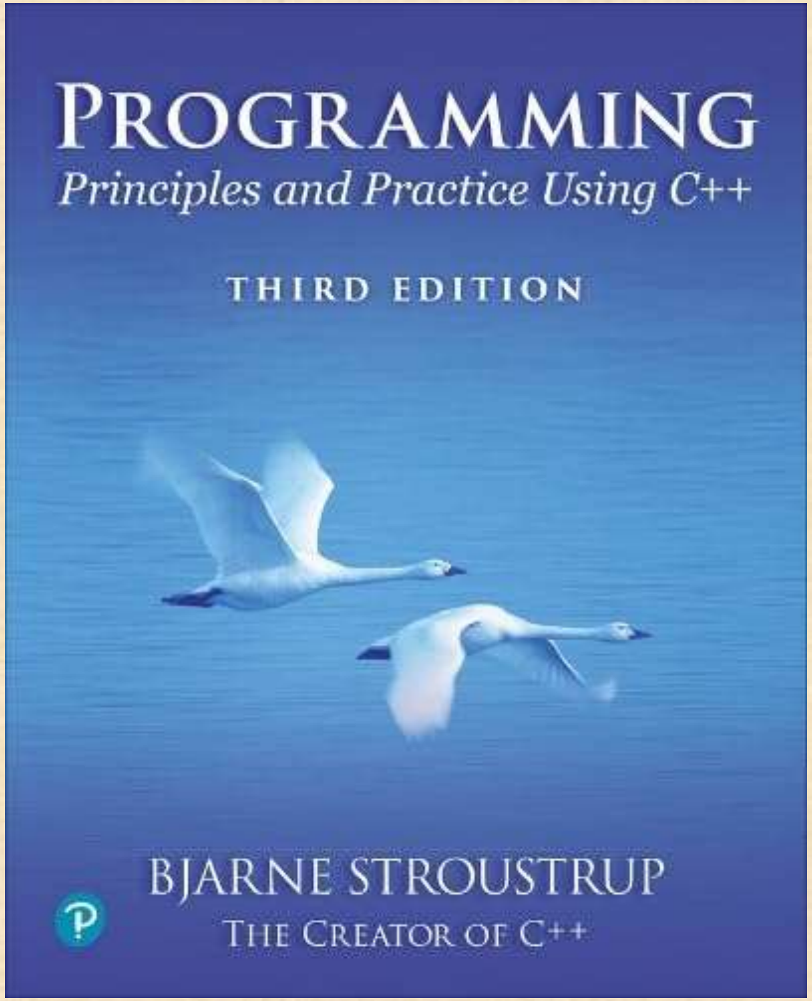
Chapter 17 – Essential Operations

When someone says
I want a programming language in which
I need only say what I wish done,
give him a lollipop.
– Alan Perlis



Overview

- Access to elements
- List initialization
- Copying and moving
- Essential operations
- Other useful operations
- Changing size

Reminder

- Why look at the vector implementation?
  - To see how the standard library vector works
  - To introduce basic concepts and language features
    - Free store (heap)
    - Copying
    - Dynamically growing data structures
    - Defining operators
  - To see how to directly deal with memory
  - To see the techniques and concepts you need to understand C
    - Including the dangerous ones
  - To demonstrate class design techniques
  - To see examples of “neat” code and good design

Vector

```
// a very simplified Vector of doubles (as far as we got in chapter 16):

class Vector {
  int sz;                    // the size
  double* elem;              // pointer to elements
public:
  Vector(int s) :sz(s), elem(new double[s]) { }     // constructor; new allocates memory
  ~Vector() { delete[ ] elem; }                     // destructor; delete[] deallocates memory

  double get(int n) { return elem[n]; }       // access: read
  void set(int n, double v) { elem[n]=v; }    // access: write
  int size() const { return sz; }             // the number of elements
};
```

Access to elements

- This really is too ugly (and not idiomatic)

```
Vector v(3);
v.set(0,1);
v.set(1,2);
v.set(2,3);
int x = v.get(2);
```

- We want

```
Vector v(3);
v[0] = 1;
v[1] = 2;
v[2] = 3
int x = v[2];
```

Access to elements

- Define subscripting, **operator[] ()**, for **Vector**

```
class Vector {
        int sz;                          // the size
        double* elem;                    // a pointer to the elements
public:
        // ...
        double& operator[](int n) { return elem[n]; }                  // return a reference
        const double& operator[](int n) const { return elem[n]; }      // return a const& for a const
};
```

- For the non-**const operator[]()** we must return a reference to allow use on the left-hand size of an assignment

```
        v[1] = v[2];        // means v1.operator[](1) = v.operator[](2);
```

List initialization

- Initialization with a list is most useful
  - **int arr[] = {0,1,2,3,4,5,6,7,8,9};** *// OK*
  - **Vector<int> vec = {0,1,2,3,4,5,6,7,8,9};** *// Not yet. How do we get it?*
- Define a list initializer constructor
  - a **{}**-list is presented to the code as an **initializer_list**

```
class Vector {
        int sz;                      // the size
        double* elem;                // a pointer to the elements
public:
        Vector(initializer_list<double> lst)            // initializer-list constructor
                :sz{lst.end()−lst.begin()},             // number of elements on the list
                elem{new double[sz]}                    //uninitialized memory for elements
        {
                copy(lst.begin(),lst.end(),elem); // initialize using std::copy()
        }
        // …
}
```

A problem

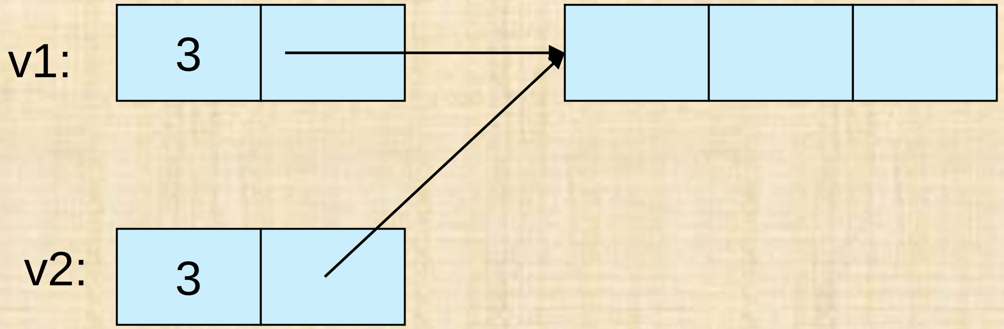
- **Vector** copy doesn't work as we should expect

```
void f(int n)
{
  Vector v1(10);
  Vector v2 = v1;        // what happens here?
                         // what would we like to happen? That v2 becomes a copy of v
  v1[2] = 2.2;
  v2[2] = 3.3;
  if (v1[2]==v2[2])
        error("very odd");     // that's what we get from our still incomplete Vector
}
```

- The standard **vector** has **v1[2]!=v2[2]**
  - but our still-too-simple **Vector** *does not*

Naïve copy initialization (the default)

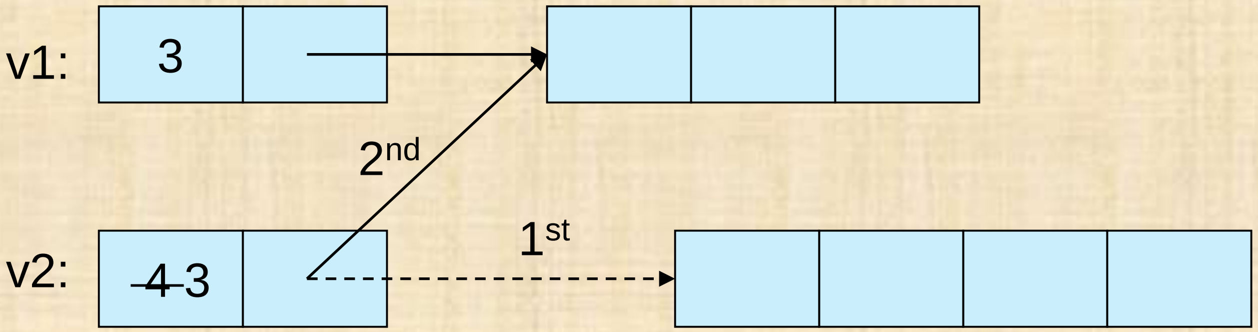
```
void f(int n)
{
  Vector v1(3);
  Vector v2 = v1;        // initialization:
                         // by default, a copy of a class copies its members
                         // so sz and elem are copied, but not the elements
}
```



- Disaster when we leave **f()**!
    - **v1**'s elements are deleted twice (by **Vector**'s destructor)

Naïve copy assignment (the default)

```
void f(int n)
{
  Vector v1(n);
  Vector v2(4);
  v2 = v1;        // assignment :
                  // by default, a copy of a class copies its members
                  // so sz and elem are copied, but not the elements
}
```



- Disaster when we leave **f()**!
  - **v1**’s elements are deleted twice (by **Vector**’s destructor)
  - memory leak: **v2**’s elements are not deleted

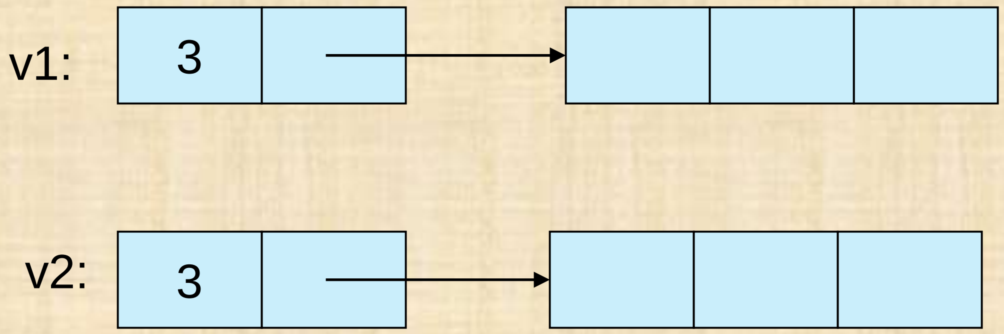
Copy constructor (initialization)

```
class Vector {
  int sz;
  double* elem;
public:
  Vector(const Vector&) ;           // copy constructor: defines copy
  // …
};

Vector::Vector(const Vector& a) // allocate space for elements, then initialize them (by copying)
  :sz(a.sz),  elem(new double[a.sz])
{
  for (int i = 0; i<sz; ++i)
        elem[i] = a.elem[i];
}
```

Copy with copy constructor

```
void f(int n)
{
  Vector v1(n);
  Vector v2 = v1;                    // copy using the copy constructor
                                 // the for loop copies each value from v1 into v2
}
```

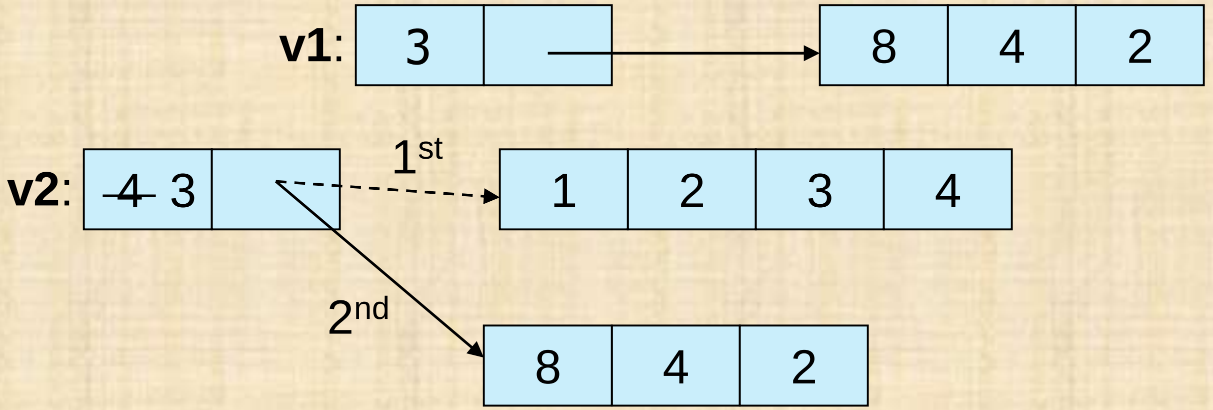


- The copy of a **Vector** is an independent object
  - **Vector**'s destructor correctly deletes all elements (once only)

Copy assignment

```
class Vector {
  int sz;
  double* elem;
public:
  Vector& operator=(const Vector& a);     // copy assignment: define copy (below)
  // ...
};

void f(Vector& v1)
{
  Vector v2 = {1, 2, 3, 4 };
  v2 = v1;
}
```



- Similarly, **Vector**'s operator = must copy elements
  - And remember to delete the old elements

Copy assignment

- Allocate space for copies of elements, copy, then deallocate the old elements

```
Vector& Vector::operator=(const Vector& a) // make this Vector a copy of a
{
        double* p = new double[a.sz];     // allocate new space
        copy(a.elem,a.elem+a.sz,p);       // copy elements [0:sz) from a.elem into p
        delete[] elem;                    // deallocate old space
        elem = p;                         // now we can reset elem and sz
        sz = a.sz;
        return *this;                     // return a self-reference (see §15.8)
}
```

- No leaks

move

- Consider

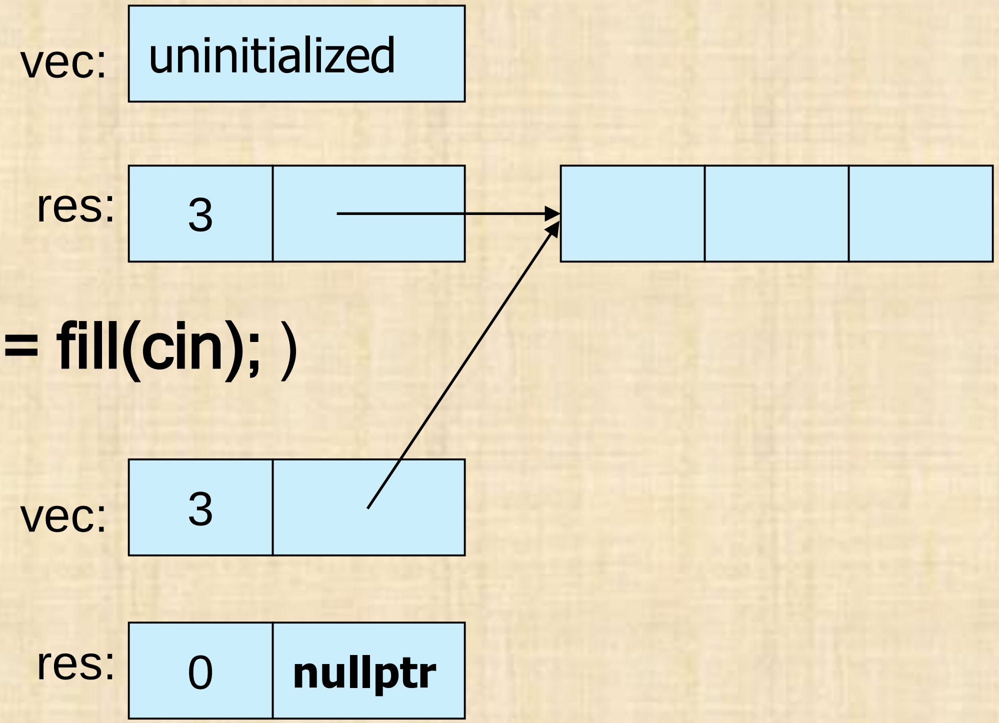
```
Vector fill(istream& is)
{
    Vector res;
    for (double x; is>>x; )
            res.push_back(x);
    return res;        // returning a copy of res could be expensive
                       // returning a copy of res would be silly!
}

void use()
{
    Vector vec = fill(cin);
    // … use vec …
}
```

- But that's what we'd like to write
  - It's the simplest and clearest way to express the this task

What we want: Move

- Before **return res;** in **fill()**
- After **return res;** (after **vector vec = fill(cin);**)



- Functions filling containers are very common and important
  - We need a general solution

Move Constructor and assignment

- Define move operations to "steal" representation

&& indicates "move"

```
class Vector {
	int sz;
	double* elem;
public:
	Vector(vector&& arg);					// move constructor: "steal" arg's elements

	Vector& operator=(Vector&& arg);	// move assignment: destroy target and "steal" arg's elements
	// . . .
};
```

Move implementation

```
vector::vector(vector&& arg)  // move constructor
   :sz{arg.sz}, elem{arg.elem}      // copy a's elem and sz
{
   arg.sz = 0;                 // make arg the empty vector
   arg.elem = nullptr;
}
```

Move implementation

```
Vector& Vector::operator=(Vector&& arg)    // move assignment
{
    if (this!=&arg)      {      // protect against self reference
(e.g., v=v)
        delete[] elem;              // deallocate old space
        elem = arg.elem;  // copy arg's elem and sz
        sz = a.sz;
        a.elem = nullptr; // make arg the empty vector
        a.sz = 0;
    }
    return *this;                         // return a self-reference (see
§15.8)
}
```

We can cheaply return potentially millions of doubles

```
Vector fill(istream& is)
{
	Vector res;
	for (double x; is>>x; )
		res.push_back(x);
	return res;	// move elements, don't copy elements
}

void use()
{
	Vector vec = fill(cin);
	// … use vec …
}
```

- Code generation alternatives:
  - Copy elision: In many cases, the compiler can figure out what we are doing and build **res** right in **vec** (cost: no cost of copying or moving)
  - Return by moving: Use Vector's move constructor (Cost: two word assignments)

Essential operations

- For every class, consider if you need
  - Constructors from one or more arguments
  - Default constructor (§17.5)
  - Copy constructor (copy object of same type; §17.4.1)
  - Copy assignment (copy object of same type; §17.4.2)
  - Move constructor (move object of same type; §17.4.4)
  - Move assignment (move object of same type; §17.4.4)
  - Destructor (§15.5)
- *Rule of zero*: If you don't need to, don't define any essential operation.
- *Rule of all*: if you need to define any essential operation, define them all.

Essential operations

- If it can, the compiler generates constructors, assignments, and destructor for a class
  - It can if all members have those operators

```
struct Club {
    string name;
    vector<Member> members;
};

Club c1;          // default constructor: c1{string{},vector<Member>{}};
Club c2 {"AGF"};       // memberwise construction: c2
{string{"AGF"},vector<Member>{}};
Club c3 {"FCB", Member{a,b,c}, Member{d,e,f}};   // c3 {"FCB",
Member{a,b,c}, Member{d,e,f}};
c1 = c3;          // copy
```

- Club has a memberwise constructor, a destructor, copy and move constructors, and copy and move assignments. All computer

Other useful operations

- Comparison operators, such as **==** and **<** (§17.6.1)

```
bool operator==(const Vector& v1, const Vector& v2)
{
	if (v1.size()!=v2.size())
		return false;
	for (int i = 0; i<v1.size(); ++i)
		if (v1[i]!=v2[i])
			return false;
	return true;
}
```

- **initializer_list** construction and assignment (§17.3)
- Iteration support functions, such as **begin()** and **end()**, as required for range-**for**

```
double* Vector::begin() const { return elem; }
double* Vector::end() const { return elem+sz; }
```

- **swap()** (§7.4.5, §18.4.3))

Other useful operations

- Define operators only with conventional semantics
  - Or confusion and chaos can erupt
- For Vector, define

  ==, **!=**, **<**, **<=**, **>**, and **>=**
- Other operators that you can define for your own types include

  () application/call

  , comma

  << and **>>**

  & bitwise and, **|** bitwise or, **^** bitwise exclusive or, and **~**bitwise complement

  && logical and, and **||** logical or

Changing size

- We would like to be able to change the size of a **Vector;** why?
  - Well, the standard-library **vector** does, but why?
  - If we have a fixed number of elements
    - We must make sure we don't add too many elements
    - If we want more space, we must decide how much, define a new **Vector,** copy the elements, delete the old **Vector,** and make sure we use the new **Vector** and not the old one.
  - To avoid changing the size often (or at all), we often allocate more space than we will ever need
  - And this kind of code is most useful (and common)

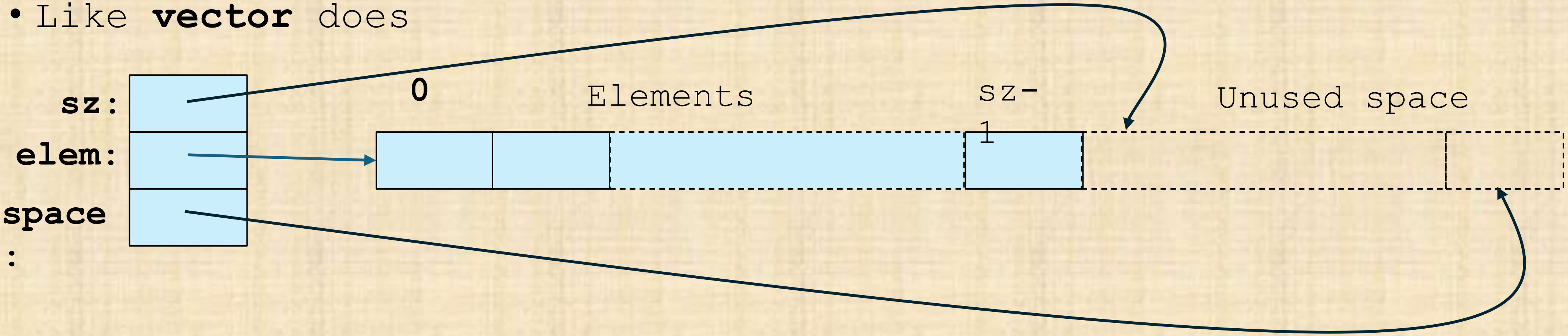
```
Vector v;
for (int x; cin>>x; )
    v.push_back(x);
```

Changing size

- For example, we'd like this to work (efficiently):

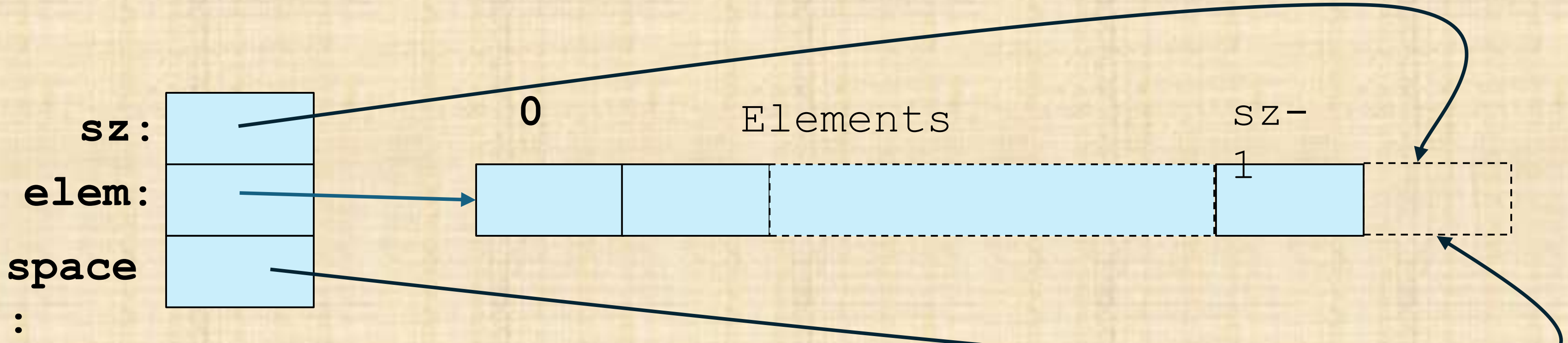
```
Vector v;
for (int x; cin>>x; )
    v.push_back(x);
```

- We change the representation of **Vector** to keep track of some free space
  - Like **vector** does

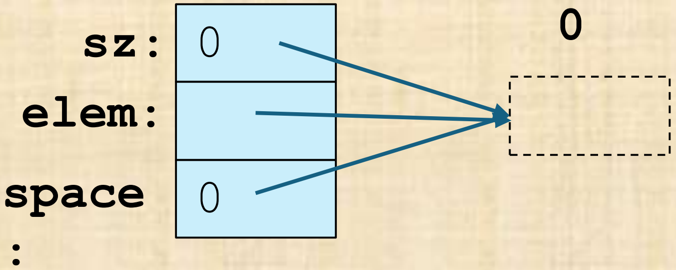


Changing size

- When first created, there is no free/unused space
  - **Vector v1(n);**



- An empty Vector
  - **Vector v2;**



Changing size

- One way to represent such a **Vector**

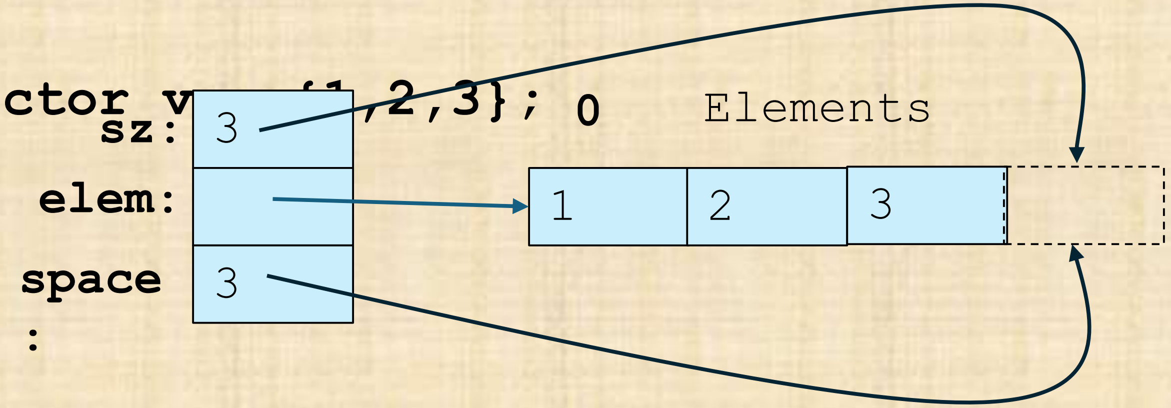
```
class Vector {
public:
	Vector() :sz{0}, elem{nullptr}, space{0} {}
	// ...
private:
	int sz;			// number of elements
	double* elem;	// address of first element
	int space;		// number of elements plus "free space"/"slots" for new elements
};
```

- The empty **Vector** becomes **{0,nullptr,0}**

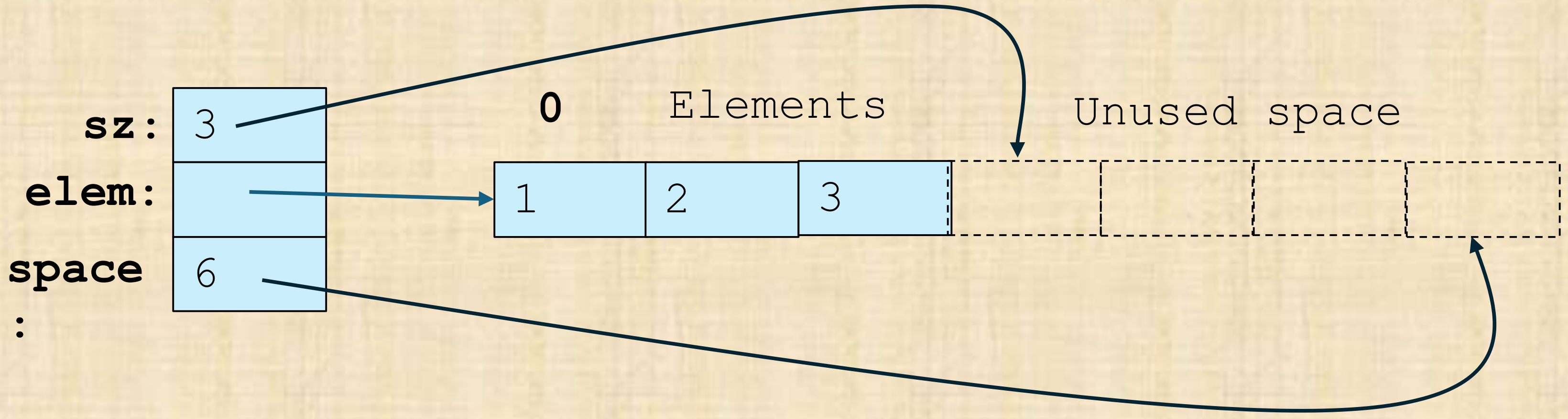
reserve()

- The key operation for relocating elements into a new and larger space

`Vector v = {1,2,3};`



`v.reserve(6);`



reserve()

- The key operation for relocating elements into a new and larger space

```
void Vector::reserve(int newalloc)
    // relocate existing elements into space with room for newalloc elements
{
    if (newalloc<=space)                        // never decrease allocation
        return;
    double* p = new double[newalloc];   // allocate new space
    for (int i=0; i<sz; ++i)                    // copy old elements
        p[i] = elem[i];
    delete[] elem;                              // deallocate old space
    elem = p;
    space = newalloc;
}
```

resize()

- All the hard work is done in **reserve()**

```
void Vector::resize(int newsiz e)
    // make the vector have newsize elements
    // initialize each new element with the default value 0.0
{
    reserve(newsize);
    for (int i=sz; i<newsize; ++i)          // initialize new elements
            elem[i] = 0;
    sz = newsize;
}
```

push_back()

- Given **reserve()**, **push_back()** is very simple

```
void Vector::push_back(double d)
      // increase vector size by one; initialize the new element with d
{
      if (space==0)                               // star t with space for 8 elements
            reserve(8);
      else if (sz==space)
            reserve(2*space);                     // get more space
      elem[sz] = d;                               // add d at end
      ++sz;                                       // increase the size (sz is the
number of elements)
}
```

Assignment

- Assignment can also change the size of a Vector

```
Vector& Vector::operator=(const Vector& a)
     // like the copy constructor, but we must deal with old elements
     // don't copy the free/unused space
{
     double* p = new double[a.sz];                // allocate new space
     for (int i = 0; i<a.sz; ++i)                       // copy elements
           p[i] = a.elem[i];
     delete[] elem;                               // deallocate old space
     space = sz = a.sz;                           // set new size
     elem = p;                              // set new elements
     return *this;                                // return self-reference
}
```

Next lecture

The next lecture completes the design and implementation of the most common and most useful STL container: **vector**.

We show how to specify containers where the element type is a parameter and how to deal with range errors.

The techniques rely on templates and exceptions, so we show how to define templates and give the basic techniques for resource management that are the keys to good use of exceptions.

We discuss the general resource-management technique called ‘‘Resource Acquisition Is Initialization’’ (RAII), and the standard-library resource-management pointers **unique_ptr** and **shared_ptr**.