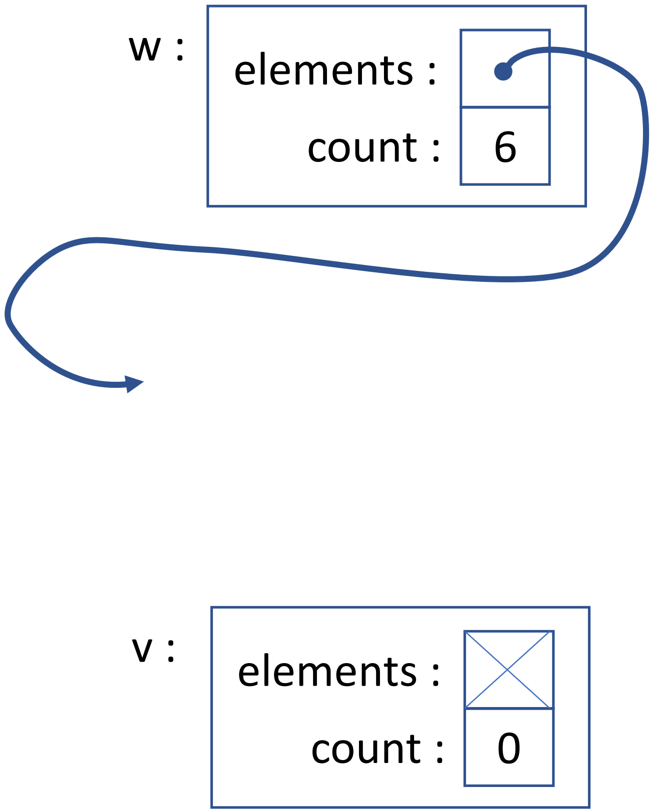
Potential issues : dangling pointers

```
int main()
{
    my_int_vec v = iota(6);
    // ...
    my_int_vec w = v;
    // ...
    release(&v);
    // ...
    write(&w, 0, 10);
}
```



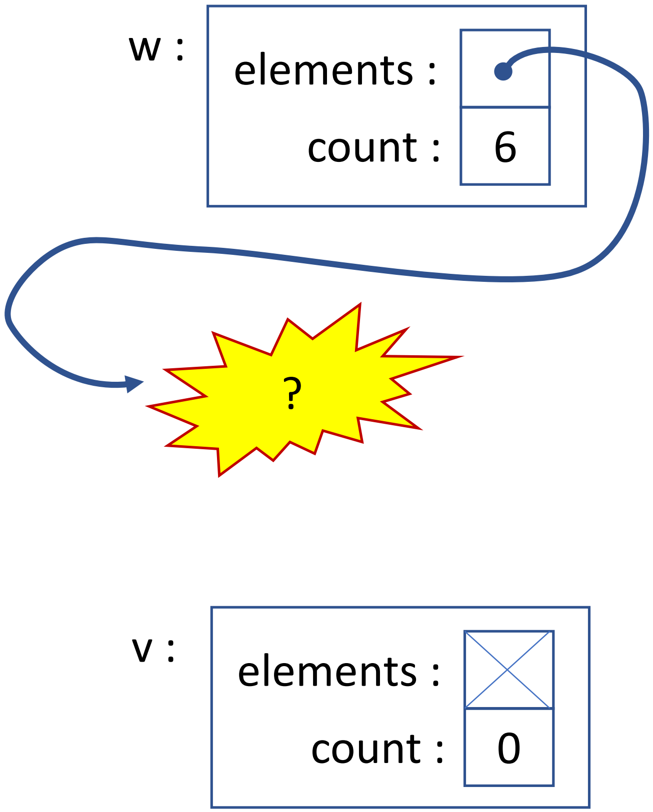
Potential issues : dangling pointers

```
int main()
{
    my_int_vec v = iota(6);
    // ...
    my_int_vec w = v;
    // ...
    release(&v);
    // ...
    write(&w, 0, 10);
}
```



Potential issues : dangling pointers

```
int main()
{
    my_int_vec v = iota(6);
    // ...
    my_int_vec w = v;
    // ...
    release(&v);
    // ...
    write(&w, 0, 10);
}
```



Managing memory management

There are many approaches to memory management

Garbage collection: no explicit delete is needed

- Needs language support: *used in Python, Java, C#,...*

Smart pointers: delete called when last pointer disappears

- Supported in C++ through objects: *we'll see next term*

Handle-based APIs: manage lifetimes through functions

- Classic C-style approach: *widely used in low-level code*

Handle-based APIs

handle : pointer to a data-structure or resource

- A creation function creates the resource
  - Takes parameters and returns a new handle
- All API calls are passed a handle to the resource
  - Identifies the resource to be modified
- A destruction function de-allocates the resource
  - Takes a handle and returns nothing

My vector : creation

```
struct my_int_vec{
    int *elements;
    int count;
};

my_int_vec *create(int size)
{
    // Create a single element of my_int_vec¹
    my_int_vec *res=new my_int_vec[1];

    // Populate with storage and size
    res->elements=new int[size];
    res->count=size;

    // Return the new instance
    return res;
}
```

[1] : Next week we'll see a scalar form of new that makes this slightly more elegant.

My vector : destruction

```
struct my_int_vec{
    int *elements;
    int count;
};
```

```
void destroy(my_int_vec *p)
{
    // Delete the storage
    delete []p->elements;

    // Delete the vector
    delete []p;
}
```

```
void destroy(my_int_vec *p)
{
    // Delete the vector
    delete []p;

    // Delete the storage
    delete []p->elements;
}
```

The complete vector API

```c
struct my_int_vec;

// Create a new vector instance vector
my_int_vec *create(int size);

// Access and modify a vector
int  size( my_int_vec *v);
int  read( my_int_vec *v, int index);
void write(my_int_vec *v, int index, int value);

// Destroy an existing vector
void destroy(my_int_vec *v);
```

We now never create `my_int_vec` directly:

always use create and destroy instead

What we can do know

We’ve hit another milestone of capabilities

- You can now write Linux or Windows
  - Both operating systems are written in this style
- You can now create custom types and data-structures
  - They’ll need to be handle-based for now
  - OOP makes this easier and more powerful later on

- Next: building more complex types and data-structures
  - Linked lists and trees